第29章　TCP的输入(续)

29.1　引言

本章从前一章结束的地方开始，继续介绍 TCP输入处理。回想一下图 28-37中最后的测试条件，如果ACK未置位，输入报文段被丢弃。

本章处理ACK标志，更新窗口信息，处理 URG标志及报文段中携带的所有数据，最后处理FIN标志，如果需要，则调用 `tcp_output`。

29.2　ACK处理概述

在本章中，我们首先讨论 ACK的处理，图29-1给出了ACK处理的框架。SYN_RCVD状态需要特殊处理，紧跟着是其他状态的通用处理代码 (前一章已讨论过在 LISTEN和SYN_SENT状态下收到 ACK时的处理逻辑)。接着是对 TCPS_FIN_WAIT_1、TCPS_CLOSING和TCPS_LAST_ACK状态的一些特殊处理，因为在这些状态下收到 ACK会导致状态的转移。此外，在TIME_WAIT状态下收到ACK还会导致2MSL定时器的重启。

```
switch (tp->t_state) {

case TCPS_SYN_RECEIVED:
    complete processing of passive open and process
        simultaneous open or self-connect;
    /* fall into ... */

case TCPS_ESTABLISHED:
case TCPS_FIN_WAIT_1:
case TCPS_FIN_WAIT_2:
case TCPS_CLOSE_WAIT:
case TCPS_CLOSING:
case TCPS_LAST_ACK:
case TCPS_TIME_WAIT:
    process duplicate ACK;
    update RTT estimators;
    if all outstanding data ACKed, turn off retransmission timer;
    remove ACKed data from socket send buffer;

    switch (tp->t_state) {

    case TCPS_FIN_WAIT_1:
        if (FIN is ACKed) {
            move to FIN_WAIT_2 state;
            start FIN_WAIT_2 timer;
        }
        break;

    case TCPS_CLOSING:
        if (FIN is ACKed) {
```

图29-1　ACK处理框架

```
                move to TIME_WAIT state;
                start TIME_WAIT timer;
            }
            break;

        case TCPS_LAST_ACK:
            if (FIN is ACKed)
                move to CLOSED state;
            break;

        case TCPS_TIME_WAIT:
            restart TIME_WAIT timer;
            goto dropafterack;
        }
    }
```

图29-1 (续)

29.3 完成被动打开和同时打开

图29-2给出了如何处理SYN_RCVD状态下收到的ACK报文段。如前一章中提到过的，这也将完成被动打开(一般情况)，或者是同时打开及自连接(特殊情况)的连接建立过程。

1. 验证收到的ACK

801-806 如果收到的ACK确认了已发送的SYN，它必须大于snd_una (tcp_sendseqinit将snd_una设定为连接的ISS，SYN报文段的序号)，且小于等于snd_max。如果条件满足，则插口进入连接状态ESTABLISHED。

tcp_input.c

```
791     /*
792      * Ack processing.
793      */
794     switch (tp->t_state) {

795         /*
796          * In SYN_RECEIVED state if the ack ACKs our SYN then enter
797          * ESTABLISHED state and continue processing, otherwise
798          * send an RST.
799          */
800     case TCPS_SYN_RECEIVED:
801         if (SEQ_GT(tp->snd_una, ti->ti_ack) ||
802             SEQ_GT(ti->ti_ack, tp->snd_max))
803             goto dropwithreset;
804         tcpstat.tcps_connects++;
805         soisconnected(so);
806         tp->t_state = TCPS_ESTABLISHED;
807         /* Do window scaling? */
808         if ((tp->t_flags & (TF_RCVD_SCALE | TF_REQ_SCALE)) ==
809             (TF_RCVD_SCALE | TF_REQ_SCALE)) {
810             tp->snd_scale = tp->requested_s_scale;
811             tp->rcv_scale = tp->request_r_scale;
812         }
813         (void) tcp_reass(tp, (struct tcpiphdr *) 0, (struct mbuf *) 0);
814         tp->snd_wl1 = ti->ti_seq - 1;
815         /* fall into ... */
```

tcp_input.c

图29-2 tcp_input 函数：在SYN_RCVD 状态收到ACK

在收到三次握手的最后一个报文段后，调用 soisconnected唤醒被动打开的应用进程(一般为服务器)。如果服务器在调用 accept上阻塞，则该调用现在返回。如果服务器调用select等待连接可读，则连接现在已经可读。

2. 查看窗口大小选项

807-812 如果TCP曾发送窗口大小选项，并且收到了对方的窗口大小选项，则在 TCP控制块中保存发送缩放因子和接收缩放因子。另外， TCP控制块中的snd_scale和rcv_scale的默认值为0 (无缩放)。

3. 向应用进程提交队列中的数据

813 现在可以向应用进程提交连接重组队列中的数据，调用 tcp_reass，第二个参数为空。重组队列中的数据可能是SYN报文段中携带的，它同时将连接状态变迁为 SYN_RCVD。

814 snd_wl1等于收到的序号减1，从图29-15可知，这样将导致更新3个窗口变量。

29.4 快速重传和快速恢复的算法

图29-3给出了ACK处理的下一部分代码，处理重复 ACK，并决定是否起用 TCP的快速重传和快速恢复算法[Jacobson 1990c]。两个算法各自独立，但一般都在一起实现 [Floyd 1994]。

- 快速重传算法用于连续出现几次 (一般为3次)重复ACK时，TCP认为某个报文段已丢失并且从中推断出丢失报文段的起始序号，丢失报文段被重传。 RFC 1122中的4.2.2.21节提到了这一算法，建议 TCP收到乱序报文段后，立即发送 ACK。我们看到，在图 27-15中，Net/3正是这样做的。这个算法最早出现在 4.3BSD Tahoe版及后续的Net/1实现中，丢失报文段被重传之后，连接执行慢起动。
- 快速恢复算法认为采用快速重传算法之后 (即丢失报文段已重传)，应执行拥塞避免算法，而非慢起动。这样，如果拥塞不严重，还能保证较大的吞吐量，尤其窗口较大时。这个算法最早出现在4.3BSD Reno版和后续的Net/2实现中。

Net/3同时实现了快速重传和快速恢复算法，下面将做简单介绍。

在图24-17节中，我们提到有效的ACK必须满足下面的不等式：

```
snd_una < 确认字段 <= snd_max
```

第一步只与snd_una做比较，之后在图 29-5中再进行不等式第二部分的比较。分开比较的原因是为了能对收到的ACK完成下列5项测试：

1) 如果确认字段小于等于snd_una；并且

2) 接收报文段长度为0；并且

3) 窗口通告大小未变；并且

4) 连接上部分发送数据未被确认 (重传定时器非零)；并且

5) 接收报文段的确认字段是TCP收到的最大的确认序号 (确认字段等于snd_una)。

之后可确认报文段是完全重复的 ACK(测试项1、2和3在图29-3中，测试4和5在图29-4的起始处)。

TCP统计连续收到的重复 ACK的个数，保存在变量 t_dupacks中，次数超过门限(tcprexmtthresh，3)时，丢失报文段被重传。这也就是卷 1第21.7节中介绍的快速重传算法。它与图27-15中的代码互相配合：当TCP收到乱序报文段时，立即生成一个重复的 ACK，

告诉对端报文段有可能丢失和等待接收的下一个序号值。快速重传算法是为了让 TCP立即重传看上去已经丢失的报文段，而不是被动地等待重传定时器超时。卷 1第21.7节举例详细说明了这个算法是如何工作的。

tcp_input.c

```
816         /*
817          * In ESTABLISHED state: drop duplicate ACKs; ACK out-of-range
818          * ACKs.  If the ack is in the range
819          *  tp->snd_una < ti->ti_ack <= tp->snd_max
820          * then advance tp->snd_una to ti->ti_ack and drop
821          * data from the retransmission queue.  If this ACK reflects
822          * more up-to-date window information we update our window information.
823          */
824     case TCPS_ESTABLISHED:
825     case TCPS_FIN_WAIT_1:
826     case TCPS_FIN_WAIT_2:
827     case TCPS_CLOSE_WAIT:
828     case TCPS_CLOSING:
829     case TCPS_LAST_ACK:
830     case TCPS_TIME_WAIT:

831         if (SEQ_LEQ(ti->ti_ack, tp->snd_una)) {
832             if (ti->ti_len == 0 && tiwin == tp->snd_wnd) {
833                 tcpstat.tcps_rcvdupack++;
834                 /*
835                  * If we have outstanding data (other than
836                  * a window probe), this is a completely
837                  * duplicate ack (ie, window info didn't
838                  * change), the ack is the biggest we've
839                  * seen and we've seen exactly our rexmt
840                  * threshold of them, assume a packet
841                  * has been dropped and retransmit it.
842                  * Kludge snd_nxt & the congestion
843                  * window so we send only this one
844                  * packet.
845                  *
846                  * We know we're losing at the current
847                  * window size so do congestion avoidance
848                  * (set ssthresh to half the current window
849                  * and pull our congestion window back to
850                  * the new ssthresh).
851                  *
852                  * Dup acks mean that packets have left the
853                  * network (they're now cached at the receiver)
854                  * so bump cwnd by the amount in the receiver
855                  * to keep a constant cwnd packets in the
856                  * network.
857                  */
```

tcp_input.c

图29-3 tcp_input 函数：判定完全重复的ACK报文段

另一方面，重复ACK的接收方也能确认某个数据分组已“离开了网络”，因为对端已收到了一个乱序报文段，从而开始发送重复的 ACK。快速恢复算法要求连续收到几个重复 ACK后，TCP应该执行拥塞避免算法(如降低速度)，而不一定必需等待连接两端间的管道清空 (慢起动)。“离开了网络”指数据分组已被对端接收，并加入到连接的重组队列中，不再滞留在传输途中。

如果前述5项测试条件只有前3项为真，说明ACK是重复报文段，统计值tcps_rcvdupack加1，而连续重复ACK计数器(t_dupacks)复位为0。如果仅有第一项测试条件为真，则计数器t_dupacks复位为0。

图29-4给出了快速重传算法其余的代码，当所有5个测试条件全部满足时，根据已连续收到的重复ACK数目的不同，运用快速重传算法处理收到的报文段。

1) t_dupacks等于3(tcprexmtthresh)，则执行拥塞避免算法，并重传丢失报文段。

2) t_dupacks大于3，则增大拥塞窗口，执行正常的TCP输出。

3) t_dupacks小于3，不做处理。

tcp_input.c

```
858                 if (tp->t_timer[TCPT_REXMT] == 0 ||
859                     ti->ti_ack != tp->snd_una)
860                     tp->t_dupacks = 0;
861                 else if (++tp->t_dupacks == tcprexmtthresh) {
862                     tcp_seq onxt = tp->snd_nxt;
863                     u_int win =
864                         min(tp->snd_wnd, tp->snd_cwnd) / 2 /
865                             tp->t_maxseg;
866                     if (win < 2)
867                         win = 2;
868                     tp->snd_ssthresh = win * tp->t_maxseg;
869                     tp->t_timer[TCPT_REXMT] = 0;
870                     tp->t_rtt = 0;
871                     tp->snd_nxt = ti->ti_ack;
872                     tp->snd_cwnd = tp->t_maxseg;
873                     (void) tcp_output(tp);
874                     tp->snd_cwnd = tp->snd_ssthresh +
875                         tp->t_maxseg * tp->t_dupacks;
876                     if (SEQ_GT(onxt, tp->snd_nxt))
877                         tp->snd_nxt = onxt;
878                     goto drop;
879                 } else if (tp->t_dupacks > tcprexmtthresh) {
880                     tp->snd_cwnd += tp->t_maxseg;
881                     (void) tcp_output(tp);
882                     goto drop;
883                 }
884             } else
885                 tp->t_dupacks = 0;
886             break;              /* beyond ACK processing (to step 6) */
887         }
```

tcp_input.c

图29-4 tcp_input 函数：处理重复的ACK

1. 连续收到的重复ACK次数已达到门限值3

861-868 t_dupacks等于3(tcprexmtthresh)时，在变量onxt中保存snd_nxt值，令慢起动门限(ssthresh)等于当前拥塞窗口大小的一半，最小值为两个最大报文段长度。这与图25-27中重传定时器超时处理中的慢起动门限设定操作类似，但我们将看到，超时处理中把拥塞窗口设定为一个最大报文段长度，快速重传算法并不这样做。

2. 关闭重传定时器

869-870 关闭重传定时器。为防止TCP正对某个报文段计时，t_rtt清零。

3. 重传缺失报文段

871-873 从连续收到的重复ACK报文段中可判断出丢失报文段的起始序号(重复ACK的确认字段)，将其赋给snd_nxt，并将拥塞窗口设定为一个最大报文段长度，从而tcp_output将只发送丢失报文段(参见卷1的图21-7中的63号报文段)。

4. 设定拥塞窗口

874-875 拥塞窗口等于慢起动门限加上对端高速缓存的报文段数。“高速缓存”指对端已收到的乱序报文段数，且为这些报文段发送了重复的ACK。除非对端收到了丢失的报文段(刚刚发送)，这些缓存报文段中的数据不会被提交给应用进程。卷1的图21-10和图21-11给出了快速重传算法起作用时,拥塞窗口和慢起动门限的变化情况。

5. 设定snd_nxt

876-878 比较下一发送序号(snd_nxt)的先前值(onxt)和当前值，将两者中最大的一个重新赋还给snd_nxt，因为重传报文段时，tcp_output会改变snd_nxt。一般情况下，snd_nxt将等于原来保存的值，意味着只有丢失报文段被重传，下一次调用tcp_output时，将继续发送序列中的下一报文段。

6. 连续收到的重复ACK数超过门限3

879-883 因为t_dupacks等于3时，已重传了丢失的报文段，再次收到重复ACK说明又有另一个报文段离开了网络。拥塞窗口大小加1，调用tcp_output发送序列中的下一报文段，并丢弃重复的ACK(参见卷1的图21-7的67号、69号和71号报文段)。

884-885 如果收到的报文段中带有重复的ACK，且长度非零或者通告窗口大小发生变化，则执行这些语句。此时，前面提到的5个测试条件中只有第一个为真，连续收到的重复ACK数被清零。

7. 略过ACK处理的其余部分

886 break语句在下列3种情况下被执行：(1)前述5个测试条件中只有第一个条件为真；(2)只有前3个条件为真；(3)重复ACK次数小于门限值3。任何一种情况下，尽管收到的是重复ACK，将执行break语句，控制跳到图29-2中switch语句的结尾处，在标注step6处继续执行。

为了理解前面的窗口操作步骤，请看下面的例子。假定对端接收窗口只能容纳8个报文段，而本地报文段1~8已发送。报文段1丢失，其余报文段均正常到达且被确认。收到对报文段2、3和4的确认后，重传丢失的报文段(1)。尽管在收到后续的对报文段5~8的确认后，TCP希望能够发送报文段9，以保证高的吞吐率。但窗口大小等于8，禁止发送报文段9及其后续报文段。因此，每当再次收到一个重复的ACK，就暂时把拥塞窗口加1，因为收到重复的ACK告诉TCP又有一个报文段已在对端离开了网络。最终收到对报文段1的确认后，下面将介绍的代码会减少拥塞窗口大小，令其等于慢起动门限。卷1的图21-10举例说明了这一过程，重复ACK到达时，增加拥塞窗口大小，之后收到新的ACK时，再相应地减少拥塞窗口。

29.5 ACK处理

图29-5中的代码继续处理ACK。

tcp_input.c

```
888         /*
889          * If the congestion window was inflated to account
890          * for the other side's cached packets, retract it.
891          */
892         if (tp->t_dupacks > tcprexmtthresh &&
893             tp->snd_cwnd > tp->snd_ssthresh)
894             tp->snd_cwnd = tp->snd_ssthresh;
895         tp->t_dupacks = 0;

896         if (SEQ_GT(ti->ti_ack, tp->snd_max)) {
897             tcpstat.tcps_rcvacktoomuch++;
898             goto dropafterack;
899         }
900         acked = ti->ti_ack - tp->snd_una;
901         tcpstat.tcps_rcvackpack++;
902         tcpstat.tcps_rcvackbyte += acked;
```

tcp_input.c

图29-5 tcp_input 函数：继续ACK处理

1. 调整拥塞窗口

888-895 如果连续收到的重复ACK数超过了门限值3，说明这是在收到了4个或4个以上的重复ACK后，收到的第一个非重复的ACK。快速重传算法结束。因为从收到的第4个重复ACK开始，每收到一个重复ACK就会导致拥塞窗口加1，如果它已超过了慢起动门限，令其等于慢起动门限。连续收到的重复ACK计数器清零。

2. 检查ACK的有效性

896-899 前面介绍过，有效的ACK必须满足下列不等式：

```
snd_una < 确认字段 <= snd_max
```

如果确认字段大于snd_max，可对端正在确认了TCP尚未发送的数据。可能的原因是，对于高速连接，某个失踪的ACK再次出现时，序号已回绕，从图24-5可知，这是极为罕见的(因为实际的网络不可能那么快)。

3. 计算确认的字节数

900-902 经过前面的测试，已知这是一个有效的ACK。acked等于确认的字节数。

图29-6给出了ACK处理的下一部分代码，完成RTT测算和重传定时器的操作。

4. 更新RTT测算值

903-915 如果(1)时间戳选项存在；或者(2)TCP对某个报文段计时，且收到的确认字段大于该报文段的起始序号，则调用tcp_xmit_timer更新RTT测算值。注意，使用时间戳时，tcp_xmit_timer的第二个参数等于当前时间(tcp_now)减去收到的时间戳回显(ts_ecr)加1(因为函数处理中减了1)。

由于延迟ACK的存在，在前面的测试不等式中应采用大于号。例如，假定TCP发送了一个报文段，携带字节1~1024，并对其计时，接着又发送了一个报文段，携带字节1025~2048。如果收到的确认字段等于2049，因为2049大于1(计时报文段的起始序号)，TCP将更新RTT测算值。

5. 是否确认了所有已发送数据

916-924 如果收到报文段的确认字段(ti_ack)等于TCP的最大发送序号(snd_max)，说明所有已发送数据都已被确认。关闭重传定时器，并置位needoutput标志，从而在函数结束

时强迫调用tcp_output。这是因为在此之前，有可能因为发送窗口已满， TCP拒绝了等待发送的数据，而现在收到了新的 ACK，确认了全部已发送数据，发送窗口能够向右移动 (图29-8中的snd_una被更新)，允许发送更多的数据。

tcp_input.c

```
903         /*
904          * If we have a timestamp reply, update smoothed
905          * round-trip time.  If no timestamp is present but
906          * transmit timer is running and timed sequence
907          * number was acked, update smoothed round-trip time.
908          * Since we now have an rtt measurement, cancel the
909          * timer backoff (cf., Phil Karn's retransmit alg.).
910          * Recompute the initial retransmit timer.
911          */
912         if (ts_present)
913             tcp_xmit_timer(tp, tcp_now - ts_ecr + 1);
914         else if (tp->t_rtt && SEQ_GT(ti->ti_ack, tp->t_rtseq))
915             tcp_xmit_timer(tp, tp->t_rtt);

916         /*
917          * If all outstanding data is acked, stop retransmit
918          * timer and remember to restart (more output or persist).
919          * If there is more data to be acked, restart retransmit
920          * timer, using current (possibly backed-off) value.
921          */
922         if (ti->ti_ack == tp->snd_max) {
923             tp->t_timer[TCPT_REXMT] = 0;
924             needoutput = 1;
925         } else if (tp->t_timer[TCPT_PERSIST] == 0)
926             tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
```

tcp_input.c

图29-6 tcp_input 函数：RTT测算值和重传定时器

6. 存在未确认的数据

925-926 由于发送缓存中还存在未被确认的数据，如果持续定时器未设定，则启动重传定时器，时限等于t_rxtcur的当前值。

Karn算法和时间戳

注意，时间戳的运用取消了 Karn算法的部分规定 (卷1的21.3节)：如果重传定时器超时，则报文段被重传，收到对重传报文段的确认时，不应据此更新 RTT测算值(重传确认的二义性问题)。在图25-26中，我们看到当发生重传时，遵从 Karn算法， t_rtt被设为0。如果时间戳不存在，且收到的是对重传报文段的确认，则图 29-6中的代码不会更新RTT测算值，因为此时t_rtt等于0。但如果时间戳存在，则不查看 t_rtt值，允许利用收到的时间戳回显字段更新RTT测算值。根据RFC 1323，时间戳的运用不存在二义性，因为 ts_ecr的值复制自被确认的报文段。Karn算法中关于重传报文段时应采用指数退避的策略依旧有效。

图29-7给出了ACK处理的下一部分代码，更新拥塞窗口。

tcp_input.c

```
927         /*
928          * When new data is acked, open the congestion window.
929          * If the window gives us less than ssthresh packets
```

图29-7 tcp_input 函数：响应收到的ACK，打开拥塞窗口

```
930             * in flight, open exponentially (maxseg per packet).
931             * Otherwise open linearly: maxseg per window
932             * (maxseg^2 / cwnd per packet), plus a constant
933             * fraction of a packet (maxseg/8) to help larger windows
934             * open quickly enough.
935             */
936            {
937                u_int    cw = tp->snd_cwnd;
938                u_int    incr = tp->t_maxseg;

939                if (cw > tp->snd_ssthresh)
940                    incr = incr * incr / cw + incr / 8;
941                tp->snd_cwnd = min(cw + incr, TCP_MAXWIN << tp->snd_scale);
942            }
```

tcp_input.c

图29-7 (续)

1. 更新拥塞窗口

927-942 慢起动和拥塞避免的一条原则是收到ACK后将增大拥塞窗口。默认情况下，每收到一个ACK(慢起动)，拥塞窗口将加1。但如果当前拥塞窗口大于慢起动门限，增加值等于1除以拥塞窗口大小，并加上一个常量。表达式

```
incr * incr / cw
```

等于

```
t_maxseg * t_maxseg / snd_cwnd
```

即1除以拥塞窗口，因为snd_cwnd的单位为字节，而非报文段。表达式的常量部分等于最大报文段长度的1/8。此外，拥塞窗口的上限等于连接发送窗口的最大值。算法的举例参见卷1的21.8节。

添加一个常量(最大报文段长度的1/8)是错误的[Floyd 1994]。但它一直存在于BSD源码中，从4.3BSD到4.4BSD和Net/3，应将其删除。

图29-8给出了tcp_input下一部分的代码，从发送缓存中删除已确认的数据。

tcp_input.c

```
943             if (acked > so->so_snd.sb_cc) {
944                 tp->snd_wnd -= so->so_snd.sb_cc;
945                 sbdrop(&so->so_snd, (int) so->so_snd.sb_cc);
946                 ourfinisacked = 1;
947             } else {
948                 sbdrop(&so->so_snd, acked);
949                 tp->snd_wnd -= acked;
950                 ourfinisacked = 0;
951             }
952             if (so->so_snd.sb_flags & SB_NOTIFY)
953                 sowwakeup(so);
954             tp->snd_una = ti->ti_ack;
955             if (SEQ_LT(tp->snd_nxt, tp->snd_una))
956                 tp->snd_nxt = tp->snd_una;
```

tcp_input.c

图29-8 tcp_input 函数：从发送缓存中删除已确认的数据

2. 从发送缓存中删除已确认的字节

943-946 如果确认字节数超过发送缓存中的字节数，则从snd_wnd中减去发送缓存中的字

节数，并且可知本地发送的FIN已被确认。调用sbdrop从发送缓存中删除所有字节。能够以这种方式检查对FIN报文段的确认，是因为FIN在序号空间中只占一个字节。

947-951 如果确认字节数小于或等于发送缓存中的字节数，ourfinisacked等于0，并从发送缓存中丢弃acked字节的数据。

3. 唤醒等待发送缓存的进程

951-956 调用sowwakeup唤醒所有等待发送缓存的应用进程，更新snd_una保存最老的未被确认的序号。如果snd_una的新值超过了snd_nxt，则更新后者，因为这说明中间的数据也被确认。

图29-9举例说明了为什么snd_nxt保存的序号有可能小于snd_una。假定传输了两个报文段，第一个携带字节1~512，而第二个携带字节513~1024。

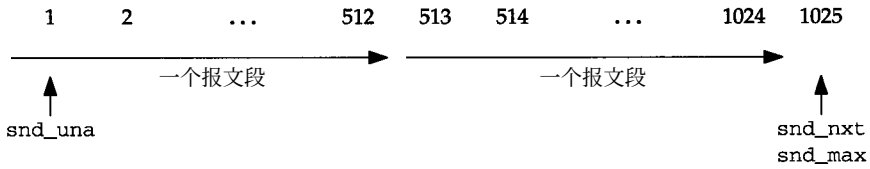

图29-9 连接上发送了两个报文段

确认返回前，重传定时器超时。图 25-26中的代码将snd_nxt设定为snd_una，进入慢起动状态，调用tcp_output重传携带1~512字节的报文段。tcp_output将snd_nxt增加为513，如图29-10所示。

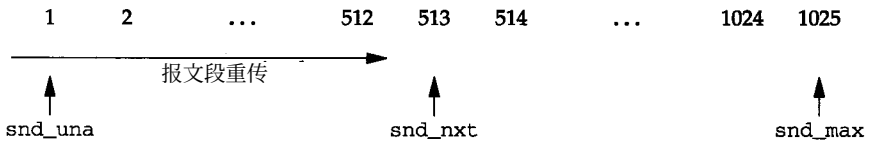

图29-10 重传定时器超时后的连接(接图29-9)

此时，确认字段等于1025的ACK到达(或者是最初发送的两个报文段或者是ACK在网络中被延迟)。这个ACK是有效的，因为它小于等于snd_max，但它也将小于更新后的snd_una值。

一般性的ACK处理现在已结束，图29-11中的switch语句接着处理了4种特殊情况。

tcp_input.c

```
957         switch (tp->t_state) {

958             /*
959              * In FIN_WAIT_1 state in addition to the processing
960              * for the ESTABLISHED state if our FIN is now acknowledged
961              * then enter FIN_WAIT_2.
962              */
963         case TCPS_FIN_WAIT_1:
964             if (ourfinisacked) {
965                 /*
966                  * If we can't receive any more
967                  * data, then closing user can proceed.
968                  * Starting the timer is contrary to the
```

图29-11 tcp_input 函数：在FIN_WAIT_1状态时收到了ACK

```
969                  * specification, but if we don't get a FIN
970                  * we'll hang forever.
971                  */
972                 if (so->so_state & SS_CANTRCVMORE) {
973                     soisdisconnected(so);
974                     tp->t_timer[TCPT_2MSL] = tcp_maxidle;
975                 }
976                 tp->t_state = TCPS_FIN_WAIT_2;
977             }
978             break;
```

tcp_input.c

图29-11 (续)

4. 在FIN_WAIT_1状态时收到了ACK

958-971 此时，应用进程已关闭了连接，TCP已发送了FIN，但还有可能收到对在FIN之前发送的报文段的确认。因此，只有在收到FIN的确认后，连接才会转移到FIN_WAIT_2状态。图29-8中，ourfinisacked标志已置位，这取决于确认的字节数是否超过发送缓存中的数据量。

5. 设定FIN_WAIT_2定时器

972-975 我们在25.6节中介绍了Net/3如何设定FIN_WAIT_2定时器，以防止在FIN_WAIT_2状态无限等待。只有当应用进程完全关闭了连接(如close系统调用，或者在应用进程被某个信号量终止时与close类似的内核调用)，而不是半关闭时(如已发送了FIN，但应用进程仍在连接上接收数据)，定时器才会启动。

图29-12给出了在CLOSING状态收到ACK时的处理代码。

tcp_input.c

```
979             /*
980              * In CLOSING state in addition to the processing for
981              * the ESTABLISHED state if the ACK acknowledges our FIN
982              * then enter the TIME-WAIT state, otherwise ignore
983              * the segment.
984              */
985         case TCPS_CLOSING:
986             if (ourfinisacked) {
987                 tp->t_state = TCPS_TIME_WAIT;
988                 tcp_canceltimers(tp);
989                 tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
990                 soisdisconnected(so);
991             }
992             break;
```

tcp_input.c

图29-12 tcp_input 函数：在CLOSING状态收到ACK

6. 在CLOSING状态收到ACK

979-992 如果收到的ACK是对FIN的确认(而非之前发送的数据报文段)，则连接转移到TIME_WAIT状态。所有等待的定时器都被清除(如等待的重传定时器)，TIME_WAIT定时器被启动，时限等于两倍的MSL。

图29-13给出了在LAST_ACK状态收到ACK的处理代码。

tcp_input.c

```
 993             /*
 994              * In LAST_ACK, we may still be waiting for data to drain
 995              * and/or to be acked, as well as for the ack of our FIN.
 996              * If our FIN is now acknowledged, delete the TCB,
 997              * enter the closed state, and return.
 998              */
 999         case TCPS_LAST_ACK:
1000             if (ourfinisacked) {
1001                 tp = tcp_close(tp);
1002                 goto drop;
1003             }
1004             break;
```

tcp_input.c

图29-13 tcp_input 函数：在LAST_ACK状态收到ACK

7. 在LAST_ACK状态收到ACK

993-1004 如果FIN已确认，连接将转移到CLOSED状态。tcp_close将负责这一状态变迁，并同时释放Internet PCB和TCP控制块。

图29-14给出了在TIME_WAIT状态收到ACK的处理代码。

tcp_input.c

```
1005             /*
1006              * In TIME_WAIT state the only thing that should arrive
1007              * is a retransmission of the remote FIN.  Acknowledge
1008              * it and restart the finack timer.
1009              */
1010         case TCPS_TIME_WAIT:
1011             tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1012             goto dropafterack;
1013         }
1014     }
```

tcp_input.c

图29-14 tcp_input 函数：在TIME_WAIT状态收到ACK

8. 在TIME_WAIT状态收到ACK

1005-1014 此时，连接两端都已发送过FIN，且两个FIN都已被确认。但如果TCP对远端FIN的确认丢失，对端将重传FIN(带有ACK)。TCP丢弃报文段并重传ACK。此外，TIME_WAIT定时器必须被重传，时限等于两倍的MSL。

29.6 更新窗口信息

TCP控制块中还有两个窗口变量我们未曾提及：snd_wl1和snd_wl2。

- nd_wl1记录最后接收报文段的序号，用于更新发送窗口(snd_wnd)。
- snd_wl2记录最后接收报文段的确认序号，用于更新发送窗口。

到目前为止，只在连接建立时(主动打开、被动打开或同时打开)遇到过这两个变量，snd_wl1被设定为ti_seq减1。当时说是为了保证窗口更新，下面的代码将证明这一点。

如果下列3个条件中的任一个被满足，则应根据接收报文段中的通告窗口值(tiwin)更新发送窗口(snd_wnd)：

1) 报文段携带了新数据。因为snd_wl1保存了用于更新窗口的最后接收报文段的起始序

号，如果snd_wl1<ti_seq，说明此条件为真。

2) 报文段未携带新数据(snd_wl1等于ti_seq)，但报文段确认了新数据。因为snd_wl2保存了用于更新窗口的最后接收报文段的确认序号，如果 snd_wl2<ti_ack，说明此条件为真。

3) 报文段未携带新数据，也未确认新数据，但通告窗口大于当前发送窗口。

这些测试条件的目的是为了防止旧的报文段影响发送窗口，因为发送窗口并非绝对的序号序列，而是从snd_una算起的偏移量。

图29-15给出了更新发送窗口的代码。

tcp_input.c

```
1015    step6:
1016      /*
1017       * Update window information.
1018       * Don't look at window if no ACK: TAC's send garbage on first SYN.
1019       */
1020      if ((tiflags & TH_ACK) &&
1021          (SEQ_LT(tp->snd_wl1, ti->ti_seq) || tp->snd_wl1 == ti->ti_seq &&
1022           (SEQ_LT(tp->snd_wl2, ti->ti_ack) ||
1023            tp->snd_wl2 == ti->ti_ack && tiwin > tp->snd_wnd))) {
1024          /* keep track of pure window updates */
1025          if (ti->ti_len == 0 &&
1026              tp->snd_wl2 == ti->ti_ack && tiwin > tp->snd_wnd)
1027              tcpstat.tcps_rcvwinupd++;
1028          tp->snd_wnd = tiwin;
1029          tp->snd_wl1 = ti->ti_seq;
1030          tp->snd_wl2 = ti->ti_ack;
1031          if (tp->snd_wnd > tp->max_sndwnd)
1032              tp->max_sndwnd = tp->snd_wnd;
1033          needoutput = 1;
1034      }
```

tcp_input.c

图29-15 tcp_input 函数：更新窗口信息

1. 是否需要更新发送窗口

1015-1023 if语句检查报文段的ACK标志是否置位，且前述3个条件中是否有一个被满足。前面介绍过，在LISTEN状态或SYN_SENT状态收到SYN后，控制将跳转到 step6，而在LISTEN状态收到的SYN不带ACK。

注释中的TAC指“终端接入控制器(terminal access controller)”，是ARPANET上的Telnet客户。

1024-1027 如果收到一个纯窗口更新报文段(长度为0，ACK未确认新数据，但通告窗口增加)，统计值tcps_rcvwinupd递增。

2. 更新变量

1028-1033 更新发送窗口，保存新的snd_wl1和snd_wl2值。此外，如果新的通告窗口是TCP从对端收到的所有窗口通告中的最大值，则新值被保存在 max_sndwnd中。这是为了猜测对端接收缓存的大小，在图 26-8中用到了此变量。更新 snd_wnd后，发送窗口可用空间增加，从而能够发送新的报文段，因此， needoutput标志置位。

29.7 紧急方式处理

TCP输入处理的下一部分是URG标志置位时的报文段。如图29-16所示。

tcp_input.c

```
1035     /*
1036      * Process segments with URG.
1037      */
1038     if ((tiflags & TH_URG) && ti->ti_urp &&
1039         TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1040         /*
1041          * This is a kludge, but if we receive and accept
1042          * random urgent pointers, we'll crash in
1043          * soreceive.  It's hard to imagine someone
1044          * actually wanting to send this much urgent data.
1045          */
1046         if (ti->ti_urp + so->so_rcv.sb_cc > sb_max) {
1047             ti->ti_urp = 0;         /* XXX */
1048             tiflags &= ~TH_URG;     /* XXX */
1049             goto dodata;            /* XXX */
1050         }
```

tcp_input.c

图29-16 tcp_input 函数：紧急方式的处理

1. 是否需要处理URG标志

1035-1039 只有满足下列条件的报文段才会被处理： URG标志置位，紧急数据偏移量(ti_urp)非零，连接还未收到 FIN。只有当连接的状态等于 TIME_WAIT时，宏TCPS_HAVERCVDFIN才会为真，因此，连接处于任何其他状态时， URG都会被处理。在后面的注释中提到，连接处于CLOSE_WAIT、CLOSING、LAST_ACK和TIME_WAIT等几个状态时，URG标志会被忽略，这种说法是错误的。

2. 忽略超出的紧急指针

1040-1050 如果紧急数据偏移量加上接收缓存中已有的数据超过了插口缓存可容纳的数据量，则忽略紧急标志。紧急数据偏移量被清零， URG标志被清除，剩余的紧急方式处理逻辑被忽略。

图29-17给出了tcp_input下一部分的代码，处理紧急指针。

tcp_input.c

```
1051         /*
1052          * If this segment advances the known urgent pointer,
1053          * then mark the data stream.  This should not happen
1054          * in CLOSE_WAIT, CLOSING, LAST_ACK or TIME_WAIT states since
1055          * a FIN has been received from the remote side.
1056          * In these states we ignore the URG.
1057          *
1058          * According to RFC961 (Assigned Protocols),
1059          * the urgent pointer points to the last octet
1060          * of urgent data.  We continue, however,
1061          * to consider it to indicate the first octet
1062          * of data past the urgent section as the original
1063          * spec states (in one of two places).
1064          */
1065         if (SEQ_GT(ti->ti_seq + ti->ti_urp, tp->rcv_up)) {
```

图29-17 tcp_input 函数：处理收到的紧急指针

```
1066                tp->rcv_up = ti->ti_seq + ti->ti_urp;
1067                so->so_oobmark = so->so_rcv.sb_cc +
1068                    (tp->rcv_up - tp->rcv_nxt) - 1;
1069                if (so->so_oobmark == 0)
1070                    so->so_state |= SS_RCVATMARK;
1071                sohasoutofband(so);
1072                tp->t_oobflags &= ~(TCPOOB_HAVEDATA | TCPOOB_HADDATA);
1073            }
1074            /*
1075             * Remove out-of-band data so doesn't get presented to user.
1076             * This can happen independent of advancing the URG pointer,
1077             * but if two URG's are pending at once, some out-of-band
1078             * data may creep in... ick.
1079             */
1080            if (ti->ti_urp <= ti->ti_len
1081 #ifdef SO_OOBINLINE
1082                && (so->so_options & SO_OOBINLINE) == 0
1083 #endif
1084                )
1085                tcp_pulloutofband(so, ti, m);
1086        } else {
1087            /*
1088             * If no out-of-band data is expected, pull receive
1089             * urgent pointer along with the receive window.
1090             */
1091            if (SEQ_GT(tp->rcv_nxt, tp->rcv_up))
1092                tp->rcv_up = tp->rcv_nxt;
1093        }
```

tcp_input.c

图29-17 (续)

1051-1065 如果接收报文段的起始序号加上紧急数据偏移量超过了当前接收紧急指针，说明已收到了一个新的紧急指针。例如，图 26-30中的携带3字节的报文段到达接收方，如图 29-18所示。

一般情况下，收到的紧急指针(rcv_up)等于rcv_nxt。这个例子中，因为if语句为真(4加3大于4)，rcv_up的新值等于7。

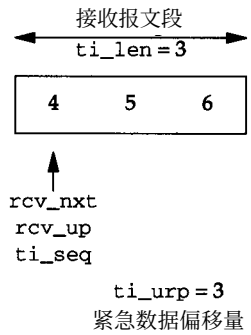

图29-18 图26-30中发送的报文段到达接收方

3. 计算收到的紧急指针

1066-1070 计算插口接收缓存中带外数据的分界点，应计入接收缓存中已有的数据(so_rcv.sb_cc)。在上面的例子中，假定接收缓存为空，so_oobmark等于2：序号为6的字节被认为是带外数据。如果这个带外数据标记等于0，说明插口正处在带外数据分界点上。如果发送带外数据的send系统调用给定长度为1，并且这个报文段到达对端时接收缓存为空，就会发生这一现象，同时也再次重申了 Berkeley系统认为紧急指针应指向带外数据后的第一字节。

4. 向应用进程通告TCP的紧急方式

1071-1072 调用sohasoutofband告知应用进程有带外数据到达了插口，清除两个标志

TCPOOB_HAVEDATA和TCPOOB_HADDATA，它们用于图30-8中的PRU_RCVOOB请求处理。

5. 从正常的数据流中提取带外数据

1074-1085 如果紧急数据偏移量小于等于接收报文段中的字节数，说明带外数据包含在报文段中。TCP的紧急方式允许紧急数据偏移量指向尚未收到的数据。如果定义了SO_OOBINLINE常量(正常情况下，Net/3定义了此常量)，而且未选用对应的插口选项，则接收进程将从正常的数据流中提取带外数据，并保存在t_iobc变量中。完成这一功能的函数，是我们将在下一节介绍的tcp_pulloutofband。

注意，无论紧急指针指向的字节是否可读，TCP都将通知接收进程发送方已进入紧急方式。这是TCP紧急方式的一个特性。

6. 如果不处于紧急方式，调整接收紧急指针

1086-1093 在接收方未处理紧急指针时，如果rcv_nxt大于接收紧急指针，则rcv_up向右移动，并等于rcv_nxt。这使接收紧急指针一直指向接收窗口的左侧，确保在收到URG标志时，图29-17起始处的宏SEQ_GT能够得出正确的结果。

如果要实现习题26.6中提出的方案，也必须相应修改图29-16和图29-17中的代码。

29.8 tcp_pulloutofband函数

图29-17中的代码调用了这个函数，如果：

1) 接收报文段中带有紧急方式标志；并且

2) 带外数据包含在接收报文段中(如，紧急指针指向接收报文段)；并且

3) 未选用SO_OOBINLINE选项。

函数从正常的数据流(保存接收报文段的mbuf链)中提取带外字节，并保存在连接TCP控制块中的t_iobc变量中。应用进程通过recv系统调用，置位MSG_OOB标志，读取这个变量：图30-8中的PRU_RCVOOB请求。图29-19给出了函数代码。

tcp_input.c

```
1282 void
1283 tcp_pulloutofband(so, ti, m)
1284 struct socket *so;
1285 struct tcpiphdr *ti;
1286 struct mbuf *m;
1287 {
1288     int     cnt = ti->ti_urp - 1;

1289     while (cnt >= 0) {
1290         if (m->m_len > cnt) {
1291             char    *cp = mtod(m, caddr_t) + cnt;
1292             struct tcpcb *tp = sototcpcb(so);

1293             tp->t_iobc = *cp;
1294             tp->t_oobflags |= TCPOOB_HAVEDATA;
1295             bcopy(cp + 1, cp, (unsigned) (m->m_len - cnt - 1));
1296             m->m_len--;
1297             return;
1298         }
1299         cnt -= m->m_len;
```

图29-19 tcp_pulloutofband 函数：将带外数据保存在t_iobc 变量中

```
1300          m = m->m_next;
1301          if (m == 0)
1302              break;
1303      }
1304      panic("tcp_pulloutofband");
1305  }
```

tcp_input.c

图29-19 (续)

1282-1289 考虑图29-20中的例子。紧急数据偏移量等于3，因此紧急指针等于7，带外字节的序号等于6。接收报文段携带了5字节的数据，全部保存在一个mbuf中。

变量cnt等于2，因为m_len(等于5)大于2，执行if语句为真部分的代码。

1290-1298 cp指向序号为6的字节，它被放入保存带外字节的变量 t_iobc中。置位TCPOOB_HAVEDATA标志，调用bcopy将接下来的两个字节(序号7和8)左移1字节，如图29-21所示。

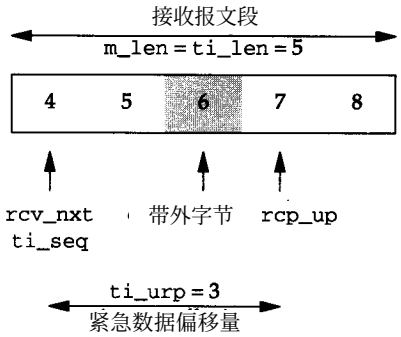

图29-20 携带带外字节的报文段

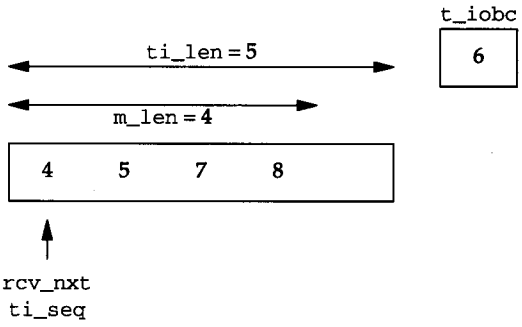

图29-21 移走带外数据后的结果(接图29-20)

注意，数字7和8指数据字节的序号，而不是其内容。 mbuf的长度从5减为4，但ti_len仍等于5不变，这是为了按序把报文段放入插口的接收缓存。 TCP_REASS宏和tcp_reass函数(在下一节调用)都会给rcv_nxt增加ti_len，本例中ti_len必须等于5，因为下一个等待接收的序号等于 9。还请注意，函数没有对第一个 mbuf中的数据分组首部长度(m_pkthdr.len)减1，这是因为负责把数据添加到插口接收缓存的 sbappend不使用此长度值。

跳至链中的下一个mbuf

1299-1302 如果带外数据未保存在此mbuf中，则从cnt中减去mbuf中的字节数，处理链中的下一个mbuf。因为只有当紧急数据移量指向接收报文段时，才会调用此函数，所以，如果链已结束，不存在下一个mbuf，则执行break语句，跳转到标注panic处。

29.9 处理已接收的数据

tcp_input接着提取收到的数据(如果存在)，将其添加到插口接收缓存，或者放入插口的乱序重组队列中。图29-22给出了完成此项功能的代码。

tcp_input.c

```
1094    dodata:                         /* XXX */
1095      /*
1096       * Process the segment text, merging it into the TCP sequencing queue,
1097       * and arranging for acknowledgment of receipt if necessary.
1098       * This process logically involves adjusting tp->rcv_wnd as data
1099       * is presented to the user (this happens in tcp_usrreq.c,
1100       * case PRU_RCVD).  If a FIN has already been received on this
1101       * connection then we just ignore the text.
1102       */
1103      if ((ti->ti_len || (tiflags & TH_FIN)) &&
1104          TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1105          TCP_REASS(tp, ti, m, so, tiflags);
1106          /*
1107           * Note the amount of data that peer has sent into
1108           * our window, in order to estimate the sender's
1109           * buffer size.
1110           */
1111          len = so->so_rcv.sb_hiwat - (tp->rcv_adv - tp->rcv_nxt);
1112      } else {
1113          m_freem(m);
1114          tiflags &= ~TH_FIN;
1115      }
```

tcp_input.c

图29-22 tcp_input 函数：把收到的数据放入插口接收队列

1094-1105 报文段数据将被处理，如果：

1) 接收数据的长度大于0，或者FIN标志置位；并且

2) 连接还未收到FIN；

则调用宏TCP_REASS处理数据。如果数据次序正确(如，连接等待接收的下一序号)，置位延迟ACK标志，增加rcv_nxt，并把数据添加到插口的接收缓存中。如果数据次序错误，宏会调用tcp_reass函数，把数据加入到连接的重组队列中(新到数据有可能填充队列中的缺口，从而将已排队的数据添加到插口的接收缓存中)。

前面介绍过，宏的最后一个参数(tiflags)是可修改的。特别地，如果数据次序错误，tcp_reass令tiflags等于0，清除FIN标志(如果它已置位)。这也就是为什么即使报文段中没有数据，只要FIN置位，if语句也为真。

考虑下面的例子。连接建立后，发送方立即发送报文段：一个携带字节 1~1024，另一个携带字节1025~2048，还有一个未带数据的FIN。第一个报文段丢失，因此，第二个报文段到达时(字节1025~2048)，接收方将其放入乱序重组队列，并立即发送 ACK。当第三个带有FIN标志的报文段到达时，图29-22中的代码被执行。即使数据长度等于0，因为FIN置位，导致调用TCP_REASS，它接着调用tcp_reass。因为ti_seq(2049，FIN的序号)不等于rcv_nxt(1)，tcp_reass返回0(图27-23)。在TCP_REASS宏中，tiflags被设为0，从而清除了FIN标志，阻止后续代码(图29-10)继续处理FIN。

猜测对端发送缓存大小

1106-1111 计算len，实际上是在猜测对端发送缓存的大小。考虑下面的例子。插口接收缓存大小等于8192(Net/3的默认值)，因此，TCP在SYN中通告窗口大小为8192。之后收到第一个报文段，携带字节1~1024。图29-23给出了在TCP_REASS增加rcv_nxt以反应收到的数据后接收空间的状态。

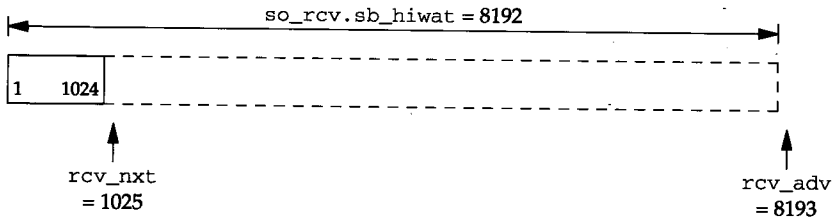

图29-23 大小为8192的接收窗口收到字节1~1024后的状态

此时，经计算，len等于1024。对端向接收窗口发送更多数据后，len值将增加，但绝不会超过对端发送缓存的大小。前面介绍过，图29-15中对变量max_sndwnd的计算，是在猜测对端接收缓存的大小。

事实上，变量len从未被使用。它是从Net/1遗留下来的，len计算后被存储到TCP控制块的max_rcvd变量中：

```
if (len > tp->max_rcvd)
   tp->max_rcvd = len;
```

但即使在Net/1中，变量max_rcvd也未被使用。

1112-1115 如果len等于0，且FIN标志未置位，或者连接上已收到了FIN，则丢弃保存接收报文段的mbuf链，并清除FIN。

29.10 FIN处理

tcp_input的下一步，在图29-24中给出，处理FIN标志。

tcp_input.c

```
1116     /*
1117      * If FIN is received ACK the FIN and let the user know
1118      * that the connection is closing.
1119      */
1120     if (tiflags & TH_FIN) {
1121         if (TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1122             socantrcvmore(so);
1123             tp->t_flags |= TF_ACKNOW;
1124             tp->rcv_nxt++;
1125         }
1126         switch (tp->t_state) {

1127             /*
1128              * In SYN_RECEIVED and ESTABLISHED states
1129              * enter the CLOSE_WAIT state.
1130              */
1131         case TCPS_SYN_RECEIVED:
1132         case TCPS_ESTABLISHED:
1133             tp->t_state = TCPS_CLOSE_WAIT;
1134             break;
```

tcp_input.c

图29-24 tcp_input 函数：FIN处理，前半部分

1. 处理收到的第一个FIN

1116-1125 如果接收报文段FIN置位，并且是连接上收到的第一个FIN，则调用socantrcvmore，把插口设为只读，置位TF_ACKNOW，从而立即发送ACK(无延迟)。

rcv_nxt加1，越过FIN占用的序号。

1126 FIN处理的其余部分是一个大的switch语句，根据连接的状态进行转换。注意，连接处于CLOSED、LISTEN和SYN_SENT状态时，不处理FIN，因为处于这3个状态时，还未收到对端发送的SYN，无法同步接收序号，也就无法验证FIN序号的有效性。此外，连接处于CLOSING、CLOSE_WAIT和LAST_ACK状态时，也不处理FIN，因为在这3个状态下收到的FIN必然是一个重复报文段。

2. SYN_RCVD和ESTABLISHED状态

1127-1134 如果连接处于SYN_RCVD或ESTABLISHED状态，收到FIN后，新的状态为CLOSE_WAIT。

尽管在SYN_RCVD状态下收到FIN是合法的，但却极为罕见。图24-15的状态图未列出这一状态变迁。它意味着处于LISTEN状态的插口收到一个同时带有SYN和FIN的报文段。或者，正在监听的插口收到了SYN，连接转移到SYN_RCVD状态，但在收到ACK之前，先收到了FIN(从分析可知，FIN未携带有效的ACK，否则，图29-2中的代码会使连接转移到ESTABLISHED状态)。

图29-25给出了FIN处理的下一部分。

tcp_input.c

```
1135             /*
1136              * If still in FIN_WAIT_1 state FIN has not been acked so
1137              * enter the CLOSING state.
1138              */
1139         case TCPS_FIN_WAIT_1:
1140             tp->t_state = TCPS_CLOSING;
1141             break;

1142             /*
1143              * In FIN_WAIT_2 state enter the TIME_WAIT state,
1144              * starting the time-wait timer, turning off the other
1145              * standard timers.
1146              */
1147         case TCPS_FIN_WAIT_2:
1148             tp->t_state = TCPS_TIME_WAIT;
1149             tcp_canceltimers(tp);
1150             tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1151             soisdisconnected(so);
1152             break;

1153             /*
1154              * In TIME_WAIT state restart the 2 MSL time_wait timer.
1155              */
1156         case TCPS_TIME_WAIT:
1157             tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1158             break;
1159         }
1160     }
```

tcp_input.c

图29-25 tcp_input 函数：FIN处理，后半部分

3. FIN_WAIT_1状态

1135-1141 因为报文段的ACK处理已结束，如果处理FIN时，连接处于FIN_WAIT_1状态，意味着连接两端同时关闭连接——两端发送的两个FIN在网络中交错。连接进入CLOSING状态。

4. FIN_WAIT_2状态

1142-1148 收到FIN将使连接进入TIME_WAIT状态。当在FIN_WAIT_1状态收到携带ACK和FIN的报文段时(典型情况)，尽管图24-15显示连接直接从FIN_WAIT_1转移到TIME_WAIT状态，但在图29-11中处理ACK时，连接实际已进入FIN_WAIT_2状态。此处的FIN处理再将连接转到TIME_WAIT状态。因为ACK在FIN之前处理，所以连接总会经过FIN_WAIT_2状态，尽管是暂时性的。

5. 启动TIME_WAIT定时器

1149-1152 关闭所有等待的TCP定时器，并启动TIME_WAIT定时器，时限等于MSL(如果接收报文段中包含ACK和FIN，图29-11中的代码会启动FIN_WAIT_2定时器)。插口断开连接。

6. TIME_WAIT状态

1153-1159 如果在TIME_WAIT状态时收到FIN，说明这是一个重复报文段。与图29-14中的处理类似，启动TIME_WAIT定时器，时限等于两倍的MSL。

29.11 最后的处理

图29-26给出了tcp_input函数中首部预测失败时，较慢的执行路径中最后一部分的代码，以及标注dropafterack。

tcp_input.c

```
1161     if (so->so_options & SO_DEBUG)
1162         tcp_trace(TA_INPUT, ostate, tp, &tcp_saveti, 0);

1163     /*
1164      * Return any desired output.
1165      */
1166     if (needoutput || (tp->t_flags & TF_ACKNOW))
1167         (void) tcp_output(tp);
1168     return;

1169   dropafterack:
1170     /*
1171      * Generate an ACK dropping incoming segment if it occupies
1172      * sequence space, where the ACK reflects our state.
1173      */
1174     if (tiflags & TH_RST)
1175         goto drop;
1176     m_freem(m);
1177     tp->t_flags |= TF_ACKNOW;
1178     (void) tcp_output(tp);
1179     return;
```

tcp_input.c

图29-26 tcp_input 函数：最后的处理

1. SO_DEBUG插口选项

1161-1162 如果选用了SO_DEBUG插口选项，则调用tcp_trace向内核的环形缓存中添加记录。回想一下，图28-7中的代码同时保存了原有连接状态， IP和TCP的首部，因为函数有可能改变这些值。

2. 调用tcp_output

1163-1168 如果needoutput标志置位(图29-6和图29-15)，或者需要立即发送ACK，则调用tcp_output。

3. dropafterack

1169-1179 只有当RST标志未置位时，才会生成ACK(带有RST的报文段不会被确认)，释放保存接收报文段的mbuf链，调用tcp_output立即发送ACK。

图29-27结束tcp_input函数。

tcp_input.c

```
1180   dropwithreset:
1181     /*
1182      * Generate an RST, dropping incoming segment.
1183      * Make ACK acceptable to originator of segment.
1184      * Don't bother to respond if destination was broadcast/multicast.
1185      */
1186     if ((tiflags & TH_RST) || m->m_flags & (M_BCAST | M_MCAST) ||
1187         IN_MULTICAST(ti->ti_dst.s_addr))
1188         goto drop;
1189     if (tiflags & TH_ACK)
1190         tcp_respond(tp, ti, m, (tcp_seq) 0, ti->ti_ack, TH_RST);
1191     else {
1192         if (tiflags & TH_SYN)
1193             ti->ti_len++;
1194         tcp_respond(tp, ti, m, ti->ti_seq + ti->ti_len, (tcp_seq) 0,
1195                     TH_RST | TH_ACK);
1196     }
1197     /* destroy temporarily created socket */
1198     if (dropsocket)
1199         (void) soabort(so);
1200     return;
1201   drop:
1202     /*
1203      * Drop space held by incoming segment and return.
1204      */
1205     if (tp && (tp->t_inpcb->inp_socket->so_options & SO_DEBUG))
1206         tcp_trace(TA_DROP, ostate, tp, &tcp_saveti, 0);
1207     m_freem(m);
1208     /* destroy temporarily created socket */
1209     if (dropsocket)
1210         (void) soabort(so);
1211     return;
1212 }
```

tcp_input.c

图29-27 tcp_input 函数：最后的处理

4. dropwithreset

1180-1188 除了接收报文段也有RST,或者接收报文段是多播和广播报文段的情况之外,应发送RST。绝不允许因为响应RST而发送新的RST,这将引起RST风暴(两个端点间连续不断地交换RST)。

此处的代码存在与图28-16同样的错误：它不检查接收报文段的目的地址是否为广播地址。

类似地，IN_MULTICAST的目的地址参数应转换为主机字节序。

5. RST报文段的序号和确认序号

1189-1196 RST报文段的序号字段值、确认字段值和ACK标志取决于接收报文段中是否带有ACK。

图29-28总结了生成RST报文段中的这些字段。

接收到的报文段	生成的RST报文段		
	序 号 值	确认序号	输出标志
带有ACK	接收到的确认字段	0	TH_RST
不带ACK	0	接收到的序号字段	TH_RST|TH_ACK

图29-28 生成RST报文段各字段的值

正常情况下，除了起始的SYN(图24-16)，所有报文段都带有ACK。tcp_respond的第四个参数是确认序号，第五个参数是序号。

6. 拒绝连接

1192-1193 如果SYN置位，则ti_len必须加1，从而生成RST的确认字段比收到的SYN报文段的起始序号大1。如果到达的SYN请求与不存在的服务器建立连接，会执行这一段代码。此时，由于图28-6中的代码找不到请求的Internet PCB，控制跳转到dropwithreset。但为了使发送的RST能被对端接受，报文段必须确认SYN(图28-18)。卷1的18.14节举例说明了这种类型的RST。

最后请注意，tcp_respond利用保存接收报文段的第一个mbuf构造RST，并且释放链上的其他mbuf。当第一个mbuf最终到达设备驱动程序后，它也会被丢弃。

7. 释放临时创建的插口

1197-1199 如果在图28-7中为监听的服务器创建了临时的插口，但图28-16中的代码发现接收报文段有错误，它会置位drop socket。如果出现了这种情况，插口在此处被释放。

8. 丢弃(不带ACK或RST)

1201-1206 如果接收报文段被丢弃，且不生成ACK或RST，则调用tcp_trace。如果SO_DEBUG置位且生成了ACK，则tcp_output将向内核的环形缓存中添加一条跟踪记录。如果SO_DEBUG置位且生成了RST，系统不会为RST添加新的跟踪记录。

1207-1211 释放保存接收报文段的mbuf链。如果dropsocket非零，则释放临时创建的插口。

29.12 实现求精

为了加速TCP处理而进行的优化与UDP类似(23.12节)。应利用复制数据计算检验和，并避免在处理中多次遍历数据。[Dalton et al. 1993]讨论了这些修订。

连接数增加时，对TCP PCB的线性搜索也是一个处理瓶颈。[McKenney and Dove 1992]讨论了这个问题，利用哈希表替代了线性搜索。

[Partridge 1993]介绍了Van Jacobson开发的一个用于研究目的的协议实现，极大地减少了TCP的输入处理。接收数据分组首先由IP进行处理(RISC系统中约有25条指令)，之后由分用器(demultiplexer)寻找PCB(约10条指令)，最后由TCP处理(约30条指令)。这30条指令完成了首部预测，并计算伪首部检验和。如果数据报文段通过了首部预测，且应用进程正等待接收数据，则复制数据到应用进程缓存，计算TCP检验和并完成验证(一次遍历中完成数据复制和检验和计算)。如果TCP首部预测失败，则执行TCP输入处理中较慢的路径。

29.13 首部压缩

下面介绍TCP首部压缩。尽管首部压缩不是TCP输入处理的一部分，但需要彻底了解TCP

的工作机制后，才能很好地理解首部压缩。RFC 1144[Jacobson 1994a]中详细定义了首部压缩，因为Van Jacobson首先提出了这一算法，通常也称为 VJ 首部压缩。本节的目的不是详细讨论首部压缩的源代码(RFC 1144给出了实现代码，其中有很好的注释，程序量与 tcp_output差不多)，而是概括性地介绍一下算法的思想。请注意区分首部预测 (28.4节)和首部压缩。

29.13.1 引言

多数的SLIP和PPP实现支持首部压缩。尽管首部压缩，在理论上，适用于任何数据链路，但主要还是面向慢速串行链路。首部压缩只处理TCP报文段 —— 与其他的IP协议无关(如ICMP、IGMP、UDP等等)。它能够把IP/TCP组合首部从正常的40字节压缩到只有3字节，从而降低了交互性应用，如远程登录或 Telnet中TCP报文段的大小，从典型的 41字节减少到只剩 4字节 —— 大大提高了慢速串行链路的效率。

串行链路的两端，每端都维护着两个连接状态表，一个用于数据报的发送，另一个用于数据报的接收。每张表最多保存 256条记录，但典型的只有 16条，即同一时间内最多允许 16条不同的TCP连接执行首部压缩算法。每条记录中保存一个 8 bit的连接ID(限制记录数最多只能为256)、某些标志和最近接收/发送的数据报的未被压缩的首部。 96 bit的插口对可惟一确定一条连接 —— 源端IP地址和TCP端口、目的IP地址和TCP端口 —— 这些信息都保存在未压缩的首部中。图29-29举例说明了这些表的结构。

因为TCP连接是全双工的，在两个方向的数据流上都可执行首部压缩算法。连接两端必须同时实现压缩和解压缩。同一条连接在两端的表中都会出现，如图 29-29所示。在这个例子上部的两张表中，连接 ID等于1的表项的源端IP地址都等于128.1.2.3，源端TCP端口号都等于1500，目的IP地址等于192.3.4.5，目的TCP端口号都等于 25。在底部的两张表中，连接 ID等于2的记录保存了同一条连接反方向数据流的信息。

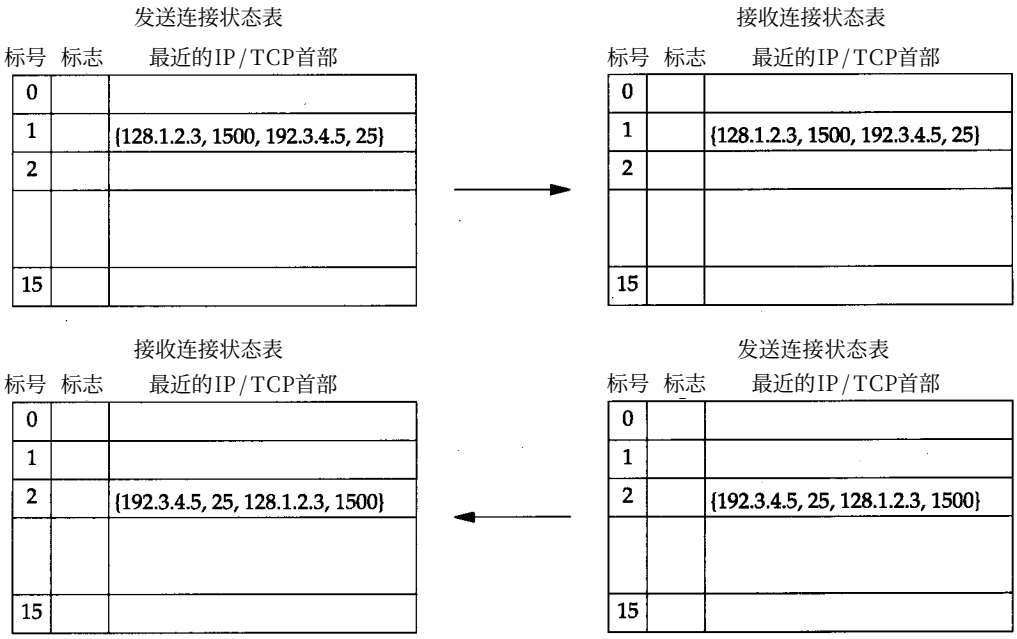

图29-29 链路(如SLIP链路)两端的一组连接状态表

我们在图29-29中利用数组表示这些表，但在源代码中，表项定义为一个结构，连接状态表定义为这些结构组成的环形链表，最近一次用过的结构位于表头。

因为连接两端都保存了最近用过的未压缩的数据报首部，所以只需在链路上传送当前数据报与前一数据报不同的字段（及一个特殊的前导字节，指明后续的是哪一个字段）。因为某些首部字段在相邻的数据报之间不会变化，而其他的首部字段变化也很小，这种差分处理是压缩算法的核心。首部压缩只适用于IP和TCP首部——TCP报文段的数据部分不变。

图29-30给出了发送方利用首部压缩算法，在串行链路上发送IP数据报时采取的步骤。

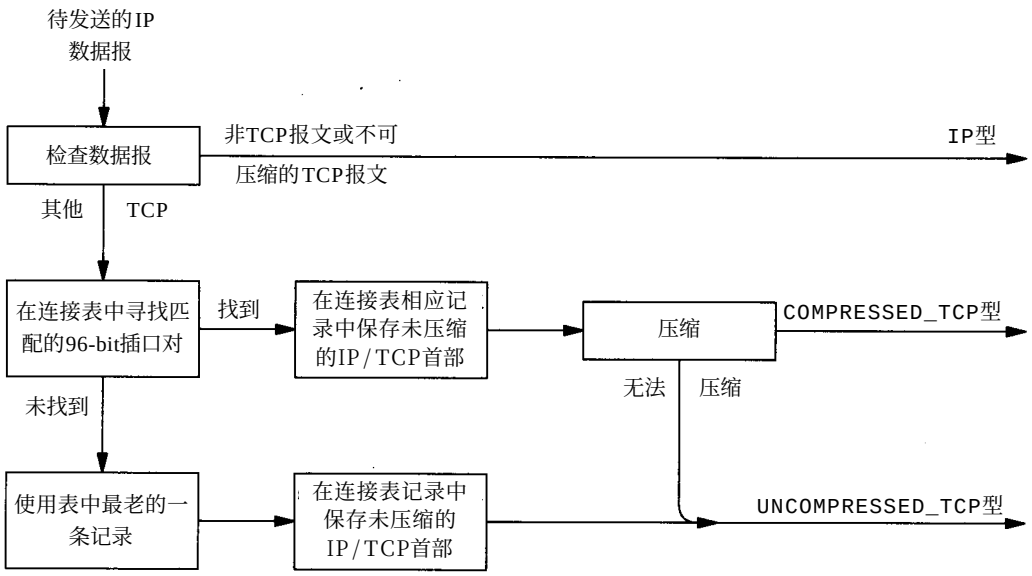

图29-30 发送方采用首部压缩时的步骤

接收方必须能够识别下面3种类型的数据报：

1) IP型数据报，前导字节的高位4比特等于4。这也是IP首部中正常的IP版本号(图8-8)，说明链路上发送的是正常的、未压缩的数据报。

2) COMPRESSED_TCP型数据报，前导字节的最高位置为1，类似于IP版本号介于8和15之间(剩余的7bit由压缩算法使用)，说明链路上发送的是压缩过的首部和未压缩的数据，接下来我们还会谈到这种类型的数据报。

3) UNCOMPRESSED_TCP型数据报，前导字节的高位4比特等于7，说明链路上发送的是正常的、未压缩的数据报，但IP的协议字段(等于6，对TCP)被替换为连接ID，接收方可据此从连接状态表中找到正确的记录。

接收方查看数据报的第一个字节，即前导字节，确定其类型，实现代码参见图5-13。图5-16中，发送方调用sl_compress_tcp确认TCP报文段是可压缩的，函数返回值与数据报首字节逻辑或后，结果依然保存在首字节中。

图29-31列出了链路上传送的前导字节，其中4位“-”表示正常的IP首部长度字段。7位“C、I、P、S、A、W和E”指明后续的是哪些可选字段，后面会简单地介绍这些字母的含义。

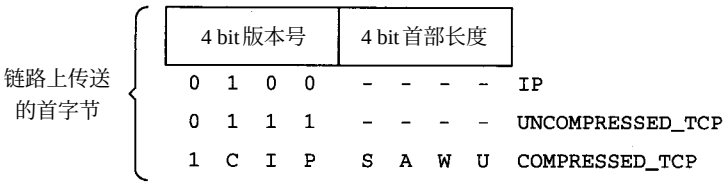

图29-31 链路上传送的前导字节

图29-32给出了使用压缩算法之后，不同类型的完整的IP数据报。

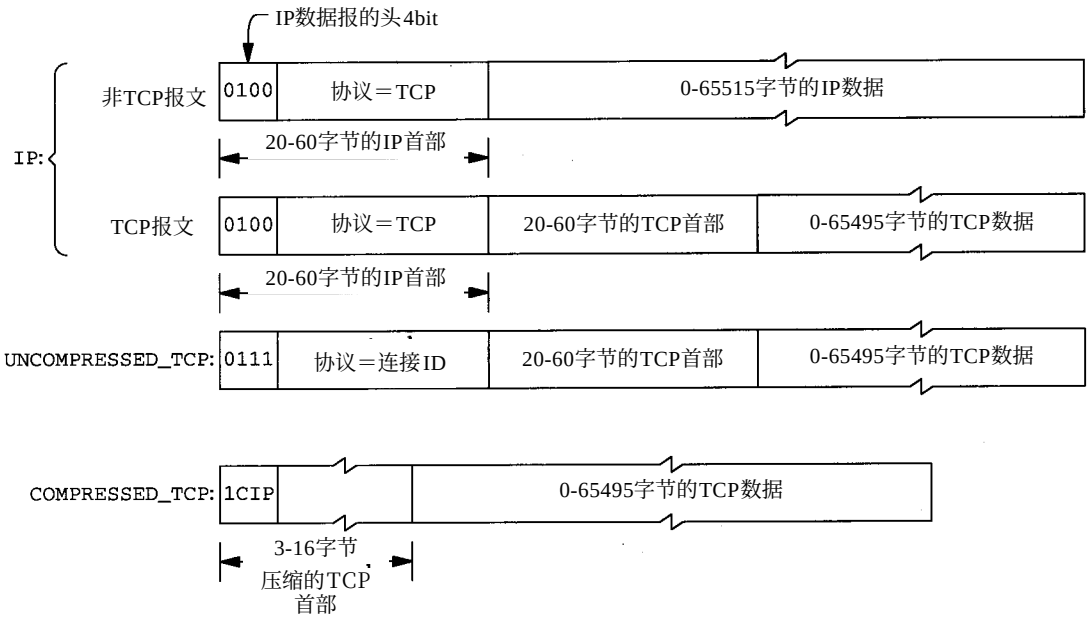

图29-32 采用首部压缩后的不同类型的IP数据报

图中给出了两个IP型数据报：一个携带了非TCP报文段(如UDP、ICMP或IGMP协议报文段)，另一个携带了TCP报文段。这是为了说明做为IP型数据报发送的TCP报文段与做为UNCOMPRESSED_TCP型数据报发送的TCP报文段间的差异：前导字节的高位4比特互不相同，类似于IP首部的协议字段。

如果IP数据报的协议字段不等于TCP，或者协议是TCP，但下列条件之一为真，都不会采用首部压缩算法。

- 数据报是一个IP分片：分片偏移量非零或者分片标志置位；
- SYN、FIN或RST中的任何一个置位；
- ACK标志未置位。

上述3个条件中只要有一个为真，都将作为IP型数据报发送。

此外，即使数据报携带了可压缩的TCP报文段，压缩算法也可能失败，生成UNCOMPRESSED_TCP型的数据报。可能因为当前数据报与连接上发送的上一个数据报比较时，有些特殊字段发生了变化，而正常情况下，对于给定的连接，它们应该不变，从而导致压缩算法无法反映存在的变化。例如，TOS字段，分片标志位。此外，如果某些字段数值的

差异超过65535，压缩算法也会失败。

29.13.2 首部字段的压缩

下面介绍如何压缩图29-33中给出的IP和TCP的首部字段，阴影字段指对于给定连接，正常情况下不会发生变化的字段。

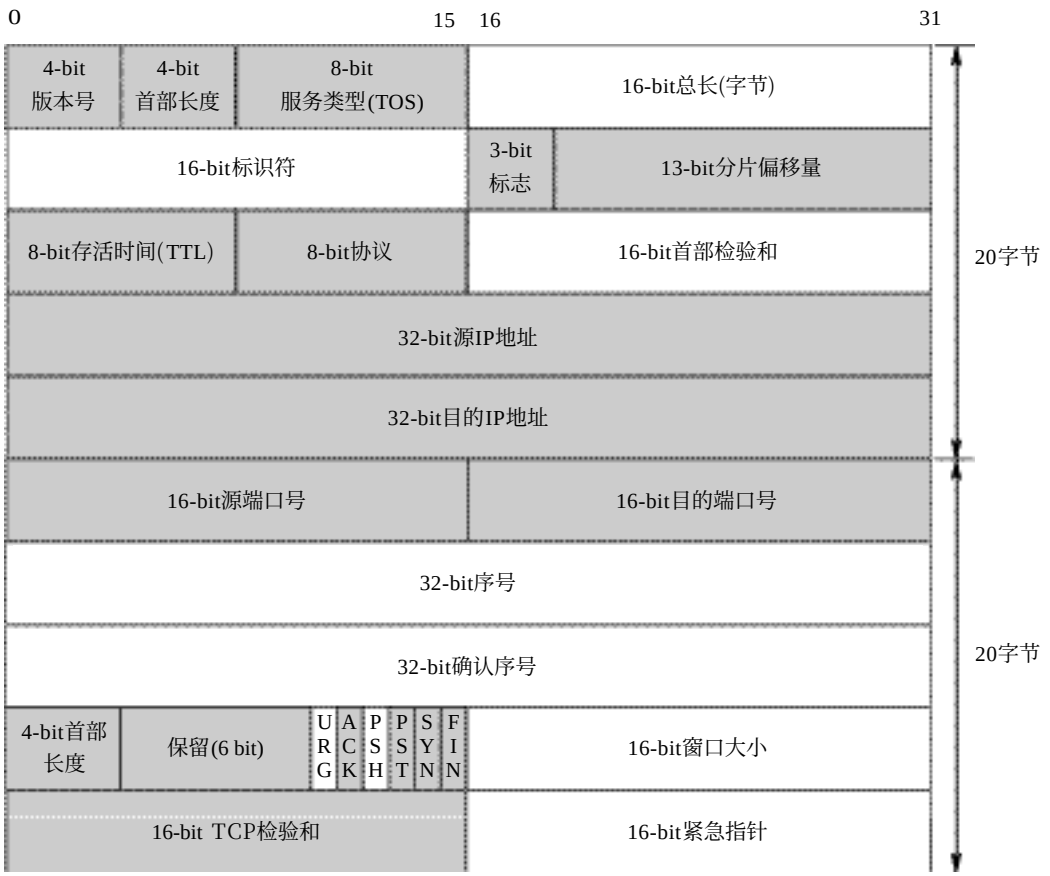

图29-33 组合的IP和TCP首部：阴影字段通常不变化

如果连接上发送的前一个报文段与当前报文段之间，有阴影字段发生变化，则压缩算法失败，报文段被直接发送。图中未列出 IP和TCP选项，但如果它们存在，且这些选项字段发生了变化，则报文段也不压缩，而被直接发送 (习题29.7)。

如果阴影字段均未变化，即使算法只传输非阴影字段，也会节省 50%的传输容量。 VJ首部压缩甚至做得更好，图 29-34给出了压缩后的IP/TCP首部格式。

最小的压缩后的IP/TCP首部只有3个字节：第一个字节(标志比特)，加上16 bit的TCP检验和。为了防止可能的链路错误，一般不改动 TCP检验和(SLIP不提供链路层的检验和，尽管PPP提供一个)。

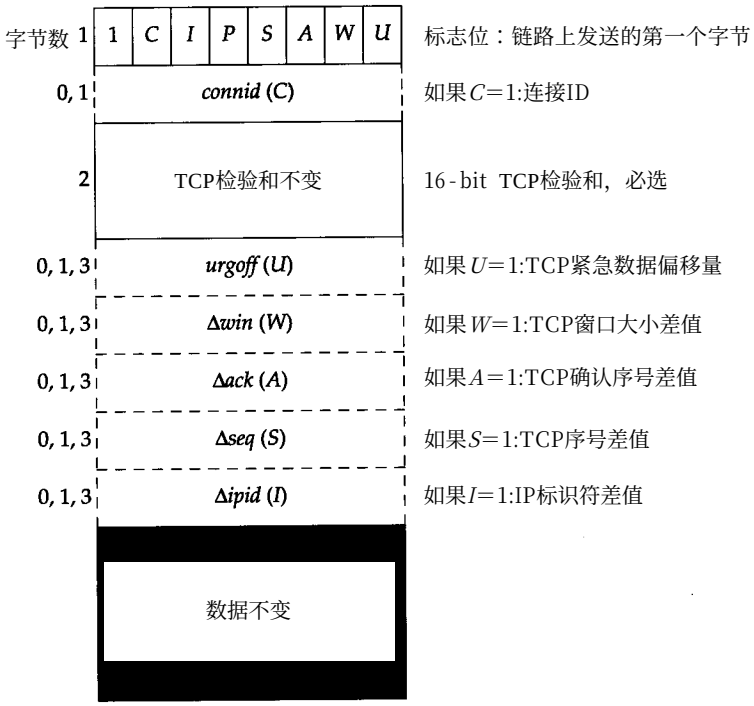

图29-34 压缩后的IP/TCP首部格式

其他的6个字段：*connid*、*urgoff*、Δ*win*、Δ*ack*、Δ*seq*和Δ*ipid*，都是可选的。图29-34的最左侧列出了各字段压缩后所需的字节数。读者可能认为压缩后的首部最大应占用 19字节，但实际上压缩后的首部中4 bit的*SAWU*绝不可能同时置位，因此，压缩首部最大为16字节，后面我们还会详细讨论这个问题。

第一个字节的最高位比特必须设为 1，说明这是`COMPRESSED_TCP`型的数据报。其余7 bit中的6个规定了后续首部中存在哪些可选字段，图29-35小结了这7 bit的用法。

标志比特	描　述	结构变量	标志等于0说明	标志等于1说明
C	连接ID		连接ID不变	*connid*=连接ID
I	IP标识符	`ip_id`	`ip_id`已加1	Δ*ipid*=IP标识符差值
P	TCP推标志		`PSH`标志清除	`PSH`标志置位
S	TCP序号	`th_seq`	`th_seq`不变	Δ*seq*=TCP序号差值
A	TCP确认序号	`th_ack`	`th_ack`不变	Δ*ack*=TCP确认序号差值
W	TCP窗口	`th_win`	`th_win`不变	Δ*win*=TCP窗口字段差值
U	TCP紧急数据偏移量	`th_urg`	URG标志未置位	*urgoff*=紧急数据偏移量

图29-35 压缩首部中的7个标志比特

C 如果*C*比特等于0，则当前报文段与前一报文段 (无论是压缩的或非压缩的)具有相同的连接ID。如果等于1，则*connid*将等于连接ID，其值位于0~255之间。

I 如果*I*比特等于0，当前报文段的 IP标识符较前一报文段加 1(典型情况)。如果等于1，Δ*ipid*等于`ip_id`的当前值减去它的前一个值。

P 这个比特复制自TCP报文段中的PSH标志位。因为PSH标志不同于其他的正常方式，必须在每个报文段中明确地定义这一标志。

S 如果*S*比特等于0，TCP序号不变。如果等于1，Δ*seq*等于th_seq的当前值减去它的前一个值。

A 如果*A*比特等于0，TCP确认序号不变(典型情况)。如果等于1，Δ*ack*等于th_ack的当前值减去它的前一个值。

W 如果*W*比特等于0，TCP窗口大小不变。如果等于1，Δ*win*等于th_win的当前值减去它的前一个值。

U 如果*U*比特等于0，报文段的URG标志未置位，紧急数据偏移量不变(典型情况)。如果等于1，说明URG标志置位，*urgoff*等于th_urg的当前值。如果URG标志未置位时，紧急数据偏移量发生改变，报文段将被直接发送(这种现象通常发生在紧急数据传送完毕后的第一个报文段)。

通过字段的当前值减去它的前一个值，得到需传输的差值。正常情况下，得到的是一个小正数(Δ*win*是个例外)。

请注意，图29-34中有5个字段的长度可变，可占用0、1或3字节。

0字节：对应标志未置位，此字段不存在；

1字节：发送值在1~255之间，只需占用1字节；

3字节：如果发送值等于0或者在256~65535之间，则需要用3个字节才能表示：第一个字节全0，后两个字节保存实际值。这种方法一般用于3个16 bit的值：*urgoff*、Δ*win*和Δ*ipid*。但如果两个32 bit字段Δ*ack*和Δ*seq*的差值小于0或者大于65535，报文段将被直接发送。

如果把图29-33中不带阴影的字段与图29-34中可能的传输字段进行比较，会发现有些字段永远不会被传输。

- IP总长度字段不会被传输，因为绝大多数链路层向接收方提供接收数据分组的长度。
- 因为IP首部中被传输的惟一字段是16 bit的IP标识符，IP检验和被忽略。因为它只在一段链路上保护IP首部，每次转发都会被重新计算。

29.13.3 特殊情况

算法检查输入报文段，如果出现两种特定情况，则用前导字节的低位4比特——*SAWU*——的两种特殊组合，分别加以表示。因为紧急数据很少出现，如果报文段中URG标志置位，并且与前一报文段相比，序号与窗口字段都发生了变化(意味着低位4比特应为1011或1111)，此种报文段会跳过压缩算法，被直接发送。因此，如果低位4比特等于1011(称为**SA*)或1111(称为**S*)，就说明出现了下面两种特定情况：

*SA 序号与确认序号都增加，差值等于前一报文段的数据量，窗口大小与紧急数据偏移量不变，URG标志未置位。采用这种表示法可以避免传送Δ*seq*和Δ*ack*。
如果对端回送终端数据，那么两个传输方向上的数据报文段中都会经常出现这一现象。卷1的图19-3和图19-4，举例说明了远程登录应用中出现的这种类型的数据。

*S 序号增加，差值等于前一报文段的数据量，确认序号、窗口大小与紧急数据偏移量均不变，URG标志未置位。采用这种表示法可以避免传送Δ*seq*。
这种类型的数据通常出现在单向数据传输(如FTP)的发送方。卷1的图20-1、图20-2

和图20-3举例说明了这种类型的数据传输。此外，如果对端不回送终端数据，那么在数据发送方的数据报文段中也会出现这种现象。

29.13.4 实例

下面的两个例子，在图1-17中的bsdi和slip两个系统间，利用SLIP链路传输数据。这条SLIP链路在两个传输方向上都采用了首部压缩算法。在主机bsdi上运行tcpdump程序(卷1的附录A)，保存所有数据帧的备份。这个程序还支持一个选项，能够输出压缩后的首部，列出图29-34中的所有字段。

在主机间已建立了两条连接：一条远程登录连接，另一条是从bsdi到slip的文件传输(FTP)。图29-36列出了两条连接上不同类型数据帧出现的次数。

帧 类 型	远程登录		FTP	
	输入	输出	输入	输出
IP	1	1	5	5
UNCOMPRESSED_TCP	3	2	2	3
COMPRESSED_TCP				
特殊情况*SA	75	75	0	0
特殊情况*S	25	1	1	325
一般情况	9	93	337	13
总数	113	172	345	346

图29-36 远程登录和FTP连接上，不同类型数据帧出现的次数

远程登录连接中，在两个传输方向上，*SA都出现了75次，从而证明了在对端回显终端流量时，这一特定情况在两个传输方向上都会经常出现。FTP连接中，在数据的发送方，*S出现了325次，也证明了对于单向数据传输，这一特定情况会经常出现在数据的发送方。

FTP连接中，IP型的数据帧出现了10次，对应于4个带有SYN的报文段，和6个带有FIN的报文段。FTP使用了两条连接：一条用于传输交互式命令，另一条用于文件传输。

UNCOMPRESSED_TCP型数据帧一般对应于连接建立后的第一个报文段，即同步连接ID的报文段。这两个例子中还有少量的其他类型的报文段，主要用于服务类型设定(Net/3中的远程登录及FTP客户及服务器都是在连接建立后才设定TOS字段)。

字节数	远程登录		FTP	
	输入	输出	输入	输出
3	102	44	2	250
4		94		78
5	7	12	5	2
6		6	325	5
7		13	2	1
8				1
9			4	1
总数	109	169	338	338

图29-37 压缩首部大小的分布

图29-37给出了压缩首部大小的分布情况，后 4栏中压缩首部的平均大小为分别等于 3.1、4.1、6.0和3.3字节，与原来的40字节相比，大大提高了系统的传输效率，尤其对于交互式连接，效果更加明显。

在FTP输入一栏中，压缩首部大小为6字节的报文段有325个，其中绝大多数只携带了值等于256的Δ*ack*字段，因为256大于255，所以必须用3个字节表示。 SLIPMTU等于296，因此，TCP采用了256的MSS。在FTP输出一栏中，压缩首部大小为 3字节的报文段有250个，其中绝大多数都代表*S类的特定情况(只有序号发生变化)，差值等于256。但因为*S的序号差值默认为前一报文段的数据量，所以只需传输前导字节和 TCP检验和。在FTP输出一栏中，78个压缩首部大小为4字节的报文段也属于同一情况，只不过 IP标识符也发生了变化(习题29.8)。

29.13.5 配置

对给定的SLIP或PPP链路，首部压缩必须被选定后才能起作用。配置 SLIP链路接口时，一般可设定两个标志：首部压缩标志和自动首部压缩标志。配置命令是 `ifconfig`，分别带选项`link0`和`link2`。正常情况下，由客户端 (拨号主机)决定是否采用首部压缩算法，服务器(客户通过拨号接入的主机或终端服务器)只选择是否置位自动首部压缩标志。如果客户选用了首部压缩算法，它的 TCP首先发送一个`UNCOMPRESSED_TCP`型的数据报，规定连接 ID。如果服务器收到这个数据报，它也开始采用首部压缩算法 (服务器处于自动方式)；如果未收到这个数据报，服务器绝不会在这条链路上采用首部压缩。

PPP允许在链路建立时，连接双方共同协商传输选项，其中的一个选项即是否支持首部压缩算法。

29.14 小结

本章结束了我们对 TCP输入处理的详细介绍。首先介绍了如果连接在 SYN_RCVD状态时收到了ACK，该如何处理，即如何完成被动打开、同时打开或自连接。

快速重传算法指 TCP在连续收到的重复 ACK数超过规定的门限值后，能够检测到丢失的报文段并进行重发，即使重传定时器还未超时。 Net/3结合了快速重传算法与快速恢复算法，执行拥塞避免算法而非慢起动，尽量保证发送方到接收方的数据流不中断。

ACK处理负责从插口的发送缓存中丢弃已确认的数据，并且在收到的 ACK会改变连接当前状态时，对一些TCP状态做特殊处理。

处理接收报文段的 URG标志，如果置位，则通过 TCP紧急方式的处理，提取带外数据。这一操作是非常复杂的，因为应用进程可以利用正常的数据流缓存，或者特殊的带外数据缓存接收带外数据，而且TCP收到URG时，紧急指针所指向的数据可能还未到达。

TCP输入处理结束时，会调用 `TCP_REASS`，提取报文段中的数据放入插口的接收缓存或重组队列，处理 FIN标志，并且在接收报文段需要响应时，调用 `tcp_output`输入响应报文段。

TCP首部压缩是用于SLIP和PPP链路的一种技术，能够把 IP和TCP首部长度从40字节减少到约为3~6字节(典型情况)。这是因为对于给定连接，相邻两个报文段之间，首部的多数字段不会改变，即使有些字段的值发生了变化，其差值也很小，从而可以通过前导字节中的标志比特，指明哪些字段发生了变化，在后续部分只传输这些字段的当前值与前一报文段间的差值。

习题

29.1 客户与服务器建立连接，不考虑报文段丢失，哪一个应用进程，客户或服务器，首先完成连接建立过程？

29.2 Net/3系统中，监听服务器收到了一个 SYN，它同时携带了50字节的数据。会发生什么？

29.3 继续前一个习题，假定客户没有重传 50字节的数据，而是在对服务器 SYN/ACK报文段的确认中置位FIN标志，会发生什么？

29.4 Net/3客户向服务器发送SYN，服务器响应SYN/ACK，其中还携带了50字节的数据和FIN标志。列出客户端TCP的处理步骤。

29.5 卷1的图18-19和RFC 793的图14，都给出了出现同时关闭时，连接双方交换的4个报文段。但如果连接两端都是 Net/3系统，出现同时关闭时，或者一个 Net/3系统的自连接关闭时，彼此将交换6个报文段，而不是4个，多余出两个报文段是因为连接两端各自收到对端的FIN后，将向对端重发FIN。问题出在什么地方，如何解决？

29.6 RFC 793第72页建议，如果发送缓存中的数据已被对端确认，“应给用户一个确认，指明缓存中已发送且被确认的数据 (例如，发送缓存返回时应带有‘ OK ’响应)”。Net/3是否提供了这种机制？

29.7 RFC 1323中定义的选项对TCP首部压缩有何影响？

29.8 Net/3对IP标识符字段的赋值方式，对TCP首部压缩有何影响？