第24章　TCP：传输控制协议

24.1　引言

传输控制协议，即TCP，是一种面向连接的传输协议，为两端的应用程序提供可靠的端到端的数据流传输服务。它完全不同于无连接的、提供不可靠的数据报传输服务的 UDP协议。

我们在第23章中详细讨论了UDP的实现，有9个函数、约800行C代码。我们将要讨论的TCP实现包括28个函数、约4500行C代码，因此，我们将TCP的实现分成7章来讨论。

这几章中不包括对TCP概念的介绍，假定读者已阅读过卷 1的第17章~第24章，熟悉TCP的操作。

24.2　代码介绍

TCP实现代码包括7个头文件，其中定义了大量的TCP结构和常量和6个C文件，包含TCP函数的具体实现代码。文件如图24-1所示。

文　件	描　述
netinet/tcp.h	tcphdr结构定义
netinet/tcp_debug.h	tcp_debug结构定义
netinet/tcp_fsm.h	TCP有限状态机定义
netinet/tcp_seg.h	实现TCP序号比较的宏定义
netinet/tcp_timer.h	TCP定时器定义
netinet/tcp_var.h	tcpcb(控制块)和tcpstat(统计)结构定义
netinet/tcpip.h	TCP+IP首部定义
netinet/tcp_debug.c	支持SO_DEBUG协议端口号调试(第27.10节)
netinet/tcp_input.c	tcp_input及其辅助函数(第28和第29章)
netinet/tcp_output.c	tcp_output及其辅助函数(第26章)
netinet/tcp_subr.c	各种TCP子函数(第27章)
netinet/tcp_timer.c	TCP定时器处理(第25章)
netinet/tcp_usrreq.c	PRU_*xxx*请求处理(第30章)

图24-1　TCP各章中将讨论的文件

图24-2描述了各TCP函数与其他内核函数之间的关系。带阴影的椭圆分别表示我们将要讨论的9个主要的TCP函数，其中8个出现在protosw结构中(图24-8)，第9个是tcp_output。

24.2.1　全局变量

图24-3列出了TCP函数中用到的全局变量。

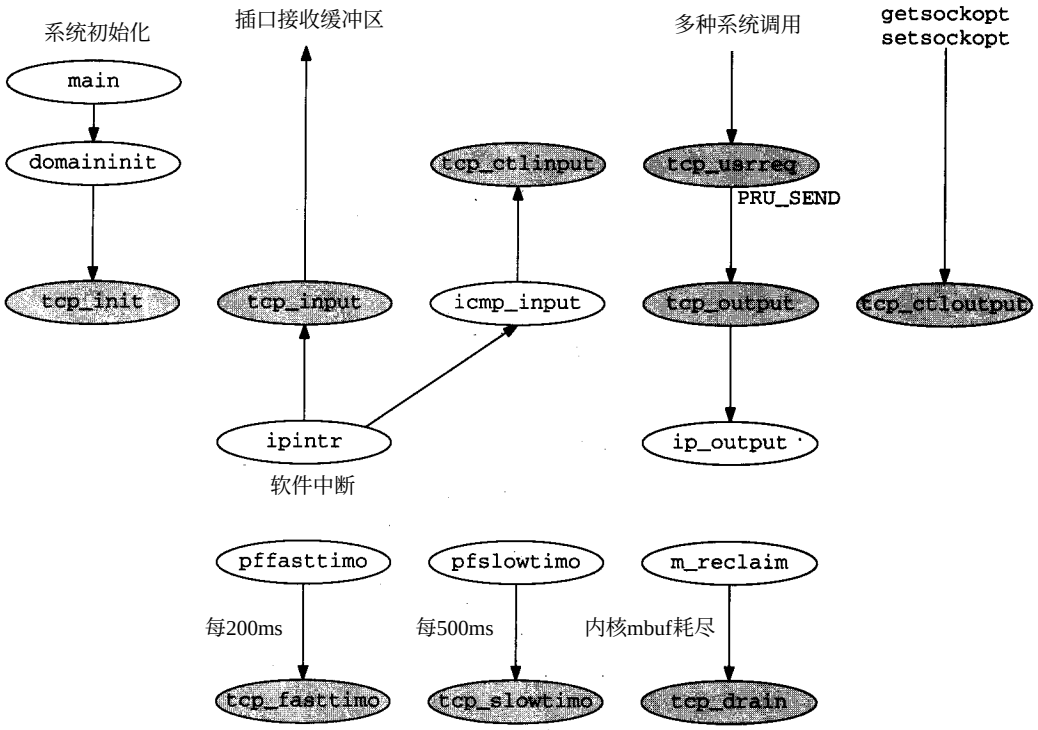

图24-2 TCP函数与其他内核函数间的关系

变 量	数据类型	描 述
tcb	struct inpcb	TCP Internet PCB表头
tcp_last_inpcb	struct inpcb *	指向最后收到报文段的PCB的指针：“后面一个”高速缓存
tcpstat	struct tcpstat	TCP统计数据(图24-4)
tcp_outflags	u_char	输出标志数组，索引为连接状态(图24-16)
tcp_recvspace	u_long	端口接收缓存大小默认值(8192字节)
tcp_sendspace	u_long	端口发送缓存大小默认值(8192字节)
tcp_iss	tcp_seq	TCP发送初始序号(ISS)
tcprexmtthresh	int	ACK重复次数的门限值(3)，触发快速重传
tcp_mssdflt	int	默认MSS值(512字节)
tcp_rttdflt	int	没有数据时RTT的默认值(3秒)
tcp_do_rfrc1323	int	如果为真(默认值)，请求窗口大小和时间戳选项
tcp_now	u_long	用于RFC 1323时间戳实现的500 ms计数器
tcp_keepidle	int	保活：第一次探测前的空闲时间(2小时)
tcp_keepintvl	int	保活：无响应时两次探测的间隔时间(75秒)
tcp_maxidle	int	保活：探测之后、放弃之前的时间(10分钟)

图24-3 后续章节中将介绍的全局变量

24.2.2 统计量

全局结构变量tcpstat中保存了各种TCP统计量，图24-4描述了各统计量的具体含义。在接下来的代码分析过程中，读者会了解到这些计数器数值的变化过程。

tcpstat成员	描　述	SNMP使用
tcps_accepts	被动打开的连接数	•
tcps_closed	关闭的连接数(包括意外丢失的连接)	
tcps_connattempt	试图建立连接的次数(调用connect)	•
tcps_conndrops	在连接建立阶段失败的连接次数(SYN收到之前)	•
tcps_connects	主动打开的连接次数(调用connect成功)	
tcps_delack	延迟发送的ACK数	
tcps_drops	意外丢失的连接数(收到SYN之后)	•
tcps_keepdrops	在保活阶段丢失的连接数(已建立或正等待SYN)	
tcps_keepprobe	保活探测指针发送次数	
tcps_keeptimeo	保活定时器或连接建立定时器超时次数	
tcps_pawsdrop	由于PSWS而丢失的报文段数	
tcps_pcbcachemiss	PCB高速缓存匹配失败次数	
tcps_persisttimeo	持续定时器超时次数	
tcps_predack	对ACK报文首部预测的正确次数	
tcps_preddat	对数据报文首部预测的正确次数	
tcps_rcvackbyte	由收到的ACK报文确认的发送字节数	
tcps_rcvackpack	收到的ACK报文数	
tcps_rcvacktoomuch	收到的对未发送数据进行确认的ACK报文数	
tcps_rcvafterclose	连接关闭后收到的报文数	
tcps_rcvbadoff	收到的首部长度无效的报文数	•
tcps_rcvbadsum	收到的检验和错误的报文数	•
tcps_rcvbyte	连续收到的字节数	
tcps_rcvbyteafterwin	在滑动窗口已满时收到的字节数	
tcps_rcvdupack	收到的重复ACK报文的次数	
tcps_rcvdupbyte	在完全重复报文中收到的字节数	
tcps_rcvduppack	内容完全一致的报文数	
tcps_rcvoobyte	收到失序的字节数	
tcps_rcvoopack	收到失序的报文数	
tcps_rcvpack	顺序接收的报文数	
tcps_rcvpackafterwin	携带数据超出滑动窗口通告值的报文数	
tcps_rcvpartdupbyte	部分内容重复的报文中的重复字节数	
tcps_rcvpartduppack	部分数据重复的报文数	
tcps_rcvshort	长度过短的报文数	•
tcps_rcvtotal	收到的报文总数	•
tcps_rcvwinprobe	收到的窗口探测报文数	
tcps_rcvwinupd	收到的窗口更新报文数	
tcps_rexmttimeo	重传超时次数	
tcps_rttupdated	RTT估算值更新次数	
tcps_segstimed	可用于RTT测算的报文数	
tcps_sndacks	发送的纯ACK报文数(数据长度=0)	
tcps_sndbyte	发送的字节数	
tcps_sndctrl	发送的控制(SYN、FIN、RST)报文数(数据长度=0)	
tcps_sndpack	发送的数据报文数(数据长度>0)	
tcps_sndprobe	发送的窗口探测次数(等待定时器强行加入1字节数据)	
tcps_sndrexmitbyte	重传的数据字节数	•
tcps_sndrexmitpack	重传的报文数	•
tcps_sndtotal	发送的报文总数	•
tcps_sndurg	只携带URG标志的报文数(数据长度=0)	
tcps_sndwinup	只携带窗口更新信息的报文数(数据长度=0)	
tcps_timeoutdrop	由于重传超时而丢失的连接数	

图24-4　tcpstat结构变量中保存的TCP统计量

在命令行输入netstat -s，系统将输出当前TCP的统计值。图24-5的例子显示了主机连续运行30天后，各统计计数器的值。由于某些统计量互相关联——一个保存数据分组数目，另一个保存相应的字节数——图中做了一些简化。例如，表中第二行tcps_snd(pack,byte)实际表示了两个统计量，tcps_sndpack和tcps_sndbyte。

tcps_sndbyte值应为3 722 884 824字节，而不是-22 194 928字节，平均每个数据分组有450字节。类似的，tcps_rcvackbyte值应为3 738 811 552字节，而不是-21 264 360字节(平均每个数据分组565字节)。这些数据之所以被错误地显示，是因为netstat程序中调用printf语句时使用了%d(符号整型)，而非%lu(无符号长整型)。所有统计量均定义为无符号长整型，上面两个统计量的值已接近无符号32位长整型的上限($2^{32}-1$=4 294 967 295)。

netstat -s 输出	tcpstat 成员
10,655,999 packets sent 9,177,823 data packets (-22,194,928 bytes) 257,295 data packets (81,075,086 bytes) retransmitted 862,900 ack-only packets (531,285 delayed) 229 URG-only packets 3,453 window probe packets 74,925 window update packets 279,387 control packets	tcps_sndtotal tcps_snd{pack,byte} tcps_sndrexmit{pack,byte} tcps_sndacks,tcps_delack tcps_sndurg tcps_sndprobe tcps_sndwinup tcps_sndctrl
8,801,953 packets received 6,617,079 acks (for -21,264,360 bytes) 235,311 duplicate acks 0 acks for unsent data 4,670,615 packets (324,965,351 bytes) rcvd in-sequence 46,953 completely duplicate packets (1,549,785 bytes) 22 old duplicate packets 3,442 packets with some dup. data (54,483 bytes duped) 77,114 out-of-order packets (13,938,456 bytes) 1,892 packets (1,755 bytes) of data after window 1,755 window probes 175,476 window update packets 1,017 packets received after close 60,370 discarded for bad checksums 279 discarded for bad header offset fields 0 discarded because packet too short	tcps_rcvtotal tcps_rcvack{pack,byte} tcps_rcvdupack tcps_rcvacktoomuch tcps_rcv{pack,byte} tcps_rcvdup{pack,byte} tcps_pawsdrop tcps_rcvpartdup{pack,byte} tcps_rcvoo{pack,byte} tcps_rcv{pack,byte}afterwin tcps_rcvwinprobe tcps_rcvwindup tcps_rcvafterclose tcps_rcvbadsum tcps_rcvbadoff tcps_rcvshort
144,020 connection requests 92,595 connection accepts 126,820 connections established (including accepts) 237,743 connections closed (including 1,061 drops) 110,016 embryonic connections dropped	tcps_connattempt tcps_accepts tcps_connects tcps_closed,tcps_drops tcps_conndrops
6,363,546 segments updated rtt (of 6,444,667 attempts) 114,797 retransmit timeouts 86 connection dropped by rexmit timeout 1,173 persist timeouts 16,419 keepalive timeouts 6,899 keepalive probes sent 3,219 connections dropped by keepalive	tcps_{rttupdated,segstimed} tcps_rexmttimeo tcps_timeoutdrop tcps_persisttimeo tcps_keeptimeo tcps_keepprobe tcps_keepdrops
733,130 correct ACK header predictions 1,266,889 correct data packet header predictions 1,851,557 cache misses	tcps_predack tcps_preddat tcps_pcbcachemiss

图24-5 TCP统计量样本

24.2.3 SNMP变量

图24-6列出了SNMP TCP组中定义的14个SNMP简单变量，以及与它们相对应的tcpstat结构中的统计量。前四项的常量值在Net/3中定义，计数器tcpCurrEstab用于保存TCP PCB表中Internet PCB的数目。

图24-7列出了tcpTable，即TCP监听表(listener table)。

SNMP变量	tcpstat成员或常量	描　述
tcpRtoAlgorithm	4	用于计算重传定时时限的算法： 1=其他； 2=RTO为固定值； 3=MIL-STD-1778附录B； 4=Van Jacobson的算法；
tcpRtoMin	1000	最小重传定时时限，以毫秒为单位
tcpRtoMax	64000	最大重传定时时限，以毫秒为单位
tcpMaxConn	-1	可支持的最大TCP连接数(-1表示动态设置)
tcpActiveOpens	tcps_connattempt	从CLOSED转换到SYN SENT的次数
tcpPassiveOpens	tcps_accepts	从LISTEN转换到SYN RCVD的次数
tcpAttemptFails	tcps_conndrops	从SYN_SENT或SYN_RCVD转换到CLOSED的次数+从SYN_RCVD转换到LISTEN的次数
tcpEstabResets	tcps_drops	从ESTABLISHED或CLOSE_WAIT转换到CLOSED的次数
tcpCurrEstab	(见正文)	当前位于ESTABLISHED或CLOSE_WAIT状态的连接数
tcpInSegs	tcps_rcvtotal	收到的报文总数
tcpOutSegs	tcps_sndtotal - tcps_sndrexmitpack	发送的报文总数，减去重传报文数
tcpRetransSegs	tcps_sndrexmitpack	重传的报文总数
tcpInErrs	tcps_rcvbadsum + tcps_rcvbadoff + tcps_rcvshort	收到的出错报文总数
tcpOutRsts	(未实现)	RST标志置位的发送报文数

图24-6　tcp组中的简单SNMP变量

index = <tcpConnLocalAddress>.<tcpConnLocalPort>.<tcpConnRemAddress>.<tcpConnRemPort>		
SNMP变量	PCB变量	描　述
tcpConnState	t_state	连接状态：1 = CLOSED, 2=LISTEN, 3 = SYN_SENT, 4 = SYN_RCVD, 5 = ESTABLISHED,6 = FIN_WAIT, 7 = FIN_WAIT_2, 8 = CLOSE_WAIT, 9 = LAST_ACK, 10 = CLOSING, 11 = TIME_WAIT,12 = 删除TCP控制块
tcpConnLocalAddress	inp_laddr	本地IP地址
tcpConnLocalPort	inp_lport	本地端口号
tcpConnRemAddress	inp_faddr	远端IP地址
tcpConnRemPort	inp_fport	远端端口号

图24-7　TCP监听表：tcpTable 中的变量

第一个PCB变量(t_state)来自TCP控制块(图24-13)，其他四个变量来自 Internet PCB (图22-4)。

24.3 TCP 的protosw结构

图24-8列出了TCP protosw结构的成员变量，它定义了TCP协议与系统内其他协议间的交互接口。

成员变量	inetsw[2]	描 述
pr_type	*SOCK_STREAM*	TCP提供字节流传输服务
pr_domain	*&inetdomain*	TCP属于Internet协议族
pr_ptotocol	*IPPROTO_TCP(6)*	填充IP首部的ip_p字段
pr_flags	*PR_CONNREQUIRED|PR_WANTRCVD*	插口层标志，协议处理中忽略
pr_input	*tcp_input*	从IP层接收消息
pr_output	*0*	TCP协议忽略该成员变量
pr_ctlinput	*tcp_ctlinput*	处理ICMP错误的控制输入函数
pr_ctloutput	*tcp_ctloutput*	在进程中响应管理请求
pr_usrreg	*tcp_usrreq*	在进程中响应通信请求
pr_init	*tcp_init*	TCP初始化
pr_fasttimo	*tcp_fasttimo*	快超时函数，每200 ms调用一次
pr_slowtimo	*tcp_slowtimo*	慢超时函数，每500 ms调用一次
pr_drain	*tcp_drain*	内核mbuf耗尽时调用
pr_sysctl	*0*	TCP协议忽略该成员变量

图24-8 TCP protosw 结构

24.4 TCP的首部

tcphdr结构定义了TCP首部。图24-9给出了tcphdr结构的定义，图24-10描述了TCP首部。

tcp.h

```
40 struct tcphdr {
41     u_short th_sport;           /* source port */
42     u_short th_dport;           /* destination port */
43     tcp_seq th_seq;             /* sequence number */
44     tcp_seq th_ack;             /* acknowledgement number */
45 #if BYTE_ORDER == LITTLE_ENDIAN
46     u_char  th_x2:4,            /* (unused) */
47             th_off:4;           /* data offset */
48 #endif
49 #if BYTE_ORDER == BIG_ENDIAN
50     u_char  th_off:4,           /* data offset */
51             th_x2:4;            /* (unused) */
52 #endif
53     u_char  th_flags;           /* ACK, FIN, PUSH, RST, SYN, URG */
54     u_short th_win;             /* advertised window */
55     u_short th_sum;             /* checksum */
56     u_short th_urp;             /* urgent offset */
57 };
```

tcp.h

图24-9 tcphdr 结构

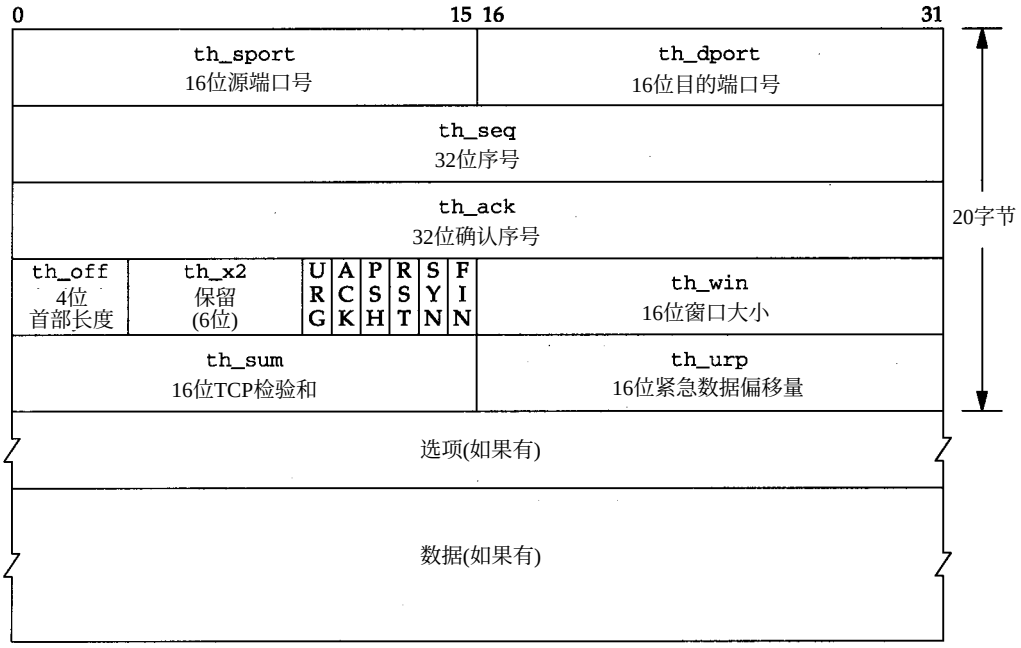

图24-10 TCP首部及可选数据

大多数RFC文档，相关书籍(包括卷1)和接下来要讨论的TCP实现代码，都把th_urp称为“紧急指针(urgent pointer)”。更准确的名称应该是“紧急数据偏移量(urgent offset)”，因为这个字段给出的16 bit无符号整数值，与th_seq序号字段相加后，得到发送的紧急数据最后一个八位组的32 bit序号(关于该序号应该是紧急数据最后一个字节的序号，或者是紧急数据结束后的第一个字节的序号，一直存在着争议。但就我们目前的讨论而言，这一点无关紧要)。图24-13中，TCP代码把保存紧急数据最后一个八位组的32 bit序号的snd_up正确地称为“紧急数据发送指针”。如果将TCP首部的16 bit偏移量也称为“指针”，容易引起误解。在练习26.6中，我们重申了“紧急指针”和“紧急数据偏移量”间的区别。

TCP首部中4 bit的首部长度、接着的6 bit的保留字段和6 bit的码元标志，在C结构中定义为两个4 bit的比特字段，和紧跟的一个8 bit字节。为了处理两个比特字段在8 bit字节中的存放次序，C代码根据不同的主机字节存储顺序使用了#ifdef语句。

还请注意，TCP中称4 bit的th_off为“首部长度”，而C代码中称之为“数据偏移量”。两种名称都正确，因为它表示TCP首部的长度，包括可选项，以32 bit为单位，也就是指向用户数据第一个字节的偏移量。

th_flags成员变量包括6个码元标志比特，通过图24-11中定义的名称读写。

Net/3中，TCP首部通常意味着“IP首部 + TCP首部”。tcp_input处理收到的IP数据报和tcp_output构造待发送的IP数据报时都采用了这一思想。图24-12中给出了tcpiphdr结构的定义，形式化地描述了组合的IP/TCP首部。

38-58 图23-19给出的ipovly结构定义了20字节长度的IP首部。通过前面章节的讨论可知，尽管长度相同(20字节)，但这个结构并不是一个真正的IP首部。

th_flags	描　述
TH_ACK	确认序号(th_ack)有效
TH_FIN	发送方字节流结束
TH_PUSH	接收方应该立即将数据提交给应用程序
TH_RST	连接复位
TH_SYN	序号同步(建立连接)
TH_URG	紧急数据偏移量(th_urp)有效

图24-11 th_flags 值

tcpip.h

```
38 struct tcpiphdr {
39     struct ipovly ti_i;          /* overlaid ip structure */
40     struct tcphdr ti_t;          /* tcp header */
41 };

42 #define ti_next     ti_i.ih_next
43 #define ti_prev     ti_i.ih_prev
44 #define ti_x1       ti_i.ih_x1
45 #define ti_pr       ti_i.ih_pr
46 #define ti_len      ti_i.ih_len
47 #define ti_src      ti_i.ih_src
48 #define ti_dst      ti_i.ih_dst
49 #define ti_sport    ti_t.th_sport
50 #define ti_dport    ti_t.th_dport
51 #define ti_seq      ti_t.th_seq
52 #define ti_ack      ti_t.th_ack
53 #define ti_x2       ti_t.th_x2
54 #define ti_off      ti_t.th_off
55 #define ti_flags    ti_t.th_flags
56 #define ti_win      ti_t.th_win
57 #define ti_sum      ti_t.th_sum
58 #define ti_urp      ti_t.th_urp
```

tcpip.h

图24-12 tcpiphdr 结构定义：组合的IP/TCP首部

24.5 TCP的控制块

在图22-1中我们看到，除了标准的Internet PCB外，TCP还有自己专用的控制块，tcpcb结构，而UDP则不需要专用控制块，它的全部控制信息都已包含在Internet PCB中。

TCP控制块较大，需占用140字节。从图22-1中可看到，Internet PCB与TCP控制块彼此对应，都带有指向对方的指针。图24-13给出了TCP控制块的定义。

tcp_var.h

```
41 struct tcpcb {
42     struct tcpiphdr *seg_next;   /* reassembly queue of received segments */
43     struct tcpiphdr *seg_prev;   /* reassembly queue of received segments */
44     short   t_state;             /* connection state (Figure 24.16) */
45     short   t_timer[TCPT_NTIMERS];  /* tcp timers (Chapter 25) */
46     short   t_rxtshift;          /* log(2) of rexmt exp. backoff */
47     short   t_rxtcur;            /* current retransmission timeout (#ticks) */
48     short   t_dupacks;           /* #consecutive duplicate ACKs received */
49     u_short t_maxseg;            /* maximum segment size to send */
50     char    t_force;             /* 1 if forcing out a byte (persist/OOB) */
51     u_short t_flags;             /* (Figure 24.14) */
```

图24-13 tcpcb 结构：TCP控制块

```
52      struct tcpiphdr *t_template;     /* skeletal packet for transmit */
53      struct inpcb *t_inpcb;       /* back pointer to internet PCB */
54  /*
55   * The following fields are used as in the protocol specification.
56   * See RFC783, Dec. 1981, page 21.
57   */
58  /* send sequence variables */
59      tcp_seq snd_una;             /* send unacknowledged */
60      tcp_seq snd_nxt;             /* send next */
61      tcp_seq snd_up;              /* send urgent pointer */
62      tcp_seq snd_wl1;             /* window update seg seq number */
63      tcp_seq snd_wl2;             /* window update seg ack number */
64      tcp_seq iss;                 /* initial send sequence number */
65      u_long  snd_wnd;             /* send window */
66  /* receive sequence variables */
67      u_long  rcv_wnd;             /* receive window */
68      tcp_seq rcv_nxt;             /* receive next */
69      tcp_seq rcv_up;              /* receive urgent pointer */
70      tcp_seq irs;                 /* initial receive sequence number */
71  /*
72   * Additional variables for this implementation.
73   */
74  /* receive variables */
75      tcp_seq rcv_adv;             /* advertised window by other end */
76  /* retransmit variables */
77      tcp_seq snd_max;             /* highest sequence number sent;
78                                    * used to recognize retransmits */
79  /* congestion control (slow start, source quench, retransmit after loss) */
80      u_long  snd_cwnd;            /* congestion-controlled window */
81      u_long  snd_ssthresh;        /* snd_cwnd size threshhold for slow start
82                                    * exponential to linear switch */
83  /*
84   * transmit timing stuff.  See below for scale of srtt and rttvar.
85   * "Variance" is actually smoothed difference.
86   */
87      short   t_idle;              /* inactivity time */
88      short   t_rtt;               /* round-trip time */
89      tcp_seq t_rtseq;             /* sequence number being timed */
90      short   t_srtt;              /* smoothed round-trip time */
91      short   t_rttvar;            /* variance in round-trip time */
92      u_short t_rttmin;            /* minimum rtt allowed */
93      u_long  max_sndwnd;          /* largest window peer has offered */
94  /* out-of-band data */
95      char    t_oobflags;          /* TCPOOB_HAVEDATA, TCPOOB_HADDATA */
96      char    t_iobc;              /* input character, if not SO_OOBINLINE */
97      short   t_softerror;         /* possible error not yet reported */
98  /* RFC 1323 variables */
99      u_char  snd_scale;           /* scaling for send window (0-14) */
100     u_char  rcv_scale;           /* scaling for receive window (0-14) */
101     u_char  request_r_scale;     /* our pending window scale */
102     u_char  requested_s_scale;   /* peer's pending window scale */
103     u_long  ts_recent;           /* timestamp echo data */
104     u_long  ts_recent_age;       /* when last updated */
105     tcp_seq last_ack_sent;       /* sequence number of last ack field */
106 };
107 #define intotcpcb(ip)    ((struct tcpcb *)(ip)->inp_ppcb)
108 #define sototcpcb(so)    (intotcpcb(sotoinpcb(so)))
```

tcp_var.h

图24-13 (续)

现在暂不讨论上述成员变量的具体含义，在后续代码中遇到时再详细分析。

图24-14列出了t_flags变量的可选值。

t_flags	描　述
TF_ACKNOW	立即发送ACK
TF_DELACK	延迟发送ACK
TF_NODELAY	立即发送用户数据，不等待形成最大报文段 (禁止Nagle算法)
TF_NOOPT	不使用TCP选项(永不填充TCP选项字段)
TF_SENTFIN	FIN已发送
TF_RCVD_SCALE	对端在SYN报文中发送窗口变化选项时置位
TF_RCVD_TSTMP	对端在SYN报文中发送时间戳选项时置位
TF_REQ_SCALE	已经/将要在SYN报文中请求窗口变化选项
TF_REQ_TSTMP	已以/将要在SYN中请求时间戳选项

图24-14　t_flags 取值

24.6　TCP的状态变迁图

TCP协议根据连接上到达报文的不同类型，采取相应动作，协议规程可抽象为图 24-15所示的有限状态变迁图。读者在本书的扉页前也可找到这张图，以便在阅读有关TCP的章节时参考。

图中的各种状态变迁组成了 TCP有限状态机。尽管TCP协议允许从LISTEN状态直接变迁到SYN_SENT状态，但使用SOCKET API编程时这种变迁不可实现 (调用listen后不可以调用connect)。

TCP控制块的成员变量t_state保存一个连接的当前状态，可选值如图 24-16所示。

图中还定义了tcp_outflags数组，保存了处于对应连接状态时 tcp_output将使用的输出标志。

图24-16还列出了与符号常量相对应的数值，因为在代码中将利用它们之间的数值关系。例如，有下面两个宏定义：

```
#define TCPS_HAVERCVDSYN(s) ((s)>=TCPS_SYN_RECEIVED)
#define TCPS_HAVERCVDFIN(s) ((s)>=TCPS_TIME_WAIT)
```

类似地，连接未建立时，即t_state小于TCPS_ESTABLISHED时，tcp_notify处理ICMP差错的方式也不同。

> TCPS_HAVERCVDSYN的命名是正确的，但TCPS_HAVERCVDFIN则可能引起误解，因为在 CLOSE_WAIT、CLOSING和LAST_ACK状态也会收到FIN。我们将在第29章中遇到该宏。

半关闭

当进程调用shutdown且第二个参数设为1时，称为“半关闭”。TCP发送FIN，但允许进程在同一端口上继续接收数据 (卷1的18.5节中举例介绍了TCP的半关闭)。

例如，尽管图 24-15中只在ESTABLISHED状态标注了“数据传输”，但如果进程执行了“半关闭”，则连接变迁到FIN_WAIT_1状态和其后的FIN_WAIT_2状态，在这两个特定状态中，进程仍然可以接收数据。

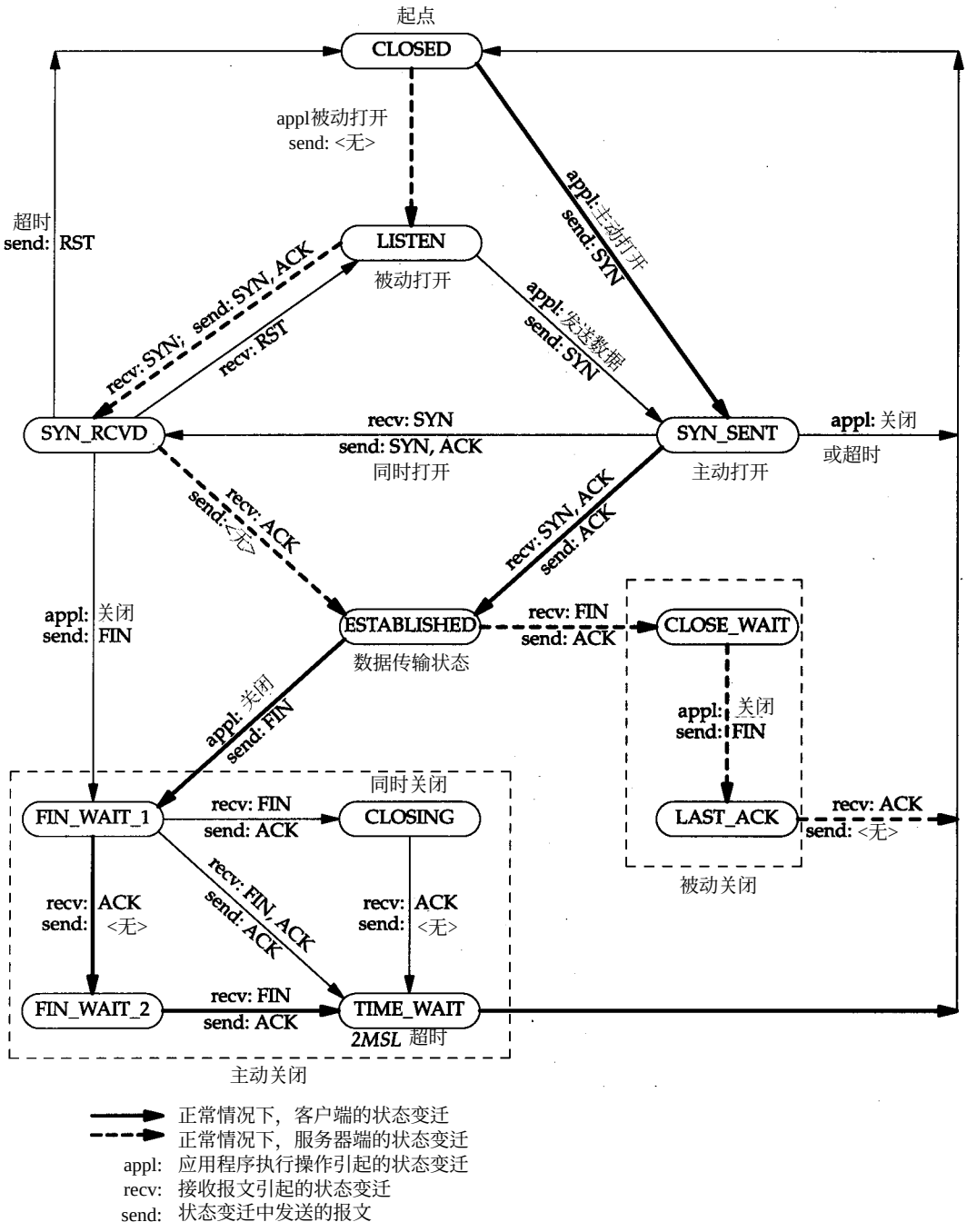

图24-15 TCP状态变迁图

24.7 TCP的序号

TCP连接上传输的每个数据字节，以及SYN、FIN等控制报文都被赋予一个32 bit的序号。TCP首部的序号字段(图24-10)填充了报文段第一个数据字节的32 bit的序号，确认号字段填充

了发送方希望接收的下一序号，确认已正确接收了所有序号小于等于确认号减 1的数据字节。换言之，确认号是ACK发送方等待接收的下一序号。只有当报文首部的 ACK标志置位时，确认序号才有效。读者将看到，除了在主动打开首次发送 SYN时(SYN_SENT状态，参见图24-16中的tcp_outflags[2])或在某些RST报文段中，ACK标志总是被置位的。

t_state	值	描 述	tcp_outflags[]
TCPS_CLOSED	0	关闭	TH_RST | TH_ACK
TCPS_LISTEN	1	监听连接请求(被动打开)	0
TCPS_SYN_SENT	2	已发送SYN(主动打开)	TH_SYN
TCPS_SYN_RECEIVED	3	已发送并接收SYN；等待ACK	TH_SYN | TH_ACK
TCPS_ESTABLISHED	4	连接建立(数据传输)	TH_ACK
TCPS_CLOSE_WAIT	5	已收到FIN，等待应用程序关闭	TH_ACK
TCPS_FIN_WAIT_1	6	已关闭，发送FIN；等待ACK和FIN	TH_FIN | TH_ACK
TCPS_CLOSING	7	同时关闭；等待ACK	TH_FIN | TH_ACK
TCPS_LAST_ACK	8	收到的FIN已关闭；等待ACK	TH_FIN | TH_ACK
TCPS_FIN_WAIT_2	9	已关闭，等待FIN	TH_ACK
TCPS_TIME_WAIT	10	主动关闭后2MSL等待状态	TH_ACK

图24-16 t_state 取值

由于TCP连接是全双工的，每一端都必须为两个方向上的数据流维护序号。 TCP控制块中(图24-13)有13个序号：8个用于数据发送(发送序号空间)，5个用于数据接收(接收序号空间)。

图24-17给出了发送序号空间中4个变量间的关系：snd_wnd、snd_una、snd_nxt和snd_max。这个例子列出了数据流的第1~第11字节。

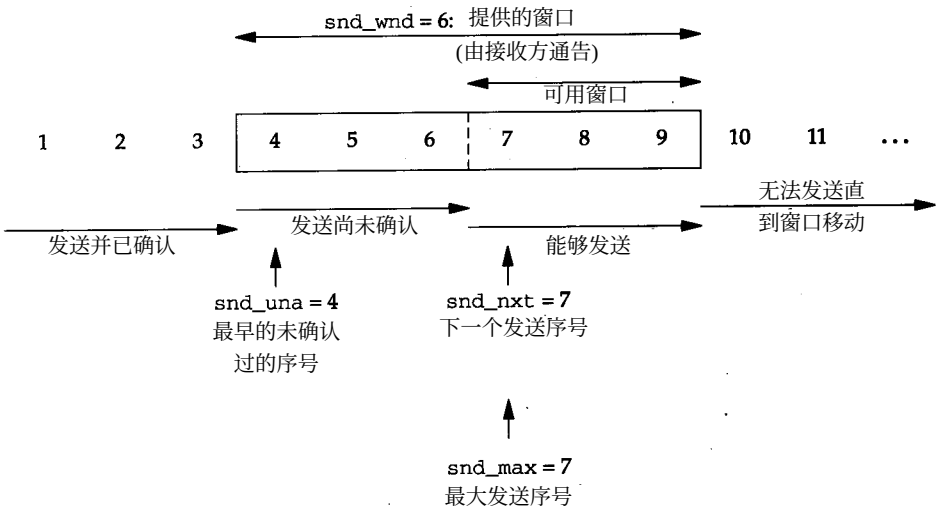

图24-17 发送序号空间举例

一个有效的ACK序号必须满足：

```
snd_una < 确认序号 <=  snd_max
```

图24-17的例子中，一个有效 ACK的确认号必须是 5、6或7。如果确认号小于或等于snd_una，则是一个重复的ACK。它确认了已确认过的八位组，否则 snd_una不会递增超过那些序号。

tcp_output中有多处用到下面的测试，如果正发送的是重传数据，则表达式为真：

```
snd_nxt < snd_max
```

图24-18给出了图24-17中连接的另一端：接收序号空间，图中假定还未收到序号为 4、5、6的报文，标出了三个变量rcv_nxt、rcv_wnd和rcv_adv。

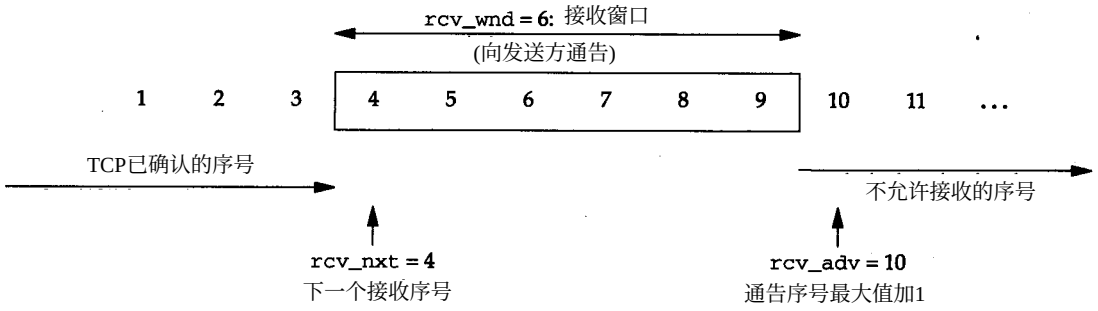

图24-18 接收序号空间举例

如果接收报文段中携带的数据落在接收窗口内，则该报文段是一个有效报文段。换言之，下面两个不等式中至少要有一个为真。

```
rcv_nxt <= 报文段起始序号  < rcv_nxt + rcv_wnd
rcv_nxt <=报文段终止序号 <  rcv_nxt + rcv_wnd
```

报文段起始序号就是TCP首部的序号字段，ti_seq。终止序号是序号字段加上 TCP数据长度后减1。

例如，图24-19中的TCP报文段，携带了图24-17中发送的三个字节，序号分别是4、5和6。

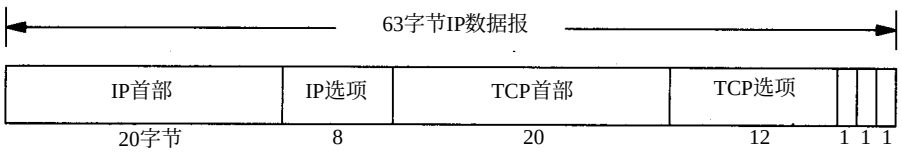

图24-19 TCP报文段在IP数据报中传输

假定IP数据报中有8字节的IP任选项和12字节的TCP任选项。图12-20列出了各有关变量的取值。

变　量	值	描　述
ip_hl	7	IP首部+IP任选项长度，以32 bit为单位(=28字节)
ip_len	63	IP数据报长度，以字节为单位 (20+8 +20+12+3)
ti_off	8	TCP首部+TCP任选项长度，以32 bit为单位(=32字节)
ti_seq	4	用户数据第一个字节的序号
ti_len	3	TCP数据的字节数：ip_len-(ip_hl×4)-(ti_off×4)
	6	用户数据最后一个字节的序号：ti_seq+ti_len-1

图24-20 图24-19中各变量的取值

ti_len并非TCP首部的字段，而是在对接收到的首部计算检验和及完成验证之后，根据图24-20中的算式得到的结果，存储到外加的 IP结构中(图24-12)。图中最后一个值并不存储到变量中，而是在需要时直接从其他值中通过计算得到。

1. 序号取模运算

TCP必须处理的一个问题是序号来自有限的32位取值空间：0~4 294 967 295。如果某个TCP连接传输的数据量超过2^{32}字节，序号从4 294 967 295回绕到0，将出现重复序号。

即使传输数据量小于2^{32}字节，仍可能遇到同样的问题，因为连接的初始序号并不一定从0开始。各数据流方向上的初始序号可以是0~4 294 967 295之间的任何值。这个问题使序号复杂化。例如，序号1可能大于序号4 294 967 295。

在tcp.h中，TCP序号定义为unsigned long

```
typedef u_long tcp_seq;
```

图24-21定义了4个用于序号比较的宏。

tcp_seq.h

```
40 #define SEQ_LT(a,b)     ((int)((a)-(b)) < 0)
41 #define SEQ_LEQ(a,b)    ((int)((a)-(b)) <= 0)
42 #define SEQ_GT(a,b)     ((int)((a)-(b)) > 0)
43 #define SEQ_GEQ(a,b)    ((int)((a)-(b)) >= 0)
```

tcp_seq.h

图24-21 TCP序号比较宏

2. 举例——序号比较

下面这个例子说明了TCP序号的操作方式。假定序号只有3 bit，0~7。图24-22列出了全部8个序号和相应的二进制补码(为求二进制补码，将二进制码中的所有0变为1，所有1变为0，最后再加1)。给出补码形式，是因为a-b =a+(b的补码)。

x	二进制码	二进制补码	0-x	1-x	2-x
0	000	000	000	001	010
1	001	111	111	000	001
2	010	110	110	111	000
3	011	101	101	110	111
4	100	100	100	101	110
5	101	011	011	100	101
6	110	010	010	011	100
7	111	001	001	010	011

图24-22 3 bit序号举例

表中最后三栏分别是0-x、1-x和2-x。在这三栏中，如果定义计算结果是带符号整数(注意图24-21中的四个宏，计算结果全部强制转换为int)，那么最高位为1表示值小于0(SEQ_LT宏)，最高位为0且值不为0表示大于0(SEQ_GT宏)。最后三栏中以横线分隔开四个负值和四个非负值。

请注意图24-22中的第四栏(标注“0-x”)，可看出0小于1、2、3和4(最高位比特为1)，而0大于5、6和7(最高位比特为0且结果非0)。图24-23显示了这种关系。

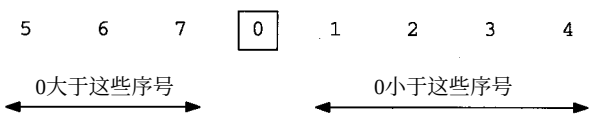

图24-23 3 bit的TCP序号的比较

图24-22中的第五栏(1 - x)也存在类似的关系，如图24-24所示。

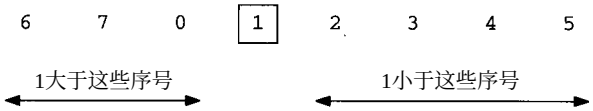

图24-24 3 bit的TCP序号的比较

图24-25是上面两图的另一种表示形式，使用圆环强调了序号的回绕现象。

图24-25 图24-23和图24-24的另一种表示形式

就TCP而言，通过序号比较来确定给定序号是新序号或重传序号。例如，在图 24-24的例子中，如果TCP正等待的序号为1，但到达序号为6，通过前面介绍的计算可知 6小于1，从而判定这是重传的数据，可予以丢弃。但如果到达序号为 5，因为5大于1，TCP判定这是新数据，予以保存，并继续等待序号为2、3和4的八位组(假定序号为5的数据字节落在接收窗口内)。

图24-26扩展了图24-25中左边的圆环，用TCP 32 bit的序号替代了3 bit的序号。

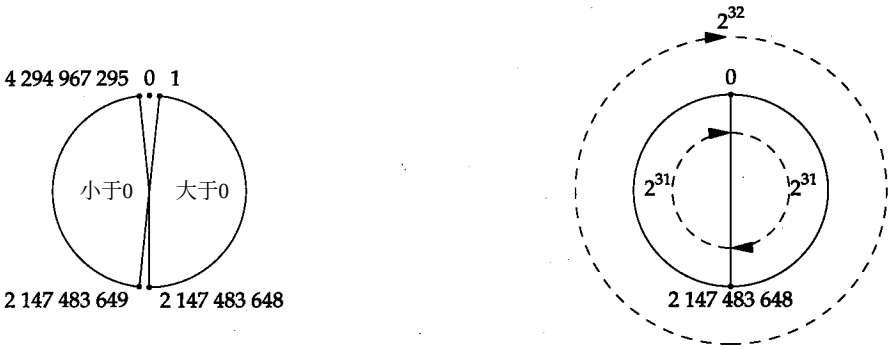

图24-26 与序号0比较：采用32 bit序号

图24-26右边的圆环强调了32 bit序号空间的一半有2^{31}个可用数字。

24.8 tcp_init函数

系统初始化时，domaininit函数调用TCP的初始化函数：tcp_init (图24-27)。

1. 设定初始发送序号

初始发送序号(ISS)，tcp_iss，被初始化为1。请注意，代码注释指出，这是错误的。后面讨论TCP的“平静时间(quite time)”时，将简单介绍这一选择的原因。请读者自行与图 7-23中IP标识符的初始化做比较，后者使用了当天的时钟。

tcp_subr.c

```
43 void
44 tcp_init()
45 {
46     tcp_iss = 1;                    /* wrong */
47     tcb.inp_next = tcb.inp_prev = &tcb;

48     if (max_protohdr < sizeof(struct tcpiphdr))
49                 max_protohdr = sizeof(struct tcpiphdr);
50     if (max_linkhdr + sizeof(struct tcpiphdr) > MHLEN)
51                 panic("tcp_init");
52 }
```

tcp_subr.c

图24-27 tcp_init 函数

2. TCP Internet PCB链表初始化

PCB首部(tcb)的previous指针和next指针都指向自己，这是一个空的双向链表。tcb PCB的其余成员均初始化为0(所有未明确初始化的全局变量均设为0)。事实上，除链表外，在该PCB首部中只用了一个字段inp_lport：下一个分配的TCP临时端口号。TCP使用的第一个临时端口号应为1024，练习22.4的解答中给出了原因。

3. 计算最大协议首部长度

到目前为止，讨论过的协议首部的长度最大不超过40字节，max_protohdr设为40(组合的IP/TCP首部长度，不带任何可选项)。图7-17定义了该变量。如果max_linkhdr (通常为16)加40后大于放入单个mbuf中带首部的数据报的数据长度(100字节，图2-7中的MHLEN)，内核将告警。

MSL和平静时间的概念

TCP协议要求如果主机崩溃，且没能保存打开TCP连接上最后使用的序号，则重启后在一个MSL(2分钟，平静时间)内，不能发送任何TCP报文段。目前，基本没有TCP实现能够在系统崩溃或操作员关机时保存这些信息。

MSL是最大报文段生存时间(maximum segment lifetime)，指任何报文段被丢弃前在网络中能够存在的最大时间。不同的实现可选择不同的MSL。连接主动关闭后，将在CLOSE_WAIT状态等待2个MSL时间(图24-15)。

RFC 793(Postel 1981c)建议MSL设定为2分钟，但Net/3实现中MSL设为30秒(图25-3中定义的常量TCPTV_MSL)。

如果报文段在网络中出现延迟，协议会出现问题(RFC 793称之为漫游重复(wandering duplicate)。假定Net/3系统启动时tcp_iss置为1(图24-27)，经过一段时间，在序号刚刚回绕时系统崩溃。后面25.5节中将介绍，tcp_iss每秒增加128 000，即重启后需经过9.3小时序号才会回绕。此外，每发送一个connect，tcp_iss将增加64 000，因此序号回绕时间必然早于9.3小时。下面的例子说明了老的报文段怎样被错误地发送到现在的连接上。

1) 一个客户和服务器建立了一个连接。客户的端口号是1024，发送了一个序号为2的报文段。该报文段在传送途中陷入路径循环，未能到达服务器。这个报文段成为“漫游重复”报文段。

2) 客户重发该报文段，序号依旧为2。重发报文段到达服务器。

3) 客户关闭连接。

4) 客户主机崩溃。

5) 客户主机在崩溃后40秒重启，TCP初始化tcp_iss为1。

6) 同一客户和同一服务器之间立即建立了一条新的连接，使用了同样的端口号：客户端口号为1024，服务器方依然是其预知的端口号。客户发送的SYN中初始序号置为1。这条新的使用同样端口对的连接称为原有连接的化身(incarnation)。

7) 步聚1中的漫游重复报文段最终到达服务器，并被认为是新建连接中的合法报文段，尽管它实际上属于原有连接。

图24-28列出了上述步骤发生的时间顺序。

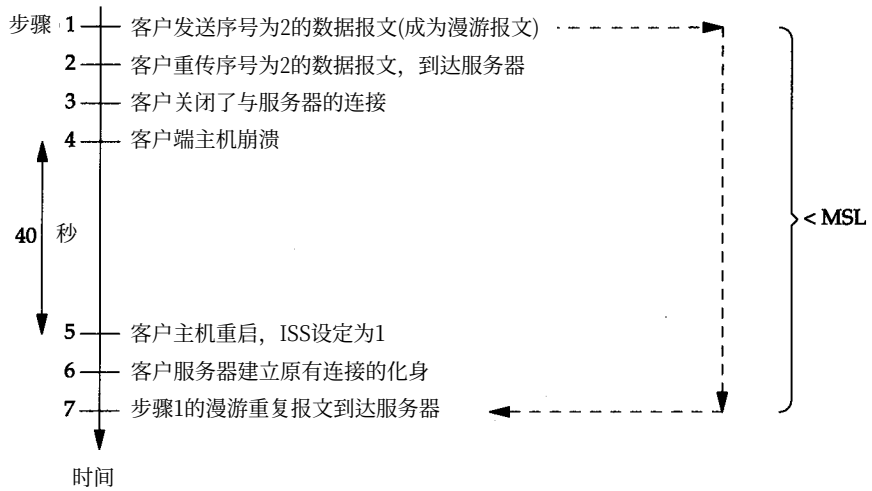

图24-28 示例：旧报文段到达原有连接的化身

即使系统重启后，TCP通过当前时钟计算ISS，问题同样存在。无论原有连接的ISS设为多少，由于序号会回绕，完全有可能重启后新建连接的ISS接近等于重启前原有连接最后使用的序号。

除了保存重启前所有已建连接的序号，解决这个问题的唯一方法就是重启后TCP在MSL内保持平静(不发送任何报文段)。尽管问题有可能出现，但绝大多数TCP中并未实现相应的解决方法，因为多数主机仅重启时间就要长于MSL。

24.9 小结

本章概要介绍了接下来的6章中将要讨论的TCP源代码。TCP为每条连接建立自己的控制块，保存该连接的所有变量和状态信息。

定义了TCP的状态变迁图，TCP在哪些条件下从一个状态变迁到另一个状态，每次变迁过程中发送和接收了哪些报文段。状态变迁图还显示了连接建立和终止的过程。在后续TCP讨论中会经常引用该图。

TCP连接上传输的每个数据字节都有相应的序号，TCP在连接控制块中维护多个序号：有些用于发送，有些用于接收(TCP工作于全双工方式)。由于序号来自有限的32 bit空间，会从

最大值回绕到0。本章解释了如何使用小于和大于测试来比较序号，在后续的 TCP代码中将不断遇到序号的比较。

最后介绍了最简单的 TCP函数，`tcp_init`，完成对Internet PCB的TCP链表的初始化。此外，还讨论了初始发送序号的选取问题。

习题

24.1 研究图24-5中的统计数据，计算每条连接上发送和接收的平均字节数。

24.2 在`tcp_init`中，内核告警是否合理？

24.3 执行`netstat -a`了解你的系统当前有多少个活跃的 TCP端点。