# TCP 提高实验

说明：

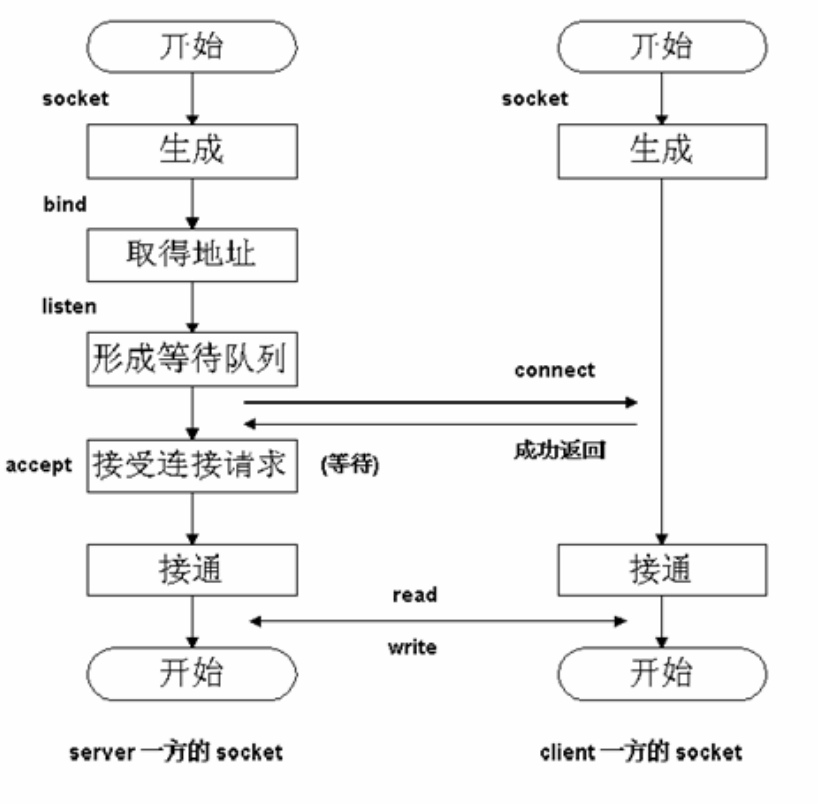
1.本文中所有源代码基于 linux-2.6.39.2 内核；

2.源码所在文件在源码前均有说明，格式为：“目录/文件”；

3.源码在 source insight 软件中注释后用 PDF 虚拟打印机输出并拷贝于此。

## 目录

# 1. SOCKET 通信过程




说明：略

## 2. 函数调用关系

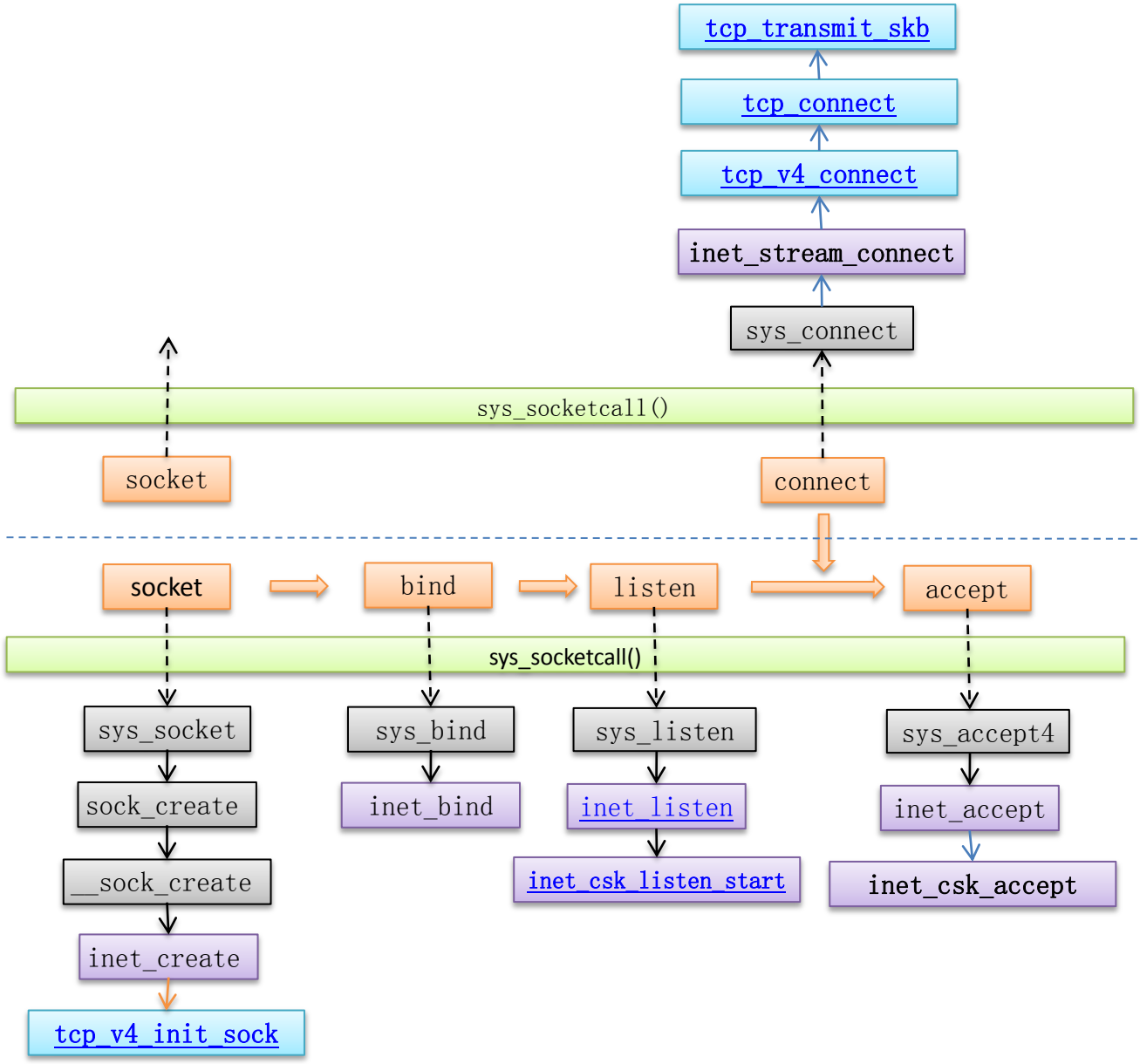



图 1 从 socket 创建到建立连接的过程

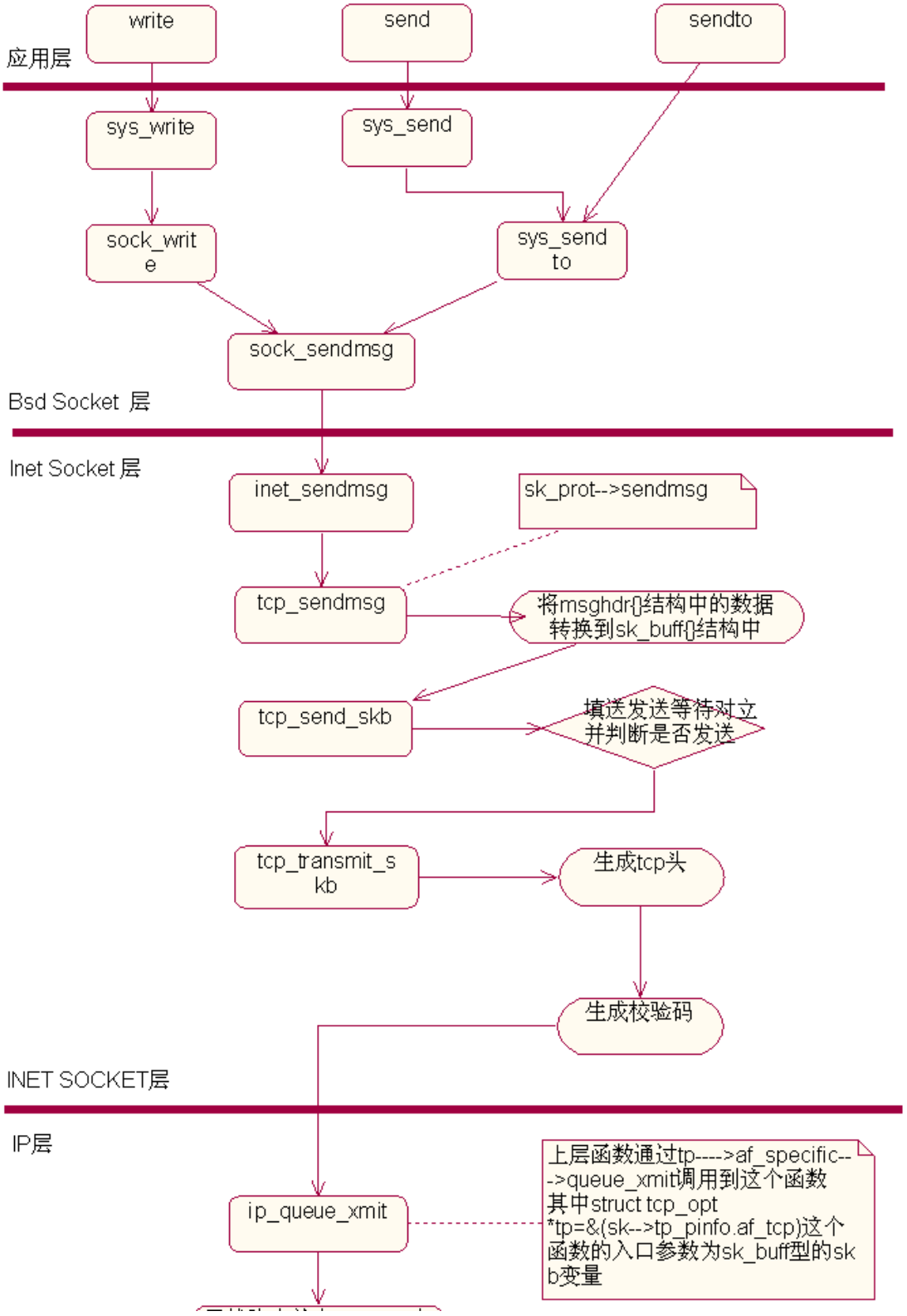



图 2 SOCKET 数据发送过程

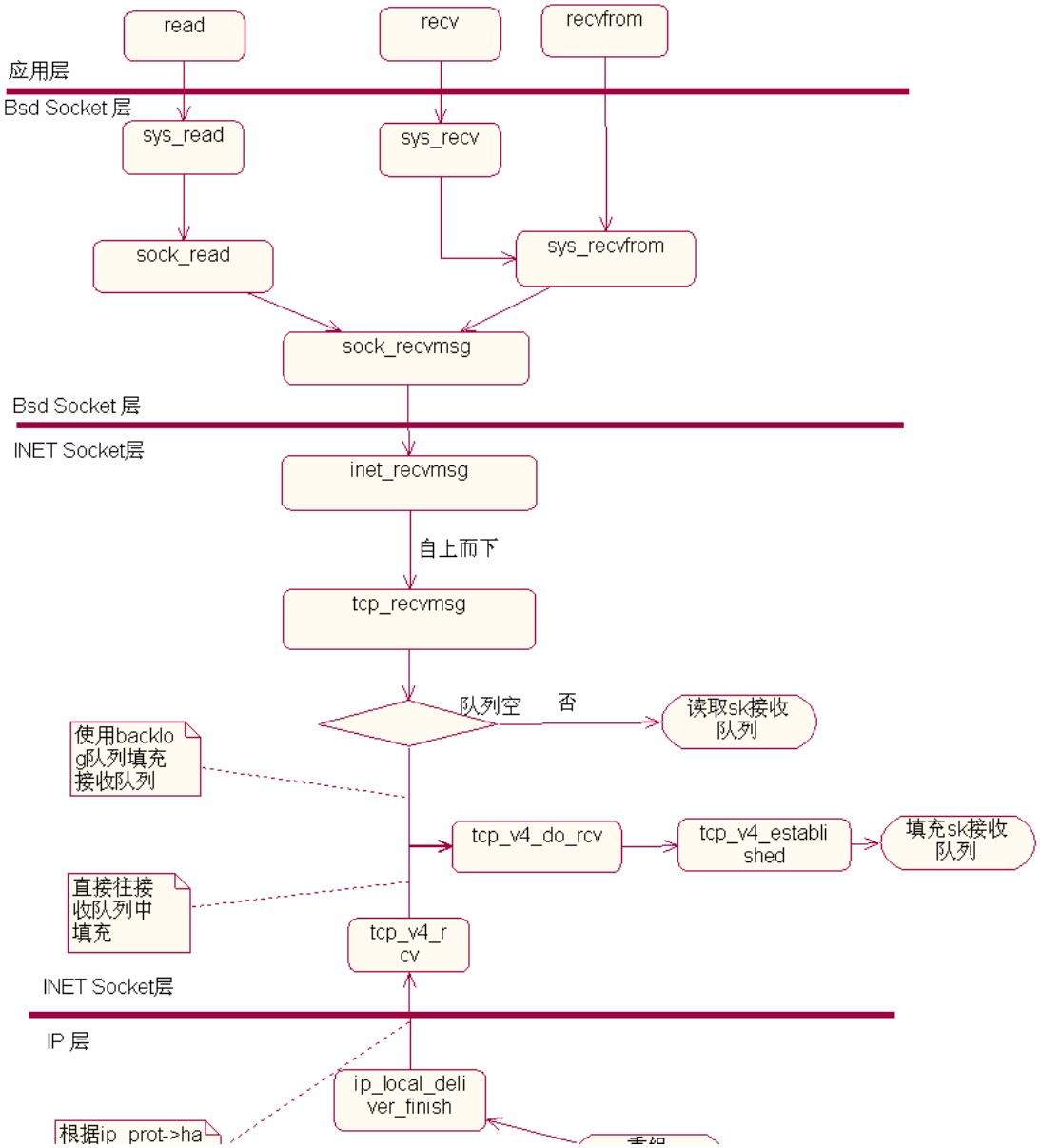



图 3 SOCKET 数据接收过程

## 2.1 socket 函数调用流程

extern int __socket (int __domain, int __type, int __protocol) attribute_hidden;
SYS_SOCKET
SYSCALL_DEFINE2(socketcall, int, call, unsigned long __user *, args)
asmlinkage long sys_socket(int, int, int);
SYSCALL_DEFINE3(socket, int, family, int, type, int, protocol)
int sock_create(int family, int type, int protocol, struct socket **res)
int __sock_create(struct net *net, int family, int type, int protocol,
struct socket **res, int kern)
static int inet_create(struct net *net, struct socket *sock, int protocol,

int kern)
static int **tcp_v4_init_sock**(struct sock *sk)

## 2.2 bind 函数调用流程

int __bind (fd, addr, len) int fd;__CONST_SOCKADDR_ARG addr;socklen_t len;
SYS_BIND
SYSCALL_DEFINE2(socketcall, int, call, unsigned long __user *, args)
asmlinkage long sys_bind(int, struct sockaddr __user *, int);
SYSCALL_DEFINE3(bind, int, fd, struct sockaddr __user *, umyaddr, int, addrlen)
int **inet_bind**(struct socket *sock, struct sockaddr *uaddr, int addr_len)

## 2.3 listen 函数调用流程

int __listen (fd, n)int fd;int n;
SYS_LISTEN
SYSCALL_DEFINE2(socketcall, int, call, unsigned long __user *, args)
asmlinkage long sys_listen(int, int);
SYSCALL_DEFINE2(listen, int, fd, int, backlog)
int **inet_listen**(struct socket *sock, int backlog)
int **inet_csk_listen_start**(struct sock *sk, const int nr_table_entries)

## 2.4 connect 函数调用流程

extern int connect (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len);
SYS_CONNECT
SYSCALL_DEFINE2(socketcall, int, call, unsigned long __user *, args)
asmlinkage long sys_connect(int, struct sockaddr __user *, int);
SYSCALL_DEFINE3(connect, int, fd, struct sockaddr __user *, uservaddr,int, addrlen)
int **inet_stream_connect**(struct socket *sock, struct sockaddr *uaddr,
int addr_len, int flags)
int **tcp_v4_connect**(struct sock *sk, struct sockaddr *uaddr, int addr_len)
int **tcp_connect**(struct sock *sk)
static int **tcp_transmit_skb**(struct sock *sk, struct sk_buff *skb, int clone_it,
gfp_t gfp_mask)

## 2.5 accept 函数调用流程

int accept (fd, addr, addr_len)int fd; __SOCKADDR_ARG addr;socklen_t *addr_len;
SYS_ACCEPT
SYSCALL_DEFINE2(socketcall, int, call, unsigned long __user *, args)
asmlinkage long sys_accept4(int, struct sockaddr __user *, int __user *, int);
SYSCALL_DEFINE4(accept4, int, fd, struct sockaddr __user *, upeer_sockaddr,
int __user *, upeer_addrlen, int, flags)
int **inet_accept**(struct socket *sock, struct socket *newsock, int flags)
struct sock ***inet_csk_accept**(struct sock *sk, int flags, int *err)

## 2.6 send 函数调用流程

extern ssize_t send (int __fd, __const void *__buf, size_t __n, int __flags);
SYS_SEND
SYSCALL_DEFINE2(socketcall, int, call, unsigned long __user *, args)
asmlinkage long sys_send(int, void __user *, size_t, unsigned);
SYSCALL_DEFINE4(send, int, fd, void __user *, buff, size_t, len,
unsigned, flags)
asmlinkage long sys_sendto(int, void __user *, size_t, unsigned,
struct sockaddr __user *, int);
SYSCALL_DEFINE6(sendto, int, fd, void __user *, buff, size_t, len,
unsigned, flags, struct sockaddr __user *, addr,
int, addr_len)
SYSCALL_DEFINE3(sendmsg, int, fd, struct msghdr __user *, msg, unsigned, flags)
int sock_sendmsg(struct socket *sock, struct msghdr *msg, size_t size)
static inline int __sock_sendmsg(struct kiocb *iocb, struct socket *sock,
struct msghdr *msg, size_t size)
int **tcp_sendmsg**(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
size_t size)
void **__tcp_push_pending_frames**(struct sock *sk, unsigned int cur_mss,
int nonagle)
static int **tcp_write_xmit**(struct sock *sk, unsigned int mss_now, int nonagle,
int push_one, gfp_t gfp)
static int **tcp_transmit_skb**(struct sock *sk, struct sk_buff *skb, int clone_it,
gfp_t gfp_mask)
int ip_queue_xmit(struct sk_buff *skb)

## 2.7 recv 函数调用流程

extern ssize_t recv (int __fd, void *__buf, size_t __n, int __flags);
SYS_RECV
SYSCALL_DEFINE2(socketcall, int, call, unsigned long __user *, args)
asmlinkage long sys_recv(int fd, void __user *ubuf, size_t size,
unsigned flags)
asmlinkage long sys_recvfrom(int, void __user *, size_t, unsigned,
struct sockaddr __user *, int __user *);
int sock_recvmsg(struct socket *sock, struct msghdr *msg,
size_t size, int flags)
static inline int __sock_recvmsg(struct kiocb *iocb, struct socket *sock,
struct msghdr *msg, size_t size, int flags)
static inline int __sock_recvmsg_nosec(struct kiocb *iocb, struct socket *sock,
struct msghdr *msg, size_t size, int flags)
int **inet_recvmsg**(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
size_t size, int flags)
int **tcp_recvmsg**(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
size_t len, int nonblock, int flags, int *addr_len)
int **tcp_v4_do_rcv**(struct sock *sk, struct sk_buff *skb)

## 3. 三次握手过程

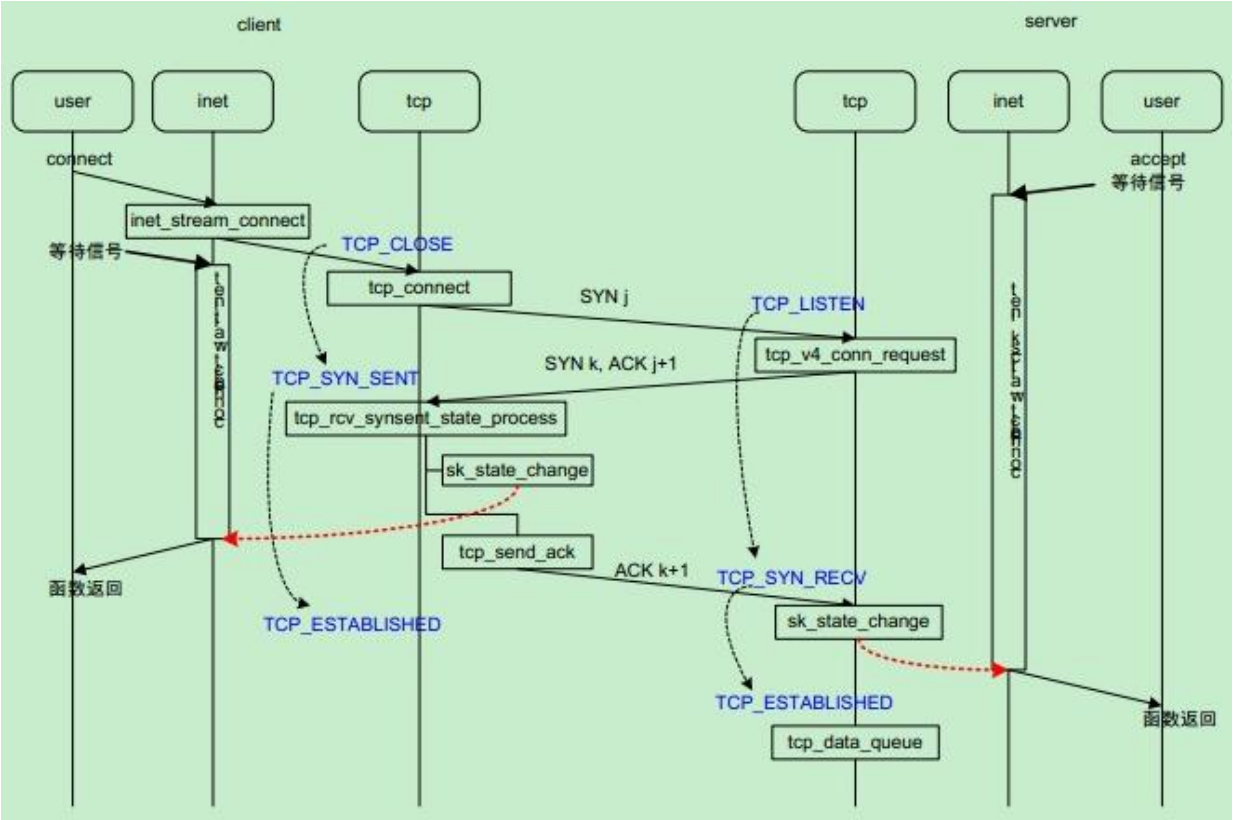



图 4 三次握手在内核中的实现序列图

**三次握手函数调用关系**

TCP 握手协议

在 TCP/IP 协议中，TCP 协议提供可靠的连接服务，采用三次握手建立一个连接。

第一次握手：建立连接时，客户端发送 syn 包(syn=j)到服务器，并进入 SYN_SEND 状态，等待服务器确认；

第二次握手：服务器收到 syn 包，必须确认客户的 SYN（ack=j+1），同时自己也发送一个 SYN 包（syn=k），即 SYN+ACK 包，此时服务器进入 SYN_RECV 状态；

第三次握手：客户端收到服务器的 SYN＋ACK 包，向服务器发送确认包 ACK(ack=k+1)，此包发送完毕，客户端和服务器进入 ESTABLISHED 状态，完成三次握手。

完成三次握手，客户端与服务器开始传送数据，在上述过程中，还有一些重要的概念：

未连接队列：在三次握手协议中，服务器维护一个未连接队列，该队列为每个客户端的 SYN 包（syn=j）开设一个条目，该条目表明服务器已收到 SYN 包， 并向客户发出确认，正在等待客户的确认包。这些条目所标识的连接在服务器处于 SYN_RECV 状态，当服务器收到客户的确认包时，删除该条目，服务器进入 ESTABLISHED 状态。

Backlog 参数：表示未连接队列的最大容纳数目。

SYN-ACK 重传次数 服务器发送完 SYN－ACK 包，如果未收到客户确认包，服

务器进行首次重传，等待一段时间仍未收到客户确认包，进行第二次重传，如果重传次数超过系统规定 的最大重传次数，系统将该连接信息从半连接队列中删除。注意，每次重传等待的时间不一定相同。

半连接存活时间：是指半连接队列的条目存活的最长时间，也即服务从收到 SYN 包到确认这个报文无效的最长时间，该时间值是所有重传请求包的最长等待时间总和。有时我们也称半连接存活时间为 Timeout 时间、SYN_RECV 存活时间。

三次握手的函数调用关系:

客户端:(发起连接请求)

```
tcp_v4_connect -> tcp_connect_init
                    -> tcp_transmit_skb -> icsk->icsk_af_ops->send_check
                                              (tcp_v4_send_check)
                                           -> icsk->icsk_af_ops->queue_xmit
                                              (ip_queue_xmit)
                                              向外发送 syn 包
                    -> inet_csk_reset_xmit_timer
                       设置从新发送的定时器
```

如果过一段时间没有接到应答:

```
tcp_retransmit_timer -> tcp_retransmit_skb -> tcp_transmit_skb
```

其余操作就跟上面的相同了.

服务器端:(接收 syn,并返回 syn/ack)

```
tcp_v4_rcv -> tcp_v4_do_rcv
                -> tcp_v4_hnd_req
                -> tcp_rcv_state_process
                     -> icsk->icsk_af_ops->conn_request
                        (tcp_v4_conn_request) ->
                             -> tcp_v4_init_sequence
                             -> tcp_v4_send_synack
                                  -> ip_build_and_send_pkt
```

客户端:(接收 syn/ack,并返回 ack)

```
tcp_v4_rcv -> tcp_v4_do_rcv
                -> tcp_rcv_state_process
                     -> tcp_rcv_synsent_state_process
                             -> tcp_ack
                             -> tcp_store_ts_recent
                             -> tcp_initialize_rcv_mss
                             -> tcp_send_ack
                                  ->tcp_transmit_skb
                     -> tcp_urg
                     -> tcp_data_snd_check
```

服务器端:(接收 ack)

```
tcp_v4_do_rcv
      -> tcp_v4_hnd_req
                -> tcp_check_req
                          ->inet_sk(sk)->icsk_af_ops->syn_recv_sock
    -> tcp_rcv_state_process
                ->    tcp_sequence
```

上面是三次握手的 tcp 协议栈部分函数调用关系的描述

# 4. TCP 状态机

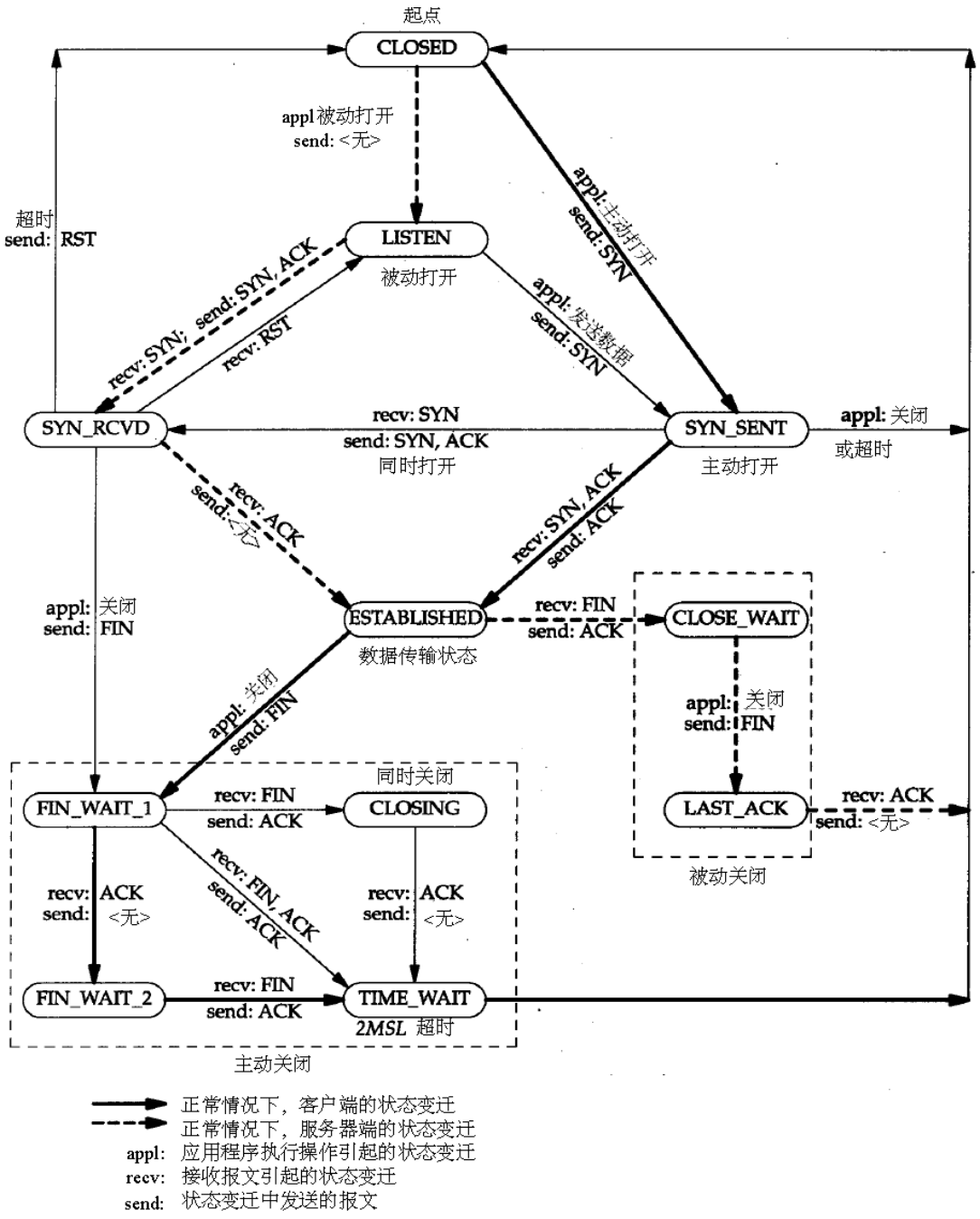



图 5 tcp 状态机全图

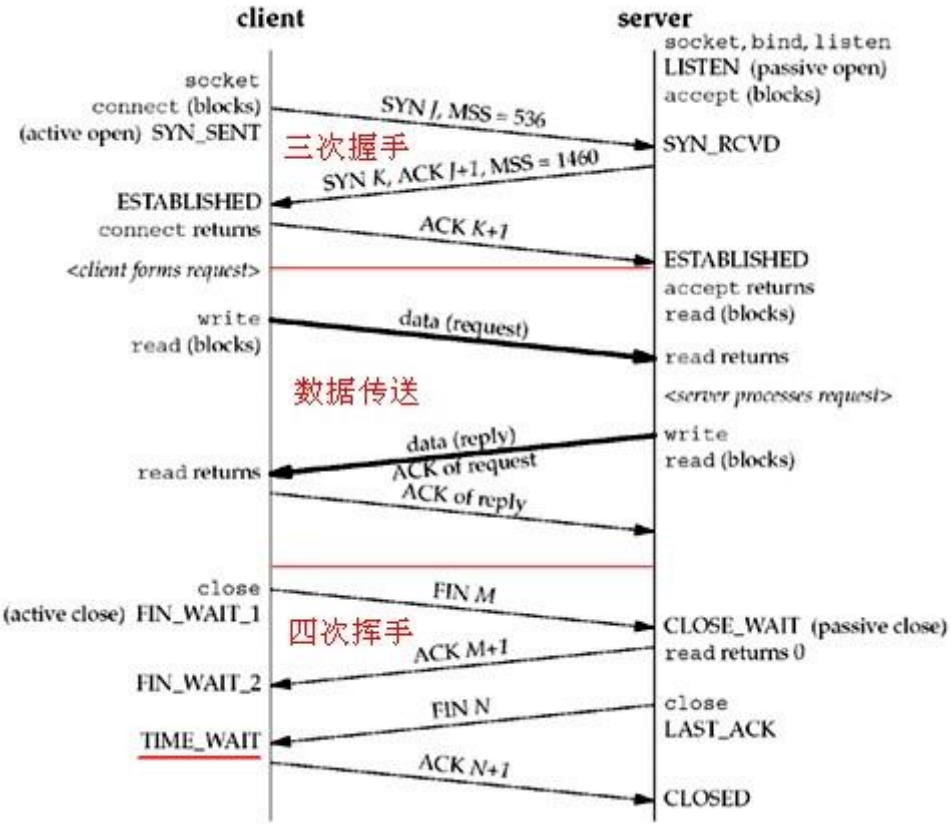



图 6 三次握手和四次握手过程中的状态转移

## 状态机中的各种状态

linux 内核中这些状态在/include/net/tcp_states.h 下声明的，该文件的源码如下：

```
00001: /*
00002: * INET An implementation of the TCP/IP protocol suite for the LINUX
00003: * operating system. INET is implemented using the BSD Socket
00004: * interface as the means of communication with the user level.
00005: *
00006: * Definitions for the TCP protocol sk_state field.
00007: *
00008: * This program is free software; you can redistribute it and/or
00009: * modify it under the terms of the GNU General Public License
00010: * as published by the Free Software Foundation; either version
00011: * 2 of the License, or (at your option) any later version.
00012: */
00013: #ifndef _LINUX_TCP_STATES_H
00014: #define _LINUX_TCP_STATES_H
00015:
00016: enum {
```

```
00017: TCP_ESTABLISHED = 1,
00018: TCP_SYN_SENT,
00019: TCP_SYN_RECV,
00020: TCP_FIN_WAIT1,
00021: TCP_FIN_WAIT2,
00022: TCP_TIME_WAIT,
00023: TCP_CLOSE,
00024: TCP_CLOSE_WAIT,
00025: TCP_LAST_ACK,
00026: TCP_LISTEN,
00027: TCP_CLOSING, /* Now a valid state */
00028:
00029: TCP_MAX_STATES /* Leave at the end! */
00030: };
00031:
00032: #define TCP_STATE_MASK 0xF
00033:
00034: #define TCP_ACTION_FIN (1 << 7)
00035:
00036: enum {
00037: TCPF_ESTABLISHED = (1 << 1),
00038: TCPF_SYN_SENT = (1 << 2),
00039: TCPF_SYN_RECV = (1 << 3),
00040: TCPF_FIN_WAIT1 = (1 << 4),
00041: TCPF_FIN_WAIT2 = (1 << 5),
00042: TCPF_TIME_WAIT = (1 << 6),
00043: TCPF_CLOSE = (1 << 7),
00044: TCPF_CLOSE_WAIT = (1 << 8),
00045: TCPF_LAST_ACK = (1 << 9),
00046: TCPF_LISTEN = (1 << 10),
00047: TCPF_CLOSING = (1 << 11)
00048: };
00049:
00050: #endif /* _LINUX_TCP_STATES_H */
```

表 1 TCP 状态描述

| 状 态 | 描 述 |
|---|---|
| CLOSED | 关闭状态，没有连接活动或正在进行 |
| LISTEN | 监听状态，服务器正在等待连接进入 |
| SYN RCVD | 收到一个连接请求，尚未确认 |
| SYN SENT | 已经发出连接请求，等待确认 |
| ESTABLISHED | 连接建立，正常数据传输状态 |
| FIN WAIT 1 | （主动关闭）已经发送关闭请求，等待确认 |
| FIN WAIT 2 | （主动关闭）收到对方关闭确认，等待对方关闭请求 |

| | |
|---|---|
| TIMED WAIT | 完成双向关闭，等待所有分组死掉 |
| CLOSING | 双方同时尝试关闭，等待对方确认 |
| CLOSE WAIT | （被动关闭）收到对方关闭请求，已经确认 |
| LAST ACK | （被动关闭）等待最后一个关闭确认，并等待所有分组死掉 |

## 状态机中的部分转移过程

tcp 的状态机在很多函数中都有体现，我们不从每个函数的角度来看，我们把 tcp 状态机全图的状态转移过程在源码中标注出来，这样比较容易理解状态转移在内核源码中如何实现。

### 三次握手过程

**客户端：**

- 初始化为 TCP_CLOSE

目录：/net/ipv4/tcp_ipv4.c
函数：tcp_v4_init_sock
实现代码：`sk->sk_state = TCP_CLOSE;`
说明：初始化为 TCP_CLOSE

- TCP_CLOSE->TCP_SYN_SENT

目录：/net/ipv4/tcp_ipv4.c
函数：tcp_v4_connect
实现代码：`tcp_set_state(sk, TCP_SYN_SENT);`
说明：发送 syn 包以后

- TCP_SYN_SENT->TCP_ESTABLISHED

目录：/net/ipv4/tcp_input.c
函数：tcp_rcv_state_process
实现代码：

```
05802:
05803: switch (sk->sk_state) {
05804: case TCP_SYN_RECV:
05805: if (acceptable) {
05806: tp->copied_seq = tp->rcv_nxt;
05807: smp_mb();
05808: tcp_set_state(sk, TCP_ESTABLISHED);
05809: sk->sk_state_change(sk);
    05810:
```

说明：在收到服务器端发来的 SYN 包以后将状态转为连接建立

**服务器端：**

TCP_CLOSE->TCP_LISTEN

目录：/net/ipv4/Inet_connection_sock.c

函数：inet_csk_listen_start

实现代码：`sk->sk_state = TCP_LISTEN;`

说明：该函数由inet_listen调用

TCP_LISTEN-> TCP_SYN_RECV

文件：/net/ipv4/tcp_input.c

函数：tcp_rcv_synsent_state_process

实现代码：

```
05666: if (th->syn) {
05667: /* We see SYN without ACK. It is attempt of
05668: * simultaneous connect with crossed SYNs.
05669: * Particularly, it can be connect to self.
05670: */
    05671: tcp_set_state(sk, TCP_SYN_RECV);
```

- TCP_SYN_RECV->TCP_ESTABLISHED

目录：/net/ipv4/tcp_input.c

函数：tcp_rcv_state_process

实现代码：`tcp_set_state(sk, TCP_ESTABLISHED);`

# 5. 附录：重要函数注释

## 5.1 tcp_v4_init_sock

/tcp/ipv4/tcp_ipv4.c

```
01929: /* NOTE: A lot of things set to zero explicitly by call to
01930: * sk_alloc() so need not be done here.
01931: */
01932: /*注意:很多在sk_alloc()函数里被明确设置为0，所以无需在此做这些事情*/
```

D:\linux-2.6.39.2\linux-2.6.39.2\net\ipv4\tcp_ipv4.c

Page 26

```
01933: static int tcp_v4_init_sock(struct sock *sk)
01934: {
01935: struct inet_connection_sock *icsk = inet_csk(sk);
01936: struct tcp_sock *tp = tcp_sk(sk);
01937:
01938: /*skb_queue_head_init，调用
01939: static inline void __skb_queue_head_init(struct sk_buff_head *list)
01940: {
```

```
01941: list->prev = list->next = (struct sk_buff *)list;
01942: list->qlen = 0;
01943: }也就是对skb队列头进行初始化*/
01944: skb_queue_head_init(&tp->out_of_order_queue);
01945: /*xmit其实就是"输出"的意思，那么tcp_init_xmit_timers显而易见初始化了很多输出计时器,
01946: 通过调用inet_csk_init_xmit_timers(sk, &tcp_write_timer, &tcp_delack_timer,&tcp_keepalive_timer);
01947: 在inet_csk_init_xmit_timers实现的过程中初始化了retransmit_timer、delack_timer、keepalive_timer
01948: 也就是对于重传、延迟的acks和探测的不同计时器*/
01949: tcp_init_xmit_timers(sk);
01950: /*当用户正在读取该套接字时，TCP包将被排入套接字的预备队列(tcp_prequeue),拷贝给用户的VJ类型的
Prequeue,结合checksumming*/
01951: tcp_prequeue_init(tp);
01952: /*初始化超时重传时间，TCP_TIMEOUT_INIT宏定义，是RFC 1122的初始化RTO值*/
01953: icsk->icsk_rto = TCP_TIMEOUT_INIT;
01954: /*mdev，即medium deviation，介质偏差*/
01955: tp->mdev = TCP_TIMEOUT_INIT;
01956:
01957: /* So many TCP implementations out there (incorrectly) count the
01958: * initial SYN frame in their delayed-ACK and congestion control
01959: * algorithms that we must have the following bandaid to talk
01960: * efficiently to them. -DaveM
01961: */
01962: /*很多TCP实现(不正确的)在它们的delayed-ACK和拥塞控制算法里计算初始的SYN帧个数，我们现在来纠正这
些错误*/
01963:
01964: /*发送方拥塞窗口值，初始化为2，RFC规定*/
01965: tp->snd_cwnd = 2;
01966:
01967: /* See draft-stevens-tcpca-spec-01 for discussion of the
01968: * initialization of these values.
01969: */
01970: /*对于这些初始值，参见draft-stevens-tcpca-spec-01*/
01971: /*拥塞窗口阈值，初始化为0x7fffffff*/
01972: tp->snd_ssthresh = TCP_INFINITE_SSTHRESH;
01973: /*snd_cwnd不要超过snd_cwnd_clamp，clamp意为"夹子"*/
01974: tp->snd_cwnd_clamp = ~0;
01975: /*Cached 有效 mss, 不包括 SACKS*/
01976: tp->mss_cache = TCP_MSS_DEFAULT;
01977:
01978: /*重排序包metric*/
01979: tp->reordering = sysctl_tcp_reordering;
01980: /*可插入的拥塞控制钩子，tcp_init_congestion_ops，其实就是一个tcp_congestion_ops结构体类型的变量
01981: 我们会详细注释*/
01982: icsk->icsk_ca_ops = &tcp_init_congestion_ops;
```

```
01983:
01984: /*这个不用说，状态*/
01985: sk->sk_state = TCP_CLOSE;
01986:
01987: /*sk_write_space是回调函数，指示有可用的bf发送空间，
01988: sk_stream_write_space就是 stream socket write_space回调函数*/
01989: sk->sk_write_space = sk_stream_write_space;
01990: /*设置sk->sk_flags，而SOCK_USE_WRITE_QUEUE就是在sock_wfree中是否调用sk->sk_write_space*/
01991: sock_set_flag(sk, SOCK_USE_WRITE_QUEUE);
01992:
01993: /*icsk_af_ops就是AF_INET{4,6}具体操作，这个挺重要的*/
01994: icsk->icsk_af_ops = &ipv4_specific;
01995: /*同步mss的函数，这里赋值为tcp_sync_mss，也就是实际上用这个函数来同步mss*/
01996: icsk->icsk_sync_mss = tcp_sync_mss;
01997:
01998: /*这里有一个小插曲，即MD5校验，tcp_sock_ipv4_specific中的函数都是为"MD5"定制的，这里我们暂时略过
*/
01999: #ifdef CONFIG_TCP_MD5SIG
02000: tp->af_specific = &tcp_sock_ipv4_specific;
02001: #endif
02002:
02003: /* TCP Cookie Transactions */
02004: /*TCP Cookie*/
02005: if (sysctl_tcp_cookie_size > 0) {
02006: /* Default, cookies without s_data_payload. */
02007: tp->cookie_values =
02008: kzalloc(sizeof(*tp->cookie_values),
02009: sk->sk_allocation);
02010: if (tp->cookie_values ! = NULL)
02011: kref_init(&tp->cookie_values- >kref);
02012: }
02013: /* Presumed zeroed, in order of appearance:
02014: * cookie_in_always, cookie_out_never,
```



```
02015: * s_data_constant, s_data_in, s_data_out
02016: */
02017: /*int，发送缓存,接收缓存*/
02018: sk->sk_sndbuf = sysctl_tcp_wmem[1];
02019: sk->sk_rcvbuf = sysctl_tcp_rmem[1];
02020:
02021: /* 禁止对本地CPU的软中断*/
02022: local_bh_disable();
02023: /*tcp_sockets_allocated就是目前的TCP socket个数*/
```

```
02024: percpu_counter_inc(&tcp_sockets_allocated);
02025: /* 恢复对本地CPU的软中断*/
02026: local_bh_enable();
02027:
02028: return 0;
02029: } ? end tcp_v4_init_sock ?
```

## 5.2 inet_bind

/tcp/ipv4/af_inet.c

```
00457: /*bind过程中要调用的inet层最主要的函数，初始化地址和端口(源和目的)
00458: 我们先来看看sockaddr和sockaddr_in
00459: struct sockaddr {
00460: sa_family_t sa_family; //address family, AF_xxx
00461: char sa_data[14]; //14比特的协议地址
00462: };
00463:
00464: //sockaddr_in
00465: struct sockaddr_in {
00466: sa_family_t sin_family; // Address family
00467: __be16 sin_port; // 端口号
00468: struct in_addr sin_addr; // 网络地址
00469:
00470: // Pad to size of `struct sockaddr'.
00471: unsigned char __pad[__SOCK_SIZE__ - sizeof(short int) -
00472: sizeof(unsigned short int) - sizeof(struct in_addr)];
00473: };
00474:
00475: */
00476: int inet_bind(struct socket *sock, struct sockaddr *uaddr, int addr_len)
00477: {
00478: struct sockaddr_in *addr = (struct sockaddr_in *)uaddr;
00479: struct sock *sk = sock->sk;
00480: struct inet_sock *inet = inet_sk(sk);
00481: unsigned short snum;
00482: int chk_addr_ret;
00483: int err;
00484:
00485: /* If the socket has its own bind function then use it. (RAW) */
00486: /*如果socket有自己的bind函数，就用自己的，当然tcp_prot里没有，raw_prot里有*/
00487: if (sk->sk_prot->bind) {
00488: err = sk->sk_prot->bind(sk, uaddr, addr_len);
00489: goto    out;
```

```
00490: }
00491: /*这个错误类型表示非法的参数*/
00492: err = - EINVAL;
00493: /*如果地址长度比sockaddr_in小，返回，错误:非法参数*/
00494: if (addr_len < sizeof(struct sockaddr_in))
00495: goto   out;
00496:
00497: /*ch_addr_ret就是地址的check结果
00498: inet_addr_type调用__inet_dev_addr_type
00499: Find address type as if only "dev" was present in the system. If
00500: on_dev is NULL then all interfaces are taken into consideration.
00501: */
00502: chk_addr_ret = inet_addr_type(sock_net(sk), addr->sin_addr.s_addr);
00503:
00504: /* Not specified by any standard per-se, however it breaks too
00505: * many applications when removed. It is unfortunate since
00506: * allowing applications to make a non-local bind solves
00507: * several problems with systems using dynamic addressing.
00508: * (ie. your servers still start up even if your ISDN link
00509: * is temporarily down)
00510: */
00511: /*未被任何标准per-se标明，但是当被移除时，会破坏很多应用。
00512: 不幸的是，因为允许应用做non-local bin，利用动态编址解决了一些系统问题
00513: (你的服务器仍然会启动，即使你的ISDN link暂时处于关闭状态)*/
00514:
00515: /*错误类型:不能分配所请求的地址*/
00516: err = - EADDRNOTAVAIL;
00517: /*如果本地绑定、非自由绑定、非透明、源地址非任意、
00518: 地址check结果不是local、multicast、broadcast，这些同时成立，
00519: 那么就返回错误:不能分配所请求的地址*/
00520: if (! sysctl_ip_nonlocal_bind &&
00521: ! (inet->freebind | | inet->transparent) &&
00522: addr->sin_addr.s_addr ! = htonl(INADDR_ANY) &&
00523: chk_addr_ret ! = RTN_LOCAL &&
00524: chk_addr_ret ! = RTN_MULTICAST &&
00525: chk_addr_ret ! = RTN_BROADCAST)
00526: goto   out;
00527:
00528: snum = ntohs(addr->sin_port);
00529: /*错误类型:Permission denied没有权限，拒绝访问*/
00530: err = - EACCES;
00531: /*主要看下PROT_SOCK，其值为1024，不能绑定到Sockets 0-1023除非你是superuser
00532: CAP_NET_BIND_SERVICE，其值为10，允许绑定到TCP/UDP sockets低于1024
00533: 允许绑定到ATM VCIs低于32 */
```

```
00534: if (snum && snum < PROT_SOCK && ! capable(CAP_NET_BIND_SERVICE))
00535: goto    out;
00536:
00537: /* We keep a pair of addresses. rcv_saddr is the one
00538: * used by hash lookups, and saddr is used for transmit.
00539: *
00540: * In the BSD API these are the same except where it
00541: * would be illegal to use them (multicast/broadcast) in
00542: * which case the sending device address is used.
00543: */
00544: /*我们保持一对地址，rcv_saddr被hash loopups使用，saddr用来发送
00545:
00546: 在BSD API中，在发送设备地址被使用的情况下，这些都是一样的，除非哪里不允许使用它们(广播/组播)
00547: */
00548: lock_sock(sk);
00549:
00550: /* Check these errors (active socket, double bind). */
00551: /*错误类型:非法*/
00552: err = - EINVAL;
00553: /*已经处于TCP_CLOSE状态，或者是本地端口，那么释放sock*/
00554: if (sk->sk_state ! = TCP_CLOSE || inet->inet_num)
00555: goto    out_release_sock;
00556:
00557: /*inet_rcv_saddr、inet_saddr、sin_addr.s_addr现在都是相同的*/
00558: inet->inet_rcv_saddr = inet->inet_saddr = addr->sin_addr.s_addr;
00559:
00560: /*地址检查结果是RTN_MULTICAST或RTN_BROADCAST，将inet_saddr初始化为0
00561: 有必要介绍一下二者:
00562: RTN_MULTICAST:Multicast route
00563: RTN_BROADCAST:Accept locally as broadcast,send as broadcast */
00564: if (chk_addr_ret == RTN_MULTICAST || chk_addr_ret == RTN_BROADCAST)
00565: inet->inet_saddr = 0; /* Use device */
00566:
00567: /* Make sure we are allowed to bind here. */
00568: /*确保我们可以在这里绑定*/
```

D:\linux-2.6.39.2\linux-2.6.39.2\net\ipv4\af_inet.c

Page 8

```
00569: /*对于tcp来说，get_port就是inet_csk_get_port，这个函数的作用是:
00570: 对于给定的sock，获得本地端口的一个索引,
00571: 如果snum是0，那么就任意分配一个端口
00572: */
00573: if (sk->sk_prot- >get_port(sk, snum)) {
00574: inet->inet_saddr = inet->inet_rcv_saddr = 0;
00575: //把错误类型置为:地址正在被使用
```

```
00576:  err = - EADDRINUSE;
00577:  goto    out_release_sock;
00578:  }
00579:  /*如果inet_rcv_saddr非0，那么锁定这个地址*/
00580:  if (inet->inet_rcv_saddr)
00581:  sk->sk_userlocks |= SOCK_BINDADDR_LOCK;
00582:  /*如果snum，也就是端口非0，那么锁定这个端口*/
00583:  if (snum)
00584:  sk->sk_userlocks |= SOCK_BINDPORT_LOCK;
00585:  /*对源端口，目的地址、目的端口进行初始化*/
00586:  inet->inet_sport = htons(inet->inet_num);
00587:  inet->inet_daddr = 0;
00588:  inet->inet_dport = 0;
00589:
00590:  sk_dst_reset(sk);
00591:  /*没有错误，err置为0*/
00592:  err = 0;
00593:  out_release_sock:
00594:  release_sock(sk);
00595:  out:
00596:  return err;
00597:  } ? end inet_bind ?
00598:  EXPORT_SYMBOL(inet_bind);__
```

## 5.3 inet_listen

/tcp/ipv4/af_inet.c

```
00189:  /*
00190:  * Move a socket into listening state.
00191:  */
00192:  /*将socket的状态变成监听*/
00193:  int inet_listen(struct socket *sock, int backlog)
00194:  {
00195:  struct sock *sk = sock->sk;
00196:  unsigned char old_state;
00197:  int err;
00198:  /*先加锁，因为要对sock进行一些修改*/
00199:  lock_sock(sk);
00200:
00201:  /*错误类型:非法*/
00202:  err = - EINVAL;
00203:  /*如果如果socket的状态是未连接，或者类型不是流类型(不是就无需监听),
00204:  那么释放sock，返回错误非法*/
```

```
00205: if (sock->state != SS_UNCONNECTED || sock->type != SOCK_STREAM)
00206: goto    out;
00207:
00208: /*保存当前状态到old_state*/
00209: old_state = sk->sk_state;
00210: if (! ((1  << old_state) & (TCPF_CLOSE | TCPF_LISTEN)))
00211: goto    out;
00212:
00213: /* Really, if the socket is already in listen state
00214: * we can only allow the backlog to be adjusted.
00215: */
00216: /*如果已经是TCP_LISTEN状态了，那么只能调整backlog*/
00217: /*如果不是TCP_LISTEN状态，调用inet_csk_listen_start
00218: 这个函数我们在它实现的地方对其进行了详细的注释，请参考*/
00219: if (old_state ! = TCP_LISTEN) {
00220: err = inet_csk_listen_start(sk, backlog);
00221: if (err)
00222: goto    out;
00223: }
00224: sk->sk_max_ack_backlog = backlog;
00225: err = 0;
00226:
00227: out:
00228: release_sock(sk);
00229: return err;
00230: } ? end inet_listen ?
00231: EXPORT_SYMBOL(inet_listen);
```

## 5.4 inet csk listen start

/tcp/ipv4/inet_connection_sock.c

```
00623: /*这个函数被af_inet.c中的inet_listen调用时，传入的参数是sk和backlog*/
00624: int inet_csk_listen_start(struct sock *sk, const int nr_table_entries)
00625: {
00626: struct inet_sock *inet = inet_sk(sk);
00627: struct inet_connection_sock *icsk = inet_csk(sk);
00628:
00629: /*icsk_accept_queue:FIFO of established children*/
00630: /*请求sk队列分配*/
00631: int rc = reqsk_queue_alloc(&icsk->icsk_accept_queue, nr_table_entries);
00632:
00633: /*如果请求到了，那么就返回*/
```

```
00634: if (rc ! = 0)
00635: return rc;
00636:
00637: /*如果请求不到，继续下面的*/
00638: /**/
00639: sk->sk_max_ack_backlog = 0;
00640: sk->sk_ack_backlog = 0;
00641:
00642: /**/
00643: inet_csk_delack_init(sk);
00644:
00645: /* There is race window here: we announce ourselves listening,
00646: * but this transition is still not validated by get_port().
00647: * It is OK, because this socket enters to hash table only
00648: * after validation is complete.
00649: */
00650: /*此处存在竞争窗口:我们声明自己在监听，但是这个转变还没有经过get_port确认。
00651: 不过没事，因为只有当确认后  socket才可以进入hash table*/
00652: /*置状态为TCP_LISTEN*/
00653: sk->sk_state = TCP_LISTEN;
00654: /*端口通过验证，对于tcp来说，可以调用inet_hash*/
00655: if (! sk->sk_prot- >get_port(sk, inet->inet_num)) {
00656: inet->inet_sport = htons(inet->inet_num);
00657:
00658: sk_dst_reset(sk);
00659: /*调用inet_hash*/
00660: sk->sk_prot->hash(sk);
00661: /*正常返回*/
00662: return 0;
00663: }
00664:
00665: /*端口没有通过验证，那么就把状态置为TCP_CLOSE*/
00666: sk->sk_state = TCP_CLOSE;
00667: /*毁掉请求的sk队列*/
00668: __reqsk_queue_destroy(&icsk->icsk_accept_queue);
00669: /*返回错误:地址正被使用*/
00670: return - EADDRINUSE;
00671: } ? end inet_csk_listen_start ?
00672: EXPORT_SYMBOL_GPL(inet_csk_listen_start);
```

## 5.5 inet_stream_connect

/tcp/ipv4/af_inet.c

```
00639: /*
00640: * Connect to a remote host. There is regrettably still a little
00641: * TCP 'magic' in here.
00642: */
00643: /*连接到远程计算机。遗憾的是不能完全和TCP分开，这里还有一些TCP的作用*/
```

D:\linux-2.6.39.2\linux-2.6.39.2\net\ipv4\af_inet.c

Page 9

```
00644: int inet_stream_connect(struct socket *sock, struct sockaddr *uaddr,
00645: int addr_len, int flags)
00646: {
00647: struct sock *sk = sock->sk;
00648: int err;
00649: long timeo;
00650: /*错误检查，这个前面讲过*/
00651: if (addr_len < sizeof(uaddr->sa_family))
00652: return - EINVAL;
00653:
00654: lock_sock(sk);
00655:
00656: /*AF_UNSPEC 地址簇未具体定义，那么免谈*/
00657: if (uaddr->sa_family == AF_UNSPEC) {
00658: err = sk->sk_prot- >disconnect(sk, flags);
00659: sock->state = err ? SS_DISCONNECTING : SS_UNCONNECTED;
00660: goto    out;
00661: }
00662:
00663: /*对各种状态分别处理*/
00664: switch (sock->state) {
00665: /*非法退出*/
00666: default:
00667: err = - EINVAL;
00668: goto    out;
00669: /*socket已经连接，返回错误"已连接"*/
00670: case SS_CONNECTED:
00671: err = - EISCONN;
00672: goto    out;
00673: /*socket正在连接，返回错误"正在连接"*/
00674: case SS_CONNECTING:
00675: err = - EALREADY;
00676: /* Fall out of switch with err, set for this state */
00677: break;
```

```
00678: /*socket未连接，我们要的就是这个*/
00679: case SS_UNCONNECTED:
00680: /*那么设置err为"已连接"，但注意不会返回这个，因为没有出错*/
00681: err = - EISCONN;
00682: /*socket状态判断了，还是要看sock状态的,如果不是TCP_CLOSE，那么退出，因为不需要connect*/
00683: if (sk->sk_state ! = TCP_CLOSE)
00684: goto    out;
00685: /*如果一切都是那么完美，走到这里才进入正题，就是调用tcp_v4_connect进行连接*/
00686: err = sk->sk_prot- >connect(sk, uaddr, addr_len);
00687:
00688: /*如果上面的tcp_v4_connect连接过程没有成功，那么退出*/
00689: if (err < 0)
00690: goto    out;
00691:
00692: /*socket状态置为SS_CONNECTING*/
00693: sock->state = SS_CONNECTING;
00694:
00695: /* Just entered SS_CONNECTING state; the only
00696: * difference is that return value in non-blocking
00697: * case is EINPROGRESS, rather than EALREADY.
00698: */
00699: /*刚刚进入SS_CONNECTING状态；唯一的不同时在non-blocking情况下返回值是EINPROGRESS,而非EALREADY*/
00700: err = - EINPROGRESS;
00701: break;
00702: } ? end switch sock->state ?
00703:
00704: timeo = sock_sndtimeo(sk, flags & O_NONBLOCK);
00705:
00706: if ((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV)) {
00707: /* Error code is set above */
00708: if (! timeo | | ! inet_wait_for_connect(sk, timeo))
00709: goto    out;
00710:
00711: err = sock_intr_errno(timeo);
00712: if (signal_pending(current))
00713: goto    out;
00714: }
00715:
00716: /* Connection was closed by RST, timeout, ICMP error
00717: * or another process disconnected us.
00718: */
00719: /*连接被RST、超时、ICMP错误报文或另一个连接断开过程关闭*/
00720: if (sk->sk_state == TCP_CLOSE)
00721: goto    sock_error;
```

```
00722:
00723: /* sk->sk_err may be not zero now, if RECVERR was ordered by user
00724: * and error was received after socket entered established state.
00725: * Hence, it is handled normally after connect() return successfully.
```



```
00726: */
00727: /*sk->sk_err现在可能不是0，如果用户要求RECVERR，而error在socket进入连接建立状态才被收到
00728: 因此，在connect成功返回后正常处理*/
00729: sock->state = SS_CONNECTED;
00730: err = 0;
00731: out:
00732: release_sock(sk);
00733: return err;
00734:
00735: sock_error:
00736: err =    sock_error(sk) ? : - ECONNABORTED;
00737: sock->state = SS_UNCONNECTED;
00738: if (sk->sk_prot- >disconnect(sk, flags))
00739: sock->state = SS_DISCONNECTING;
00740: goto    out;
00741: } ? end inet_stream_connect ?
00742: EXPORT_SYMBOL(inet_stream_connect);
```

## 5.6 tcp_v4_connect

/tcp/ipv4/tcp_ipv4.c

```
00146: /* This will initiate an outgoing connection. */
00147: /*初始化向外的连接*/
00148: int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
00149: {
00150: /*变量inet指向套接字struct sock中的inet选项*/
00151: struct inet_sock *inet = inet_sk(sk);
00152: /*变量tp指向套接字struct sock中的TCP选项*/
00153: struct tcp_sock *tp = tcp_sk(sk);
00154: /*将通用地址结构转换为IPv4的地址结构*/
00155: struct sockaddr_in *usin = (struct sockaddr_in *)uaddr;
00156: __be16 orig_sport, orig_dport;
00157:
00158: /*记录路由表项*/
00159: struct rtable *rt;
00160: /*daddr记录目的地址，nesthop记录下一条地址*/
```

```
00161: __be32 daddr, nexthop;
00162: int err;
00163:
00164: /*每次都检测这个,判断地址长度是否合法，应该要大于或者等于其地址的长度*/
00165: if (addr_len < sizeof(struct sockaddr_in))
00166: return - EINVAL;
00167: /*检查协议簇，不是AF_INET,返回不支持，为什么?因为tcp在AF_INET协议簇里*/
00168: if (usin->sin_family ! = AF_INET)
00169: return - EAFNOSUPPORT;
00170:
00171: /*下一跳和目的地址，刚开始先初始化为源地址*/
00172: nexthop = daddr = usin->sin_addr.s_addr;
00173:
00174: /*inet->opt是什么?是一个ip_options结构体变量，srr是*/
00175: if (inet->opt && inet->opt->srr) {
00176: if (! daddr)
00177: return - EINVAL;
00178: /*faddr是保存的第一跳地址*/
00179: nexthop = inet->opt->faddr;
00180: }
00181:
00182: orig_sport = inet->inet_sport;
00183: orig_dport = usin->sin_port;
00184:
00185: /*到路由这一块了,查询路由表,并通过变量rt记录下来*/
00186: rt = ip_route_connect(nexthop, inet->inet_saddr,
00187: RT_CONN_FLAGS(sk), sk->sk_bound_dev_if,
00188: IPPROTO_TCP,
00189: orig_sport, orig_dport, sk, true);
00190: if (IS_ERR(rt)) {
00191: err = PTR_ERR(rt);
00192: /*如果错误类型是:网络不可达*/
00193: if (err == - ENETUNREACH)
00194: IP_INC_STATS_BH(sock_net(sk), IPSTATS_MIB_OUTNOROUTES);
00195: return err;
00196: }
00197:
00198: /*不能是组播路由或广播路由
00199: RTCF的意思是Route cache flags*/
00200: if (rt->rt_flags & (RTCF_MULTICAST | RTCF_BROADCAST)) {
00201: ip_rt_put(rt);
00202: return - ENETUNREACH;
```

```
00203: }
00204:
00205: if (! inet->opt | | ! inet->opt->srr)
00206: daddr = rt->rt_dst;
00207:
00208: /*如果报文没有源地址(源地址为0)，就是用查询到的路由表项的源地址*/
00209: if (! inet->inet_saddr)
00210: inet->inet_saddr = rt->rt_src;
00211: inet->inet_rcv_saddr = inet->inet_saddr;
00212:
00213: if (tp->rx_opt.ts_recent_stamp && inet->inet_daddr ! = daddr) {
00214: /* Reset inherited state */
00215: /*重置继承的状态*/
00216: /*Time stamp to echo next*/
00217: tp->rx_opt.ts_recent = 0;
00218: /*Time we stored ts_recent (for aging)*/
00219: tp->rx_opt.ts_recent_stamp = 0;
00220: /*序号初始化为0*/
00221: tp- >write_seq = 0;
00222: }
00223:
00224: if (tcp_death_row.sysctl_tw_recycle &&
00225: ! tp->rx_opt.ts_recent_stamp && rt->rt_dst == daddr) {
00226: struct inet_peer *peer = rt_get_peer(rt);
00227: /*
00228: * VJ's idea. We save last timestamp seen from
00229: * the destination in peer table, when entering state
00230: * TIME-WAIT * and initialize rx_opt.ts_recent from it,
00231: * when trying new connection.
00232: */
00233: if (peer) {
00234: inet_peer_refcheck(peer);
00235: if ((u32)get_seconds() - peer->tcp_ts_stamp <= TCP_PAWS_MSL) {
00236: tp->rx_opt.ts_recent_stamp = peer->tcp_ts_stamp;
00237: tp->rx_opt.ts_recent = peer->tcp_ts;
00238: }
00239: }
00240: }
00241:
00242: /*指定目的端口和地址*/
```



```
00243: inet->inet_dport = usin->sin_port;
00244: inet->inet_daddr = daddr;
```

```
00245:
00246: /*Network protocol overhead (IP/IPv6 options)*/
00247: inet_csk(sk)->icsk_ext_hdr_len = 0;
00248: /*inet->opt->optlen就是inet的opt选项的长度*/
00249: if (inet->opt)
00250: inet_csk(sk)->icsk_ext_hdr_len = inet->opt->optlen;
00251:
00252: tp->rx_opt.mss_clamp = TCP_MSS_DEFAULT;
00253:
00254: /* Socket identity is still unknown (sport may be zero).
00255: * However we set state to SYN-SENT and not releasing socket
00256: * lock select source port, enter ourselves into the hash tables and
00257: * complete initialization after this.
00258: */
00259: /*到此socket的二元组还没有唯一确定，因为sport可能是0
00260: 但是我们设置状态为SYN-SENT，并且不会释放socket锁，选择源端口，
00261: 让我们进入hash表，完成初始化*/
00262: tcp_set_state(sk, TCP_SYN_SENT);
00263:
00264: /*tcp_death_row是一个全局变量，它有一个指向tcp_hashinfo的指针，这个函数会用到它*/
00265: err = inet_hash_connect(&tcp_death_row, sk);
00266: if (err)
00267: goto    failure;
00268:
00269: rt = ip_route_newports(rt, IPPROTO_TCP,
00270: orig_sport, orig_dport,
00271: inet->inet_sport, inet->inet_dport, sk);
00272: if (IS_ERR(rt)) {
00273: err = PTR_ERR(rt);
00274: rt = NULL;
00275: goto    failure;
00276: }
00277: /* OK, now commit destination to socket. */
00278: sk->sk_gso_type = SKB_GSO_TCPV4;
00279: sk_setup_caps(sk, &rt->dst);
00280:
00281: /*生成一个序列号，用作将来的3次握手协议*/
00282: if (! tp- >write_seq)
00283: tp- >write_seq = secure_tcp_sequence_number(inet->inet_saddr,
00284: inet->inet_daddr,
00285: inet->inet_sport,
00286: usin->sin_port);
00287:
00288: inet->inet_id = tp- >write_seq ^ jiffies;
```

```
00289:
00290: /*在tcp_connect里会发送SYN包，直到收到ack*/
00291: err = tcp_connect(sk);
00292: rt = NULL;
00293: if (err)
00294:     goto    failure;
00295:
00296: return 0;
00297:
00298: failure:
00299: /*
00300: * This unhashes the socket and releases the local port,
00301: * if necessary.
00302: */
00303: /*从hash表中删除socket，并释放本地端口，如果需要的话*/
00304: /*如果连接失败，则需把套接字状态设置为：TCP_CLOSE*/
00305: tcp_set_state(sk, TCP_CLOSE);
00306: ip_rt_put(rt);
00307: /*sk_route_caps，网络驱动特征标志*/
00308: sk->sk_route_caps = 0;
00309: inet->inet_dport = 0;
00310: return err;
00311: } ? end tcp_v4_connect ?
00312: EXPORT_SYMBOL(tcp_v4_connect);__
```

## 5.7 tcp_connect

/tcp/ipv4/tcp_output.c

```
02650: /* Build a SYN and send it off. */
02651: /*建立SYN包，并发送*/
02652: int tcp_connect(struct sock *sk)
02653: {
02654: struct tcp_sock *tp = tcp_sk(sk);
02655: struct sk_buff *buff;
02656: int err;
02657: /* Do all connect socket setups that can be done AF independent. */
02658: tcp_connect_init(sk);
02659: /*allocate a network buffer*/
```

D:\linux-2.6.39.2\linux-2.6.39.2\net\ipv4\tcp_output.c

Page 35

```
02660: buff = alloc_skb_fclone(MAX_TCP_HEADER + 15, sk->sk_allocation);
02661: /*没有分配到，返回错误"无BUFS"*/
02662: if (unlikely(buff == NULL))
02663: return - ENOBUFS;
```

```
02664:
02665: /* Reserve space for headers. */
02666: /*在buff中为头部留出空间*/
02667: skb_reserve(buff, MAX_TCP_HEADER);
02668:
02669: /*下一个要发送的包就是write_seq，即tcp发送缓存中所保持数据的Tail(+1)*/
02670: tp->snd_nxt = tp- >write_seq;
02671: /* 构建无数据skb的的常用控制比特.如果是SYN/FIN,自动增加end seqno.*/
02672: tcp_init_nondata_skb(buff, tp- >write_seq++, TCPHDR_SYN);
02673:
02674: /* Packet ECN state for a SYN. ECN是显示拥塞通告*/
02675: TCP_ECN_send_syn(sk, buff);
02676:
02677: /* Send it off. */
02678: /*发送*/
02679: /*设置控制块when为tcp_time_stamp，就是当前时间，when用来就是rtt*/
02680: TCP_SKB_CB(buff)- >when = tcp_time_stamp;
02681: /*retrans_stamp: Timestamp of the last retransmit,
02682: also used in SYN-SENT to remember stamp of
02683: the first SYN. */
02684: tp->retrans_stamp = TCP_SKB_CB(buff)- >when;
02685: /*释放头部索引*/
02686: skb_header_release(buff);
02687: /*Insert an sk_buff on a list*/
02688: __tcp_add_write_queue_tail(sk, buff);
02689:
02690: /*初始persistent queue size，持久储蓄队列大小*/
02691: sk->sk_wmem_queued += buff->truesize;
02692: sk_mem_charge(sk, buff->truesize);
02693: /*packets_out就是"in flight"的包数，加上buff中的包数*/
02694: tp->packets_out += tcp_skb_pcount(buff);
02695: /*赶紧发吧*/
02696: err = tcp_transmit_skb(sk, buff, 1, sk->sk_allocation);
02697: /*如果错误是"连接被拒绝"*/
02698: if (err == - ECONNREFUSED)
02699: return err;
02700:
02701: /* We change tp->snd_nxt after the tcp_transmit_skb() call
02702: * in order to make this packet get counted in tcpOutSegs.
02703: */
02704: tp->snd_nxt = tp- >write_seq;
02705: tp->pushed_seq = tp- >write_seq;
02706: TCP_INC_STATS(sock_net(sk), TCP_MIB_ACTIVEOPENS);
02707:
```

```
02708: /* Timer for repeating the SYN until an answer. */
02709: inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
02710: inet_csk(sk)->icsk_rto, TCP_RTO_MAX);
02711: return 0;
02712: } ? end tcp_connect ?
02713: EXPORT_SYMBOL(tcp_connect);
```

## 5.8 tcp_transmit_skb

/tcp/ipv4/tcp_output.c

```
00783: /* This routine actually transmits TCP packets queued in by
00784: * tcp_do_sendmsg(). This is used by both the initial
00785: * transmission and possible later retransmissions.
00786: * All SKB's seen here are completely headerless. It is our
00787: * job to build the TCP header, and pass the packet down to
00788: * IP so it can do the same plus pass the packet off to the
00789: * device.
00790: *
00791: * We are working here with either a clone of the original
00792: * SKB, or a fresh unique copy made by the retransmit engine.
00793: */
00794: /*该函数是真正来发送由tcp_do_sendmsg()送进队列的TCP包的。
00795: 在初始发送和重传中使用。此处所有的SKB都是无头部的。所以我们要加上TCP头部
00796: 并把包发送给IP层，那么在下一层可以同样加头部，然后传给设备
00797:
00798: 这里可以使用原始SKB的克隆，也可以是传输机制生成的原始独一无二的拷贝
00799: */
00800: static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int clone_it,
00801: gfp_t gfp_mask)
00802: {
00803: const struct inet_connection_sock *icsk = inet_csk(sk);
00804: struct inet_sock *inet;
00805: struct tcp_sock *tp;
00806: struct tcp_skb_cb *tcb;
00807: struct tcp_out_options opts;
00808: unsigned tcp_options_size, tcp_header_size;
00809: struct tcp_md5sig_key *md5;
00810: struct tcphdr *th;
00811: int err;
00812:
00813: BUG_ON(!skb || !tcp_skb_pcount(skb));
00814:
00815: /* If congestion control is doing timestamping, we must
```

```
00816: * take such a timestamp before we potentially clone/copy.
00817: */
00818: /*如果拥塞控制正在timpstamping，我们必须在克隆/复制前取一个时间戳*/
00819: if (icsk->icsk_ca_ops- >flags & TCP_CONG_RTT_STAMP)
00820: __net_timestamp(skb);
00821:
00822: if (likely(clone_it)) {
00823: if (unlikely(skb_cloned(skb)))
00824: skb = pskb_copy(skb, gfp_mask);
00825: else
00826: skb = skb_clone(skb, gfp_mask);
00827: if (unlikely(! skb))
00828: return - ENOBUFS;
00829: }
00830:
00831: /*得到三个东西:inet_sock、tcp_sock、tcp_skb_cb*/
00832: inet = inet_sk(sk);
00833: tp = tcp_sk(sk);
00834: tcb = TCP_SKB_CB(skb);
00835:
00836: /*设置存储空间*/
00837: memset(&opts, 0, sizeof(opts));
00838:
00839: /*如果是为SYN包建头部，则调用tcp_syn_options*/
00840: if (unlikely(tcb->flags & TCPHDR_SYN))
00841: tcp_options_size = tcp_syn_options(sk, skb, &opts, &md5);
00842: else
00843: /*为连接建立以后的包构建头部，则调用tcp_established_options*/
00844: tcp_options_size = tcp_established_options(sk, skb, &opts,
00845: &md5);
00846: /*tcp头部总长度*/
00847: tcp_header_size = tcp_options_size + sizeof(struct tcphdr);
00848: /*CA_EVENT_TX_START:第一次发送 当没有包in flight*/
00849: if (tcp_packets_in_flight(tp) == 0) {
00850: tcp_ca_event(sk, CA_EVENT_TX_START);
00851: skb->ooo_okay = 1;
00852: } else
00853: skb->ooo_okay = 0;
00854:
00855: /*add data to the start of a buffer*/
00856: /*向buffer的开头添加数据*/
00857: skb_push(skb, tcp_header_size);
00858: /*添加完数据后重置传输层的头部*/
00859: skb_reset_transport_header(skb);
```

```
00860: /**/
00861: skb_set_owner_w(skb, sk);
00862:
00863: /* Build TCP header and checksum it. */
00864: /*构建TCP头部并checksum*/
00865: /*先取出来*/
00866: th = tcp_hdr(skb);
00867: /*给源端口、目的端口、序列号、ack序列号赋值*/
00868: th->source = inet->inet_sport;
00869: th->dest = inet->inet_dport;
00870: th->seq = htonl(tcb->seq);
00871: th->ack_seq = htonl(tp->rcv_nxt);
00872: /*用无符号短整型指针来替代th时,+6代表指针移动6次，每次16位，按照hdr的结构，刚好到flags开始的地方
00873: 所以这句话是在给flags赋值，*/
00874: *(((__be16 *)th) + 6) = htons(((tcp_header_size >> 2) << 12) |
00875: tcb->flags);
00876: /*下面是对头部中的窗口值进行赋值*/
00877: if (unlikely(tcb->flags & TCPHDR_SYN)) {
00878: /* RFC1323: The window in SYN & SYN/ACK segments
00879: * is never scaled.
00880: */
00881: /*对于SYN包来说，因为是刚开始，所以窗口值设置为接收窗口和65535的最小值*/
00882: th->window = htons(min(tp->rcv_wnd, 65535U));
00883: } else {
00884: /*对于其他包来说，用更通用的选择窗口函数来选择一个窗口*/
00885: th->window = htons(tcp_select_window(sk));
00886: }
00887: /*校验和，初始化为0*/
00888: th->check = 0;
00889: /*紧急指针，初始化为0*/
00890: th->urg_ptr = 0;
00891:
00892: /* The urg_mode check is necessary during a below snd_una win probe */
00893: /*在下面的snd_una窗口探查时，检查urg_mode是必要的*/
00894: /*tcp_urg_mode函数返回snd_una!=snd_up,即我们需要ack的第一个byte!=紧急指针，一般情况下二者相等*/
00895: if (unlikely(tcp_urg_mode(tp) && before(tcb->seq, tp->snd_up))) {
00896: if (before(tp->snd_up, tcb->seq + 0x10000)) {
00897: th->urg_ptr = htons(tp->snd_up - tcb->seq);
00898: th->urg = 1;
00899: } else if (after(tcb->seq + 0xFFFF, tp->snd_nxt)) {
00900: th->urg_ptr = htons(0xFFFF);
00901: th->urg = 1;
```

```
00902: }
00903: }
00904:
00905: /*将先前计算的TCP options写入包*/
00906: tcp_options_write((__be32 *)(th + 1), tp, &opts);
00907: if (likely((tcb->flags & TCPHDR_SYN) == 0))
00908: TCP_ECN_send(sk, skb, tcp_header_size);
00909:
00910: #ifdef CONFIG_TCP_MD5SIG
00911: /* Calculate the MD5 hash, as we have all we need now */
00912: /*计算MD5哈希，因为已经得到了所有*/
00913: if (md5) {
00914: sk_nocaps_add(sk, NETIF_F_GSO_MASK);
00915: tp->af_specific- >calc_md5_hash(opts.hash_location,
00916: md5, sk, NULL, skb);
00917: }
00918: #endif
00919:
00920: /*调用tcp_v4_send_check计算校验和*/
00921: icsk->icsk_af_ops- >send_check(sk, skb);
00922:
00923: /*如果发送ACK包，计算我们发送的ACK包个数*/
00924: if (likely(tcb->flags & TCPHDR_ACK))
00925: tcp_event_ack_sent(sk, tcp_skb_pcount(skb));
00926: /*如果发送数据包，计算拥塞状态*/
00927: if (skb->len != tcp_header_size)
00928: tcp_event_data_sent(tp, skb, sk);
00929:
00930: if (after(tcb->end_seq, tp->snd_nxt) || tcb->seq == tcb->end_seq)
00931: TCP_ADD_STATS(sock_net(sk), TCP_MIB_OUTSEGS,
00932: tcp_skb_pcount(skb));
00933:
00934: /*调用 ip_queue_xmit 把 skb 交给 ip 层*/
```

## 5.9 tcp_sendmsg

/tcp/ipv4/tcp.c

```
00916: int tcp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
00917: size_t size)
00918: {
00919: struct iovec *iov;
00920: struct tcp_sock *tp = tcp_sk(sk);
00921: struct sk_buff *skb;
```

```
00922: int iovlen, flags;
00923: int mss_now, size_goal;
00924: int sg, err, copied;
00925: long timeo;
00926: /*上锁*/
00927: lock_sock(sk);
00928: /*取出flag，主要是看是非阻塞还是阻塞模式*/
00929: flags = msg->msg_flags;
00930:
00931: /*设置发送超时时间，这个时间是跟发送缓存有关系的*/
00932: /*sock_sndtimeo执行的结果是，如果flags里有MSG_DONTWAIT,那么timeo=0，否则timeo=sk->sk_sndtimeo*/
00933: timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);
00934:
00935: /* Wait for a connection to finish. */
00936: /*等待连接建立，如果是非阻塞则直接返回*/
00937: /*可能有人不理解阻塞和非阻塞是什么意思:简单来说，阻塞就是干不完不回来，
00938: 非阻塞就是我先去忙其他的了，干完了告诉我一声*/
00939: /*还有很多人会疑惑TCPF_ESTABLISHED和TCP_ESTABLISHED，有什么不一样，主要的区别在于TCPF_*是按位的，
可以组合*/
00940:
00941: /*如果sock的状态不是连接建立和关闭等待*/
00942: if ((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT))
00943: /*等待sock到达连接状态，等待的时间是timeo*/
00944: if ((err = sk_stream_wait_connect(sk, &timeo)) != 0)
00945: goto    out_err;
00946:
00947: /* This should be in poll */
00948: /*清除对应比特*/
00949: clear_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);
00950:
00951: /*取tcp的发送mss，在tcp_current_mss还会设置size_goal，通过tcp_xmit_size_goal,
00952: 这个值一般都是等于mss,除非有gso的情况下,有所不同.
00953: 这里我们就认为他是和mms相等的*/
00954: mss_now = tcp_send_mss(sk, &size_goal, flags);
00955:
00956: /* Ok commence sending. */
00957: /*------------------开始发送------------------------*/
00958: /*块个数、数据块*/
00959: /*iov的意思是iovec*/
00960: iovlen = msg->msg_iovlen;
00961: iov = msg->msg_iov;
00962: /*统计拷贝的比特数，还没拷贝呢，所以为0*/
00963: copied = 0;
00964:
```

```
00965: /*EPIPE:broken pipe*/
```

```
00966: err = - EPIPE;
00967: /*下面的sk->sk_shutdown是SEND_SHUTDOWN的掩码*/
00968: /*如果发送端已经完全关闭或者sock有错误，则返回，并设置errno*/
00969: if (sk->sk_err | | (sk->sk_shutdown & SEND_SHUTDOWN))
00970: goto    out_err;
00971: /*sk_route_caps:route capabilities 路由功能，在后面select_size函数调用时sg作为第二个参数*/
00972: sg = sk->sk_route_caps & NETIF_F_SG;
00973:
00974: /*---------对iovlen进行循环---------*/
00975: while (- - iovlen >= 0) {
00976: /*取出当前buf的长度*/
00977: size_t seglen = iov->iov_len;
00978: /* iov_base:BSD uses caddr_t (1003.1g requires void *) */
00979: /*取出buf的基地址*/
00980: unsigned char __user *from = iov->iov_base;
00981:
00982: iov++;
00983:
00984: /*----------循环直到seglen小于等于0---------*/
00985: while (seglen > 0) {
00986: int copy = 0;
00987: int max = size_goal;
00988: /*我们知道sock的发送队列sk_write_queue是一个双向链表,
00989: 而用tcp_write_queue_tail则是取得链表的最后一个元素.
00990: (如果链表为空则返回NULL).*/
00991: skb = tcp_write_queue_tail(sk);
00992: /*front of stuff to transmit*/
00993: /*看当前buf空间不够时的情况,它这里判断buf空间是否足够是通过*/
00994: if (tcp_send_head(sk)) {
00995: if (skb->ip_summed == CHECKSUM_NONE)
00996: max = mss_now;
00997: copy = max - skb->len;
00998: }
00999: /*来判断的,这里稍微解释下这个:
01000: 这里tcp_send_head返回值为sk->sk_send_head,也就是指向当前的将要发送的buf的front位置.
01001: 如果为空,则说明buf没有空间,我们就需要alloc一个段来保存将要发送的msg.
01002: 而skb->len指的是当前的skb的所包含的数据的大小(包含头的大小).而这个值如果大于size_goal,
01003: 则说明buf已满,我们需要重新alloc一个段.如果小于size_goal,
01004: 则说明buf还有空间来容纳一些数据来组成一个等于mss的数据包再发送给ip层.*/
01005:
01006: /*----如果max比skb->len小，那么肯定要分配新段----*/
```

```
01007: if (copy <= 0) {
01008: new_segment:
01009: /* Allocate new segment. If the interface is SG,
01010: * allocate skb fitting to single page.
01011: */
01012: /*分配新段，如果接口是SG，分配适应单页的skb*/
01013:
01014: /*如果没有空间了，没有就去排队*/
01015: if (! sk_stream_memory_free(sk))
01016: goto    wait_for_sndbuf;
01017:
01018: /*分配新的skb*/
01019: /*alloc的大小一般都是等于mss的大小,这里是通过select_size得到的*/
01020: /*三个参数:sk、分配大小、分配模式*/
01021: skb = sk_stream_alloc_skb(sk,
01022: select_size(sk, sg),
01023: sk->sk_allocation);
01024: /*没有分到就去排队*/
01025: if (! skb)
01026: goto    wait_for_memory;
01027:
01028: /*
01029: * Check whether we can use HW checksum.
01030: */
01031: if (sk->sk_route_caps & NETIF_F_ALL_CSUM)
01032: skb->ip_summed = CHECKSUM_PARTIAL;
01033: /*将这个skb加入到sk_write_queue队列中,并更新sk_send_head域(即当前发送buf)*/
01034: skb_entail(sk, skb);
01035: /*更新copy值*/
01036: copy = size_goal;
01037: max = size_goal;
01038: } ? end if copy<=0 ?
01039:
01040: /* Try to append data to the end of skb. */
01041: /*将数据追加到skb后面*/
01042: /*如果copy大于buf的大小，则缩小copy,copy就是当前所拷贝数据大小*/
01043: if (copy > seglen)
01044: copy = seglen;
01045:
01046: /* Where to copy to? */
01047: /*这里要查看skb的空间，如果大于0，说明是新建的skb，因为如果不是新建的，
```



```
01048: 肯定要把它填满，能到这一步的，如果还有空间，肯定是新建的*/
```

```
01049: if (skb_tailroom(skb) > 0) {
01050: /* We have some space in skb head. Superb! */
01051: /*还有点空间，不错*/
01052: /*如果要复制的数据大于所剩余的空间，那么先复制skb所能容纳的大小*/
01053: if (copy > skb_tailroom(skb))
01054: copy = skb_tailroom(skb);
01055: /*复制数据到sk_buff，大小为copy，如果成功，就进入do_fault*/
01056: if ((err = skb_add_data(skb, from, copy)) ! = 0)
01057: goto    do_fault;
01058: } else {
01059: int merge = 0;
01060: /*取得nr_frags，也就是保存物理页的数组*/
01061: int i = skb_shinfo(skb)->nr_frags;
01062: /*从sk中取得当前发送物理页cached page for sendmsg*/
01063: struct page *page = TCP_PAGE(sk);
01064: /*取得当前页偏移*/
01065: int off = TCP_OFF(sk);
01066: /*这里主要是判断skb的发送页是否已经存在于nr_frags中,
01067: 如果存在并且也没有满,则我们只需要将数据合并到这个页就可以了,
01068: 而不需要在frag再添加一个页*/
01069: if (skb_can_coalesce(skb, i, page, off) &&
01070: off ! = PAGE_SIZE) {
01071: /* We can extend the last page
01072: * fragment. */
01073: merge = 1;
01074: } else if (i == MAX_SKB_FRAGS | | ! sg) {
01075: /* Need to add new fragment and cannot
01076: * do this because interface is non-SG,
01077: * or because all the page slots are
01078: * busy. */
01079: /*到这里说明要么设备不支持SG IO,要么页已经满了.
01080: 因为我们知道nr_frags的大小是有限制的.
01081: 此时调用tcp_mark_push来加一个PSH标记*/
01082: tcp_mark_push(tp, skb);
01083: goto    new_segment;
01084: } else if (page) {
01085: if (off == PAGE_SIZE) {
01086: /*这里说明当前的发送页已满*/
01087: put_page(page);
01088: TCP_PAGE(sk) = page = NULL;
01089: off = 0;
01090: }
01091: } else
01092: off = 0;
```

```
01093:
01094: /*如果copy大于当前页所剩余的空间，那么缩小copy*/
01095: if (copy > PAGE_SIZE - off)
01096: copy = PAGE_SIZE - off;
01097:
01098: if (! sk_wmem_schedule(sk, copy))
01099: goto    wait_for_memory;
01100: /*如果page为NULL则需要新alloc一个物理页*/
01101: if (! page) {
01102: /* Allocate new cache page. */
01103: if (! (page = sk_stream_alloc_page(sk)))
01104: goto    wait_for_memory;
01105: }
01106:
01107: /* Time to copy data. We are close to
01108: * the end! */
01109: /*开始复制数据到这个物理页*/
01110: err = skb_copy_to_page(sk, from, skb, page,
01111: off, copy);
01112: /*出错的情况*/
01113: if (err) {
01114: /* If this page was new, give it to the
01115: * socket so it does not get leaked.
01116: */
01117: if (! TCP_PAGE(sk)) {
01118: TCP_PAGE(sk) = page;
01119: TCP_OFF(sk) = 0;
01120: }
01121: goto    do_error;
01122: }
01123:
01124: /* Update the skb. */
01125: /*判断是否是新建的物理页*/
01126: if (merge) {
01127: /*如果只是在存在的物理页上添加数据，则只需要更新size*/
```



```
01128: skb_shinfo(skb)->frags[i - 1].size +=
01129: copy;
01130: } else {
01131: /*下面这个函数负责添加此物理页到skb的frags*/
01132: skb_fill_page_desc(skb, i, page, off, copy);
01133: if (TCP_PAGE(sk)) {
01134: /*设置物理页的引用计数*/
```

```
01135: get_page(page);
01136: } else if (off + copy < PAGE_SIZE) {
01137: get_page(page);
01138: TCP_PAGE(sk) = page;
01139: }
01140: }
01141: /*设置物理页的位移*/
01142: TCP_OFF(sk) = off + copy;
01143: } ? end else ?
01144:
01145: /*数据复制完毕，那么就开始发送数据*/
01146:
01147: /*几个tcp_push、tcp_one_push最终都要调用__tcp_push_pending_frames
01148: ,而在它中间最终会调用tcp_write_xmit,而tcp_write_xmit则会调用tcp_transmit_skb,
01149: 这个函数最终会调用ip_queue_xmit来讲数据发送给ip层.
01150: 这里要注意,我们这里的分析忽略掉了,tcp的一些管理以及信息交互的过程.*/
01151:
01152: /*如果第一次组完一个段，则设置PSH*/
01153: if (! copied)
01154: TCP_SKB_CB(skb)->flags &= ~TCPHDR_PSH;
01155: /*然后设置写队列长度 Tail(+1) of data held in tcp send buffer*/
01156: tp- >write_seq += copy;
01157: TCP_SKB_CB(skb)->end_seq += copy;
01158: skb_shinfo(skb)->gso_segs = 0;
01159:
01160: /*更新buf基地址以及复制的buf大小*/
01161: from += copy;
01162: copied += copy;
01163: /*buf已经复制完则退出循环，并发送这个段*/
01164: if ((seglen -= copy) == 0 && iovlen == 0)
01165: goto    out;
01166: /*如果skb的数据大小小于所需拷贝的数据大小或者存在带外数据，我们继续循环,
01167: 而当存在带外数据时,我们接近着的循环会退出循环,然后调用tcp_push将数据发出.*/
01168: if (skb->len < max || (flags & MSG_OOB))
01169: continue;
01170:
01171: /*forced_push用来判断我们是否已经写了多于一半窗口大小的数据到对端.
01172: 如果是,我们则要发送一个推数据(PSH)*/
01173: if (forced_push(tp)) {
01174: tcp_mark_push(tp, skb);
01175: /*调用__tcp_push_pending_frames将开启NAGLE算法的缓存的段全部发送出去.*/
01176: __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
01177: } else if (skb == tcp_send_head(sk))
01178: /*如果当前将要发送的buf刚好为skb,则会发送当前的buf*/
```

```
01179: tcp_push_one(sk, mss_now);
01180: continue;
01181:
01182: wait_for_sndbuf:
01183: set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
01184: wait_for_memory:
01185: if (copied)
01186: /*内存不够,则尽量将本地的NAGLE算法所缓存的数据发送出去.*/
01187: tcp_push(sk, flags & ~MSG_MORE, mss_now, TCP_NAGLE_PUSH);
01188:
01189: if ((err = sk_stream_wait_memory(sk, &timeo)) ! = 0)
01190: goto    do_error;
01191: /*更新相关域*/
01192: mss_now = tcp_send_mss(sk, &size_goal, flags);
01193: } ? end while seglen>0 ?
01194: } ? end while --iovlen>=0 ?
01195:
01196: /*这里是成功返回所做的*/
01197: out:
01198: if (copied)
01199: /*这里可以看到最终的flag是tp->nonagle,而这个就是看套接口选项是否有开nagle算法,
01200: 如果没开的话,立即把数据发出去,否则则会村讯nagle算法,将小数据缓存起来.*/
01201: tcp_push(sk, flags, mss_now, tp->nonagle);
01202: release_sock(sk);
01203: return copied;
01204:
01205: do_fault:
01206: if (! skb->len) {
01207: /*从write队列unlink掉当前的buf.*/
```



```
01208: tcp_unlink_write_queue(skb, sk);
01209: /* It is the one place in all of TCP, except connection
01210: * reset, where we can be unlinking the send_head.
01211: */
01212: /*更新send_head*/
01213: tcp_check_send_head(sk, skb);
01214: /*释放skb*/
01215: sk_wmem_free_skb(sk, skb);
01216: }
01217:
01218: do_error:
01219: if (copied)
01220: /*如果copied不为0,则说明发送成功一部分数据,因此此时返回out*/
```

```
01221: goto   out;
01222: out_err:
01223: /*否则进入错误处理*/
01224: err = sk_stream_error(sk, flags, err);
01225: release_sock(sk);
01226: return err;
01227: } ? end tcp_sendmsg ?
01228: EXPORT_SYMBOL(tcp_sendmsg);__
```

## 5.10__tcp_push_pending_frames

/tcp/ipv4/tcp_output.c

```
01852: /* Push out any pending frames which were held back due to
01853: * TCP_CORK or attempt at coalescing tiny packets.
01854: * The socket must be locked by the caller.
01855: */
01856: /*推出所有的挂起帧，这些帧受阻，因为TCP_CORK或者尝试聚合小的包，调用者必须锁定socket*/
01857: void __tcp_push_pending_frames(struct sock *sk, unsigned int cur_mss,
01858: int nonagle)
01859: {
01860: /* If we are closed, the bytes will have to remain here.
01861: * In time closedown will finish, we empty the write queue and
01862: * all will be happy.
01863: */
01864: /*如果关闭了，数据留在这里，最后关闭会完成，我们清空写入队列*/
01865: /*如果关闭状态，返回*/
01866: if (unlikely(sk->sk_state == TCP_CLOSE))
01867: return;
01868: /*否则调用tcp_write_xmit来发送*/
01869: if (tcp_write_xmit(sk, cur_mss, nonagle, 0, GFP_ATOMIC))
01870: tcp_check_probe_timer(sk);
01871: }
```

## 5.11tcp_write_xmit

/tcp/ipv4/tcp_output.c

```
01757: /* This routine writes packets to the network. It advances the
01758: * send_head. This happens as incoming acks open up the remote
01759: * window for us.
01760: *
01761: * LARGESEND note: !tcp_urg_mode is overkill, only frames between
```

```
01762: * snd_up-64k-mss .. snd_up cannot be large. However, taking into
01763: * account rare use of URG, this is not a big flaw.
01764: *
01765: * Returns 1, if no segments are in flight and we have queued segments, but
01766: * cannot send anything now because of SWS or another problem.
01767: */
01768: /*这个函数实现将包写入网络的功能，它使send_head前进，发生在收到的ack打开了远程窗口
01769:
01770: LARGESEND 注意:
01771: */
01772: static int tcp_write_xmit(struct sock *sk, unsigned int mss_now, int nonagle,
01773: int push_one, gfp_t gfp)
01774: {
01775: struct tcp_sock *tp = tcp_sk(sk);
01776: struct sk_buff *skb;
01777: unsigned int tso_segs, sent_pkts;
01778: int cwnd_quota;
01779: int result;
01780:
01781: sent_pkts = 0;
01782:
01783: if (! push_one) {
01784: /* Do MTU probing. */
01785: /*探测mtu*/
01786: result = tcp_mtu_probe(sk);
01787: if (! result) {
01788: return 0;
01789: } else if (result > 0) {
01790: sent_pkts = 1;
01791: }
01792: }
01793: /*开始处理数据包*/
01794: while ((skb = tcp_send_head(sk))) {
01795: unsigned int limit;
01796:
01797: tso_segs = tcp_init_tso_segs(sk, skb, mss_now);
01798: BUG_ON(! tso_segs);
01799: /*主要用来测试拥塞窗口*/
01800: cwnd_quota = tcp_cwnd_test(tp, skb);
01801: if (! cwnd_quota)
01802: break;
01803:
01804: if (unlikely(! tcp_snd_wnd_test(tp, skb, mss_now)))
01805: break;
```

```
01806: /*主要看这里,如果这个skb是写队列的最后一个buf,则传输TCP_NAGLE_PUSH给tcp_nagle_test,
01807: 这个时侯直接返回1,于是接着往下面走,否则则说明数据包不要求理解发送,
01808: 我们就跳出循环(这时数据段就不会被发送).比如设置了TCP_CORK*/
```



```
01809: if (tso_segs == 1) {
01810: if (unlikely(! tcp_nagle_test(tp, skb, mss_now,
01811: (tcp_skb_is_last(sk, skb) ?
01812: nonagle : TCP_NAGLE_PUSH))))
01813: break;
01814: } else {
01815: if (! push_one && tcp_tso_should_defer(sk, skb))
01816: break;
01817: }
01818:
01819: limit = mss_now;
01820: if (tso_segs > 1 && ! tcp_urg_mode(tp))
01821: limit = tcp_mss_split_point(sk, skb, mss_now,
01822: cwnd_quota);
01823:
01824: if (skb->len > limit &&
01825: unlikely(tso_fragment(sk, skb, limit, mss_now, gfp)))
01826: break;
01827:
01828: TCP_SKB_CB(skb)- >when = tcp_time_stamp;
01829: /*传输数据给3层*/
01830: if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp)))
01831: break;
01832:
01833: /* Advance the send_head. This one is sent out.
01834: * This call will increment packets_out.
01835: */
01836: tcp_event_new_data_sent(sk, skb);
01837:
01838: tcp_minshall_update(tp, mss_now, skb);
01839: sent_pkts++;
01840:
01841: if (push_one)
01842: break;
01843: } ? end while (skb=tcp_send_head(sk)) ?
01844:
01845: if (likely(sent_pkts)) {
```

```
01846: tcp_cwnd_validate(sk);
01847: return 0;
01848: }
01849: return ! tp->packets_out && tcp_send_head(sk);
01850: } ? end tcp_write_xmit ?
```

## 5.12inet_recvmsg

/tcp/ipv4/af_inet.c
选择一项。

## 5.13tcp_recvmsg

/tcp/ipv4/tcp.c

```
01490: /*
01491: * This routine copies from a sock struct into the user buffer.
01492: *
01493: * Technical note: in 2.3 we work on _locked_ socket, so that
01494: * tricks with *seq access order and skb->users are not required.
01495: * Probably, code can be easily improved even more.
01496: */
01497: /*这个函数拷贝sock结构到用户缓存
01498: 从网络协议栈接收数据的动作,自上而下的触发动作一直到这个函数为止,出现了一次等待的过程.
01499: 函数tcp_recvmsg可能会被动地等待在sk的接收数据队列上,也就是说,系统中肯定有其他地方会去
01500: 修改这个队列使得tcp_recvmsg可以进行下去.入口参数sk是这个网络连接对应的sock{}指针,msg用
01501: 于存放接收到的数据.接收数据的时候会去遍历接收队列中的数据,找到序列号合适的.
01502: 但读取队列为空时tcp_recvmsg就会调用tcp_v4_do_rcv使用backlog队列填充接收队列
01503: */
01504: int tcp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
01505: size_t len, int nonblock, int flags, int *addr_len)
01506: {
01507: struct tcp_sock *tp = tcp_sk(sk);
01508: int copied = 0;
01509: u32 peek_seq;
01510: u32 *seq;
01511: unsigned long used;
01512: int err;
01513: int target; /* Read at least this many bytes */
01514: long timeo;
01515: struct task_struct *user_recv = NULL;
```

```
01516: int copied_early = 0;
01517: struct sk_buff *skb;
01518: u32 urg_hole = 0;
01519:
01520: /*上锁*/
01521: lock_sock(sk);
01522:
01523: /*错误类型:未连接*/
```

```
01524: err = - ENOTCONN;
01525: /*sock状态是监听，直接out*/
01526: if (sk->sk_state == TCP_LISTEN)
01527: goto    out;
01528:
01529: /*设置接收超时时间，如果设置了非阻塞，那么超时时间=0；
01530: 如果没有设置非阻塞，那么超时时间=sk->sk_rcvtimeo*/
01531: timeo = sock_rcvtimeo(sk, nonblock);
01532:
01533: /* Urgent data needs to be handled specially. */
01534: /*如果要接收带外数据，那么要特殊处理，转到recv_urg*/
01535: if (flags & MSG_OOB)
01536: goto    recv_urg;
01537:
01538: /*copied_seq:未读数据头*/
01539: /*设置现在要读的序号指针*/
01540: seq = &tp->copied_seq;
01541: /*如果设置了MSG_PEEK，那么从缓冲区复制完数据以后，不会将缓冲区中对应的数据删除*/
01542: if (flags & MSG_PEEK) {
01543: /*peek_seq初始化为还未读数据头，而seq要读序号指针指向peek_seq，因为在设置了MSG_PEEK后
01544: 就是从peek_seq开始读*/
01545: peek_seq = tp->copied_seq;
01546: seq = &peek_seq;
01547: }
01548:
01549: /*MSG_WAITALL:Wait for a full request*/
01550: /*sock_rcvlowat的作用是等待数据缓冲区数据长度达到
01551: 传递数据之前缓冲区内的最小字节数. 在 Linux 中这两个值是不可改变的, 固定为 1 字节.*/
01552: /*在这个函数里
01553: 如果设置了MSG_WAITALL，那么返回值为len
01554: 如果没有设置，那么返回值为
01555: sk->sk_rcvlowat<len那么返回0，也就是说现在缓冲区的数据长度已经满足要求了，target=0
01556: sk->sk_rcvlowat>len那么返回1，就是说缓冲区的数据长度还没达到要求，target=1(最少为1字节)*/
01557: target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);
```

```
01558:
01559: #ifdef CONFIG_NET_DMA
01560: /*dma_chan:异步拷贝的成员*/
01561: tp->ucopy.dma_chan = NULL;
01562: /*主要执行inc_preempt_count()(增加PREEMPT,从而禁止内核态抢占), preempt的意思就是"抢占"*/
01563: preempt_disable();
01564: /*skb_peek_tail:peek at the tail of an &sk_buff_head*/
01565: /*把接收队列中的最后一个skb取出来*/
01566: skb = skb_peek_tail(&sk->sk_receive_queue);
01567: {
01568: int available = 0;
01569:
01570: if (skb)
01571: available = TCP_SKB_CB(skb)->seq + skb->len - (*seq);
01572: if ((available < target) &&
01573: (len > sysctl_tcp_dma_copybreak) && ! (flags & MSG_PEEK) &&
01574: ! sysctl_tcp_low_latency &&
01575: dma_find_channel(DMA_MEMCPY)) {
01576: preempt_enable_no_resched();
01577: tp->ucopy.pinned_list =
01578: dma_pin_iovec_pages(msg->msg_iov, len);
01579: } else {
01580: preempt_enable_no_resched();
01581: }
01582: }
01583: #endif
01584:
01585: do {
01586: u32 offset;
01587:
01588: /* Are we at urgent data? Stop if we have read anything or have SIGURG pending. */
01589: /*我们在处理紧急数据吗?如果读到了数据或者有SIGURG挂起数据*/
01590: /*urg_data:保存的OOB数据或控制标识的八进制数*/
01591: /*urg_seq:接收紧急指针的序号*/
01592: if (tp->urg_data && tp->urg_seq == *seq) {
01593: /*如果读到了数据，停止*/
01594: if (copied)
01595: break;
01596: /*signal_pending(current):检查当前进程是否有信号处理，返回不为0表示有信号需要处理。
01597: 返回 -ERESTARTSYS 表示信号函数处理完毕后重新执行信号函数前的某个系统调用。
01598: 也就是说，如果信号函数前有发生系统调用，在调度信号处理函数之前,
01599: 内核会检查系统调用的返回值，看看是不是因为这个信号而中断了系统调用.
01600: 如果返回值-ERESTARTSYS,并且当前调度的信号具备-ERESTARTSYS属性,
01601: 系统就会在用户信号函数返回之后再执行该系统调用。*/
```

```
01602: if (signal_pending(current)) {
01603: copied = timeo ? sock_intr_errno(timeo) : - EAGAIN;
01604: break;
01605: }
01606: }
01607: /*----------------------------------------------------------------------------*/
01608: /* Next get a buffer. */
01609: /*取buf*/
01610: /*skb_queue_walk(宏):对sk_receive_queue进行遍历*/
01611: skb_queue_walk(&sk->sk_receive_queue, skb) {
01612: /* Now that we have two receive queues this
01613: * shouldn't happen.
01614: */
01615: /*现在我们有两个接收队列，不应该如此*/
01616: if (WARN(before(*seq, TCP_SKB_CB(skb)->seq),
01617: "recvmsg bug: copied %X seq %X rcvnxt %X fl %X\n",
01618: *seq, TCP_SKB_CB(skb)->seq, tp->rcv_nxt,
01619: flags))
01620: break;
01621: /*偏移值=现在要读的序号-(读到的buf)开始序号*/
01622: offset = *seq - TCP_SKB_CB(skb)->seq;
01623: /*如果取出的skb为syn包,因为syn包不带数据，所以offset减去1*/
01624: if (tcp_hdr(skb)->syn)
01625: offset- -;
01626: /*如果offset比skb的长度还小，那么我要的序号肯定在skb的范围内，
01627: 在skb里能找到，那么转到found_ok_skb，意为:找到了合适的skb*/
01628: if (offset < skb->len)
01629: goto found_ok_skb;
01630: /*如果取出的skb为fin包，那么转到found_fin_ok*/
01631: if (tcp_hdr(skb)->fin)
01632: goto found_fin_ok;
01633: /*WARN是为了调试用的*/
01634: WARN(! (flags & MSG_PEEK),
01635: "recvmsg bug 2: copied %X seq %X rcvnxt %X fl %X\n",
01636: *seq, TCP_SKB_CB(skb)->seq, tp->rcv_nxt, flags);
01637: }
01638:
01639: /* Well, if we have backlog, try to process it now yet. */
01640: /*如果我们有后备队列，那么现在试着处理它*/
01641: /*如果已经拷贝的数据>=target并且后备队列的队尾指针为空*/
01642: if (copied >= target && ! sk->sk_backlog.tail)
01643: break;
```

```
01644: /*如果已经有拷贝的数据*/
01645: if (copied) {
01646: /*如果有错或者状态为TCP_CLOSE、或者已经不接收数据了、或者timeo为0，或者选择挂起*/
01647: if (sk->sk_err | |
01648: sk->sk_state == TCP_CLOSE ||
01649: (sk->sk_shutdown & RCV_SHUTDOWN) ||
01650: ! timeo ||
01651: signal_pending(current))
01652: break;
01653: } else {
01654: /*如果没有已经拷贝的数据*/
01655:
01656: /*SOCK_DONE*/
01657: if (sock_flag(sk, SOCK_DONE))
01658: break;
01659:
01660: /*如果有错*/
01661: /*sock_error:Recover an error report and clear atomically*/
01662: if (sk->sk_err) {
01663: copied = sock_error(sk);
01664: break;
01665: }
01666:
01667: /*如果已经不接收了*/
01668: if (sk->sk_shutdown & RCV_SHUTDOWN)
01669: break;
01670:
01671: /*如果TCP_CLOSE*/
01672: if (sk->sk_state == TCP_CLOSE) {
01673: /*而SOCK_DONE却没有*/
01674: if (! sock_flag(sk, SOCK_DONE)) {
01675: /* This occurs when user tries to read
01676: * from never connected socket.
01677: */
01678: /*当用户从没有连接的socket中读取数据时会出现这种情况*/
```



```
01679: copied = - ENOTCONN;
01680: break;
01681: }
01682: break;
01683: }
01684: /*如果接收超时时间为0，那么很遗憾要再来一次了*/
01685: if (! timeo) {
```

```
01686: copied = - EAGAIN;
01687: break;
01688: }
01689:
01690: if (signal_pending(current)) {
01691: copied = sock_intr_errno(timeo);
01692: break;
01693: }
01694: } ? end else ?
01695:
01696: /*把已经接收到的从接收缓冲区里清除*/
01697: tcp_cleanup_rbuf(sk, copied);
01698:
01699: /*低延时没有开启并且
01700: 直接拷贝给用户的数据(ucopy)任务现在是用户接收*/
01701: if (! sysctl_tcp_low_latency && tp->ucopy.task == user_recv) {
01702: /* Install new reader */
01703: /*建立新的读者*/
01704: if (! user_recv && ! (flags & (MSG_TRUNC | MSG_PEEK))) {
01705: user_recv = current;
01706: tp->ucopy.task = user_recv;
01707: tp->ucopy.iov = msg->msg_iov;
01708: }
01709:
01710: tp->ucopy.len = len;
01711:
01712: WARN_ON(tp->copied_seq ! = tp->rcv_nxt &&
01713: ! (flags & (MSG_PEEK | MSG_TRUNC)));
01714:
01715: /* Ugly... If prequeue is not empty, we have to
01716: * process it before releasing socket, otherwise
01717: * order will be broken at second iteration.
01718: * More elegant solution is required!!!
01719: *
01720: * Look: we have the following (pseudo)queues:
01721: *
01722: * 1. packets in flight
01723: * 2. backlog
01724: * 3. prequeue
01725: * 4. receive_queue
01726: *
01727: * Each queue can be processed only if the next ones
01728: * are empty. At this point we have empty receive_queue.
01729: * But prequeue _can_ be not empty after 2nd iteration,
```

```
01730: * when we jumped to start of loop because backlog
01731: * processing added something to receive_queue.
01732: * We cannot release_sock(), because backlog contains
01733: * packets arrived _after_ prequeued ones.
01734: *
01735: * Shortly, algorithm is clear --- to process all
01736: * the queues in order. We could make it more directly,
01737: * requeueing packets from backlog to prequeue, if
01738: * is not empty. It is more elegant, but eats cycles,
01739: * unfortunately.
01740: */
01741:
01742: /* 令人讨厌... 如果prequeue非空, 我们必须在释放socket之前处理prequeue,
01743: * 否则在第二轮顺序就全乱了
01744: * 就会需要更巧妙的解决方案!!!
01745: *
01746: * 我们有以下的(伪)队列:
01747: *
01748: * 1. in flight的包
01749: * 2. backlog
01750: * 3. prequeue
01751: * 4. receive_queue
01752: *
01753: * 只有当其下一个队列是空的时候, 才可以处理该队列。这里我们有空的receive_queue.
01754: * 但 prequeue 在2nd迭代后还不是空的,
01755: * when we jumped to start of loop because backlog
01756: * processing added something to receive_queue.
01757: * 我们不能release_sock(), because backlog contains
01758: * packets arrived _after_ prequeued ones.
01759: *
01760: * Shortly, algorithm is clear --- 按顺序处理各队列
01761: * 可以更直接一点,如果非空, 可以将包从backlog转到prequeue
```



```
01762: * 方法不巧妙, 但是很有效
01763: */
01764:
01765: /*如果接收队列为空*/
01766: if (! skb_queue_empty(&tp->ucopy.prequeue))
01767: goto   do_prequeue;
01768:
01769: /* __ Set realtime policy in scheduler __ */
01770: } ? end if !sysctl_tcp_low_laten... ?
01771:
```

```
01772: #ifdef CONFIG_NET_DMA
01773: if (tp->ucopy.dma_chan)
01774: dma_async_memcpy_issue_pending(tp->ucopy.dma_chan);
01775: #endif
01776: if (copied >= target) {
01777: /* Do not sleep, just process backlog. */
01778: release_sock(sk);
01779: lock_sock(sk);
01780: } else
01781: /*等待数据到达sk_receive_queue*/
01782: sk_wait_data(sk, &timeo);
01783:
01784: #ifdef CONFIG_NET_DMA
01785: tcp_service_net_dma(sk, false); /* Don't block */
01786: tp->ucopy.wakeup = 0;
01787: #endif
01788:
01789: if (user_recv) {
01790: int chunk;
01791:
01792: /* __ Restore normal policy in scheduler __ */
01793:
01794: if ((chunk = len - tp->ucopy.len) != 0) {
01795: NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMBACKLOG,
chunk);
01796: len -= chunk;
01797: copied += chunk;
01798: }
01799:
01800: /*如果rcv_nxt我们要接收的下一个段序号，copied_seq未读数据的头
01801: 并且prequeue队列非空*/
01802: if (tp->rcv_nxt == tp->copied_seq &&
01803: !skb_queue_empty(&tp->ucopy.prequeue)) {
01804: do_prequeue:
01805: /*prequeue队列处理过程*/
01806: tcp_prequeue_process(sk);
01807:
01808: /*数据块大小不为0*/
01809: if ((chunk = len - tp->ucopy.len) != 0) {
01810: NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE,
01810: chunk);
01811: /*要读的数据长度少了一些*/
01812: len -= chunk;
01813: /*已经拷贝到用户空间的数据多了一些*/
```

```
01814: copied += chunk;
01815: }
01816: }
01817: } ? end if user_recv ?
01818: /*MSG_PEEK:如果在recv的时候，flag字段设置了MSG_PEEK，则读取数据包的时候，
01819: 不会把该数据包从缓存队列中删除；下次读取时还是这个数据包*/
01820:
01821: /**/
01822: if ((flags & MSG_PEEK) &&
01823: (peek_seq - copied - urg_hole ! = tp->copied_seq)) {
01824: if (net_ratelimit())
01825: printk(KERN_DEBUG "TCP(%s:%d): Application bug, race in MSG_PEEK.\n",
01826: current->comm, task_pid_nr(current));
01827: peek_seq = tp->copied_seq;
01828: }
01829: continue;
01830:
01831: found_ok_skb:
01832: /* Ok so how much can we use? */
01833: /*找到了合适的包以后*/
01834: /*可用长度*/
01835: used = skb->len - offset;
01836: /*还要取的长度<可用长度，那就先取要取的长度，让可用长度和len相等*/
01837: if (len < used)
01838: used = len;
01839:
01840: /* Do we have urgent data here? */
01841: /*有没有要取紧急数据?*/
```

D:\linux-2.6.39.2\linux-2.6.39.2\net\ipv4\tcp.c

Page 24

```
01842: if (tp->urg_data) {
01843: /*要取的urg数据序号偏移量*/
01844: u32 urg_offset = tp->urg_seq - *seq;
01845: /*如果所取紧急数据序号在可用包长度范围内*/
01846: if (urg_offset < used) {
01847: /*如果urg_offset为0*/
01848: if (! urg_offset) {
01849: if (! sock_flag(sk, SOCK_URGINLINE)) {
01850: ++*seq;
01851: urg_hole++;
01852: offset++;
01853: used- -;
01854: /*如果used减去1以后为0，那么就是说现在只有紧急数据了，那么直接跳过拷贝*/
01855: if (! used)
```

```
01856: goto    skip_copy;
01857: }
01858: } else
01859: used = urg_offset;
01860: }
01861: } ? end if tp->urg_data ?
01862:
01863: if (! (flags & MSG_TRUNC)) {
01864: #ifdef CONFIG_NET_DMA
01865: if (! tp->ucopy.dma_chan && tp->ucopy.pinned_list)
01866: tp->ucopy.dma_chan = dma_find_channel(DMA_MEMCPY);
01867:
01868: if (tp->ucopy.dma_chan) {
01869: tp->ucopy.dma_cookie = dma_skb_copy_datagram_iovec(
01870: tp->ucopy.dma_chan, skb, offset,
01871: msg->msg_iov, used,
01872: tp->ucopy.pinned_list);
01873:
01874: if (tp->ucopy.dma_cookie < 0) {
01875:
01876: printk(KERN_ALERT "dma_cookie < 0\n");
01877:
01878: /* Exception. Bailout! */
01879: if (! copied)
01880: copied = - EFAULT;
01881: break;
01882: }
01883:
01884: dma_async_memcpy_issue_pending(tp->ucopy.dma_chan);
01885:
01886: if ((offset + used) == skb->len)
01887: copied_early = 1;
01888:
01889: } ? end if tp->ucopy.dma_chan ? else
01890: #endif
01891: {
01892: /*拷贝数据包到iovec*/
01893: err = skb_copy_datagram_iovec(skb, offset,
01894: msg->msg_iov, used);
01895: if (err) {
01896: /* Exception. Bailout! */
01897: if (! copied)
01898: copied = - EFAULT;
01899: break;
```

```
01900: }
01901: }
01902: } ? end if !(flags&MSG_TRUNC) ?
01903:
01904: /*接收成功一个包，那么对*seq，copied，len进行调整*/
01905: *seq += used;
01906: copied += used;
01907: len -= used;
01908:
01909: /*在每次将数据拷贝到用户空间后都要调用下面这个函数.
01910: 计算适当的TCP接收缓存空间*/
01911: tcp_rcv_space_adjust(sk);
01912:
01913: skip_copy:
01914: if (tp->urg_data && after(tp->copied_seq, tp->urg_seq)) {
01915: tp->urg_data = 0;
01916: tcp_fast_path_check(sk);
01917: }
01918: /*还有数据没取，继续*/
01919: if (used + offset < skb->len)
01920: continue;
01921:
01922: /*如果是fin包，去found_fin_ok*/
```



```
01923: if (tcp_hdr(skb)->fin)
01924: goto    found_fin_ok;
01925: /*如果没有设置MSG_PEEK，那么要清除缓存中已经拷贝过的数据*/
01926: if (! (flags & MSG_PEEK)) {
01927: sk_eat_skb(sk, skb, copied_early);
01928: copied_early = 0;
01929: }
01930: /*继续*/
01931: continue;
01932:
01933: found_fin_ok:
01934: /* Process the FIN. */
01935: /*处理FIN包*/
01936: ++*seq;
01937: if (! (flags & MSG_PEEK)) {
01938: sk_eat_skb(sk, skb, copied_early);
01939: copied_early = 0;
01940: }
01941: break;
```

```
01942: } ? end do ? while (len > 0);
01943: /*----------------------跳出while循环了-----------------------------------------*/
01944: if (user_recv) {
01945: if (! skb_queue_empty(&tp->ucopy.prequeue)) {
01946: int chunk;
01947:
01948: tp->ucopy.len = copied > 0 ? len : 0;
01949:
01950: /*处理prequeue了*/
01951: tcp_prequeue_process(sk);
01952:
01953: if (copied > 0 && (chunk = len - tp->ucopy.len) ! = 0) {
01954: NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE,
chunk);
01955: len -= chunk;
01956: copied += chunk;
01957: }
01958: }
01959:
01960: tp->ucopy.task = NULL;
01961: tp->ucopy.len = 0;
01962: }
01963:
01964: #ifdef CONFIG_NET_DMA
01965: tcp_service_net_dma(sk, true); /* Wait for queue to drain */
01966: tp->ucopy.dma_chan = NULL;
01967:
01968: if (tp->ucopy.pinned_list) {
01969: dma_unpin_iovec_pages(tp->ucopy.pinned_list);
01970: tp->ucopy.pinned_list = NULL;
01971: }
01972: #endif
01973:
01974: /* According to UNIX98, msg_name/msg_namelen are ignored
01975: * on connected socket. I was just happy when found this 8) --ANK
01976: */
01977:
01978: /* Clean up data we have read: This will do ACK frames. */
01979: /*清除我们读了的数据:可能会发ACK*/
01980: tcp_cleanup_rbuf(sk, copied);
01981:
01982: release_sock(sk);
01983: return copied;
01984:
```

```
01985: out:
01986: release_sock(sk);
01987: return err;
01988:
01989: recv_urg:
01990: /*处理带外数据部分，转入紧急数据处理*/
01991: err = tcp_recv_urg(sk, msg, len, flags);
01992: goto    out;
01993: } ? end tcp_recvmsg ?
01994: EXPORT_SYMBOL(tcp_recvmsg);
```

## 5.14tcp_v4_do_rcv

/tcp/ipv4/tcp_ipv4.c

```
01576: /* The socket must have it's spinlock held when we get
01577: * here.
01578: *
01579: * We have a potential double-lock case here, so even when
01580: * doing backlog processing we use the BH locking scheme.
01581: * This is because we cannot sleep with the original spinlock
01582: * held.
01583: */
01584: /*由mytcp_v4_rcv函数调用，从下往上走的*/
01585: int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
01586: {
01587: struct sock *rsk;
01588: #ifdef CONFIG_TCP_MD5SIG
01589: /*
01590: * We really want to reject the packet as early as possible
01591: * if:
01592: * o We're expecting an MD5'd packet and this is no MD5 tcp option
01593: * o There is an MD5 option and we're not expecting one
01594: */
01595: if (tcp_v4_inbound_md5_hash(sk, skb))
01596: goto    discard;
01597: #endif
01598: /*sock的状态是TCP_ESTABLISHED,如果状态是TCP_ESTABLISHED，表明连接已经建立，
01599: 调用tcp_rcv_established函数将SKB加入sk-> receive_queue中
01600: tcp_rcv_established中语句 __skb_queue_tail(&sk->sk_receive_queue, skb);完成队列的添加*/
01601: /*快速通道*/
01602: if (sk->sk_state == TCP_ESTABLISHED) { /* Fast path */
01603: /*RPS是针对于多核的 Receive Packet Steering ，略*/
```

```
01604: sock_rps_save_rxhash(sk, skb->rxhash);
01605: if (tcp_rcv_established(sk, skb, tcp_hdr(skb), skb->len)) {
01606: rsk = sk;
01607: goto    reset;
01608: }
01609: return 0;
01610: }
01611: /*检查包长度和校验和是否正确*/
01612: if (skb->len < tcp_hdrlen(skb) || tcp_checksum_complete(skb))
01613: goto    csum_err;
01614:
01615:
01616: if (sk->sk_state == TCP_LISTEN) {
01617: /*tcp_v4_hnd_req的返回值，不同情况下不同。
01618: NULL 出现错误
```

D:\linux-2.6.39.2\linux-2.6.39.2\net\ipv4\tcp_ipv4.c

Page 22

```
01619: nsk==sk 接受到SYN
01620: nsk! =sk 接受到ACK
01621: 接受到ACK包时，tcp_v4_hnd_req函数会新建一个sock结构，并设置其初始状态为SYN_RECV，
01622: 并返回新建的sock结构。接着调用tcp_child_process函数，改变新建的sock的状态为ESTABLISHED.*/
01623: struct sock *nsk = tcp_v4_hnd_req(sk, skb);
01624: if (! nsk)
01625: goto    discard;
01626:
01627: if (nsk ! = sk) {
01628: if (tcp_child_process(sk, nsk, skb)) {
01629: rsk = nsk;
01630: goto    reset;
01631: }
01632: return 0;
01633: }
01634: } else
01635: sock_rps_save_rxhash(sk, skb->rxhash);
01636: /*处理sk的状态转换
01637: 实现RFC 793中所有接受处理过程中的sk状态变化，除了ESTABLISHED和TIME_WAIT*/
01638: if (tcp_rcv_state_process(sk, skb, tcp_hdr(skb), skb->len)) {
01639: rsk = sk;
01640: goto    reset;
01641: }
01642: return 0;
01643:
01644: reset:
01645: tcp_v4_send_reset(rsk, skb);
```

```
01646: discard:
01647: kfree_skb(skb);
01648: /* Be careful here. If this function gets more complicated and
01649: * gcc suffers from register pressure on the x86, sk (in %ebx)
01650: * might be destroyed here. This current version compiles correctly,
01651: * but you have been warned.
01652: */
01653: return 0;
01654:
01655: csum_err:
01656: TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_INERRS);
01657: goto    discard;
01658: } ? end tcp_v4_do_rcv ?
01659: EXPORT_SYMBOL(tcp_v4_do_rcv);
```