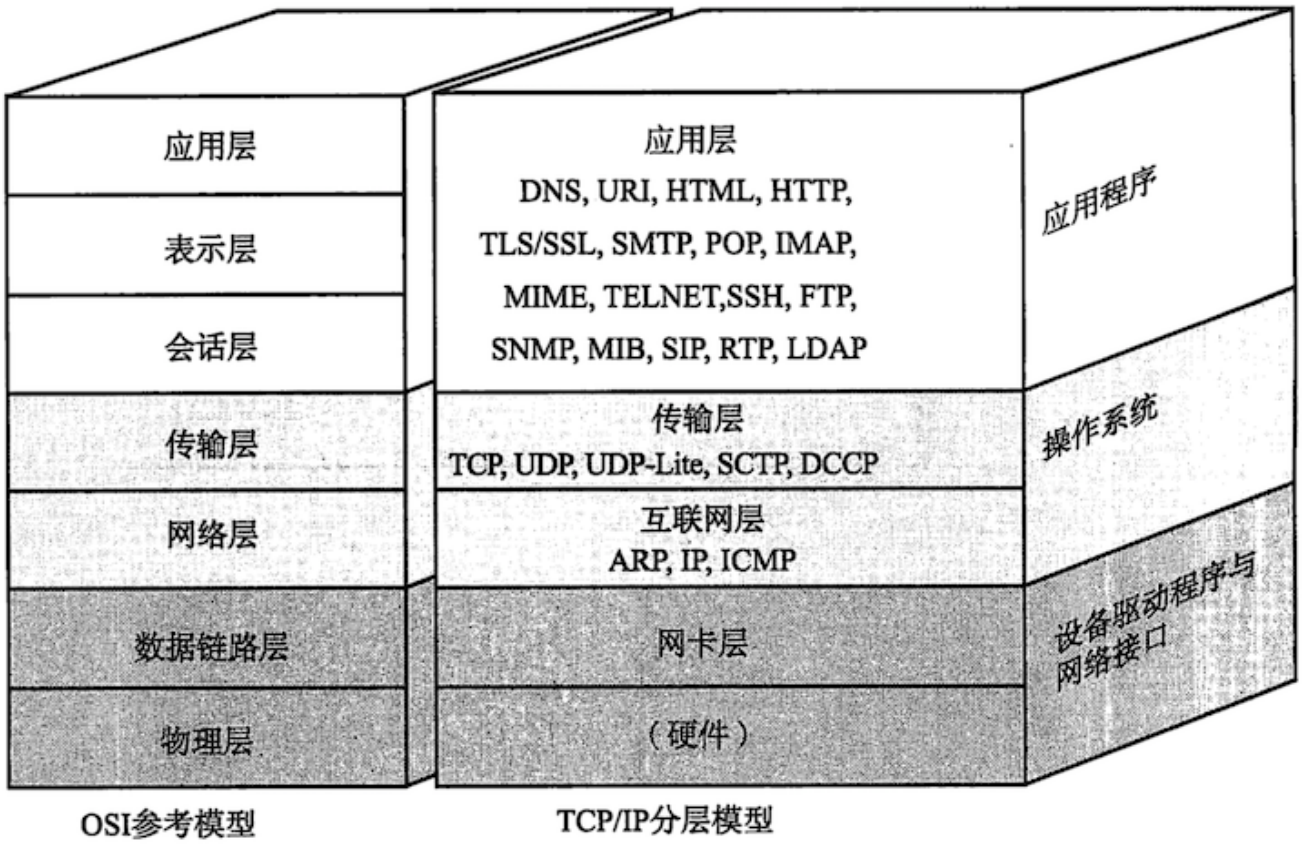
走读网络协议栈 tcp 发送数据的内核源码（Linux - 5.0.1 下载）。

数据发送实现比较复杂，牵涉到 OSI 所有分层，所以需要对每个分层都要有所了解，才能理清工作流程和实现思路。

- content
  {:toc}

1. 通信分层

- OSI 模型。

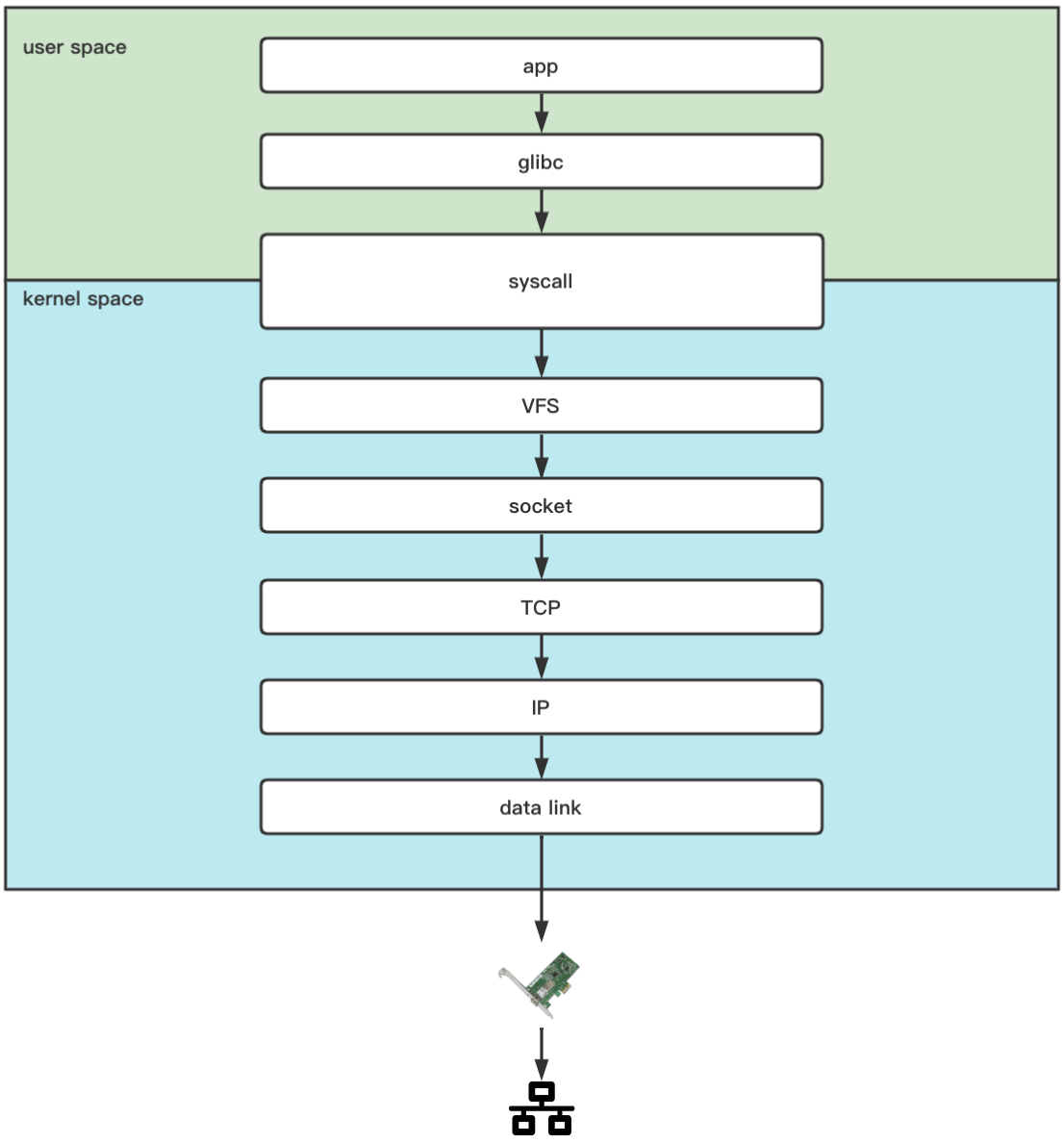


> 图片来源：《图解 TCP_IP》

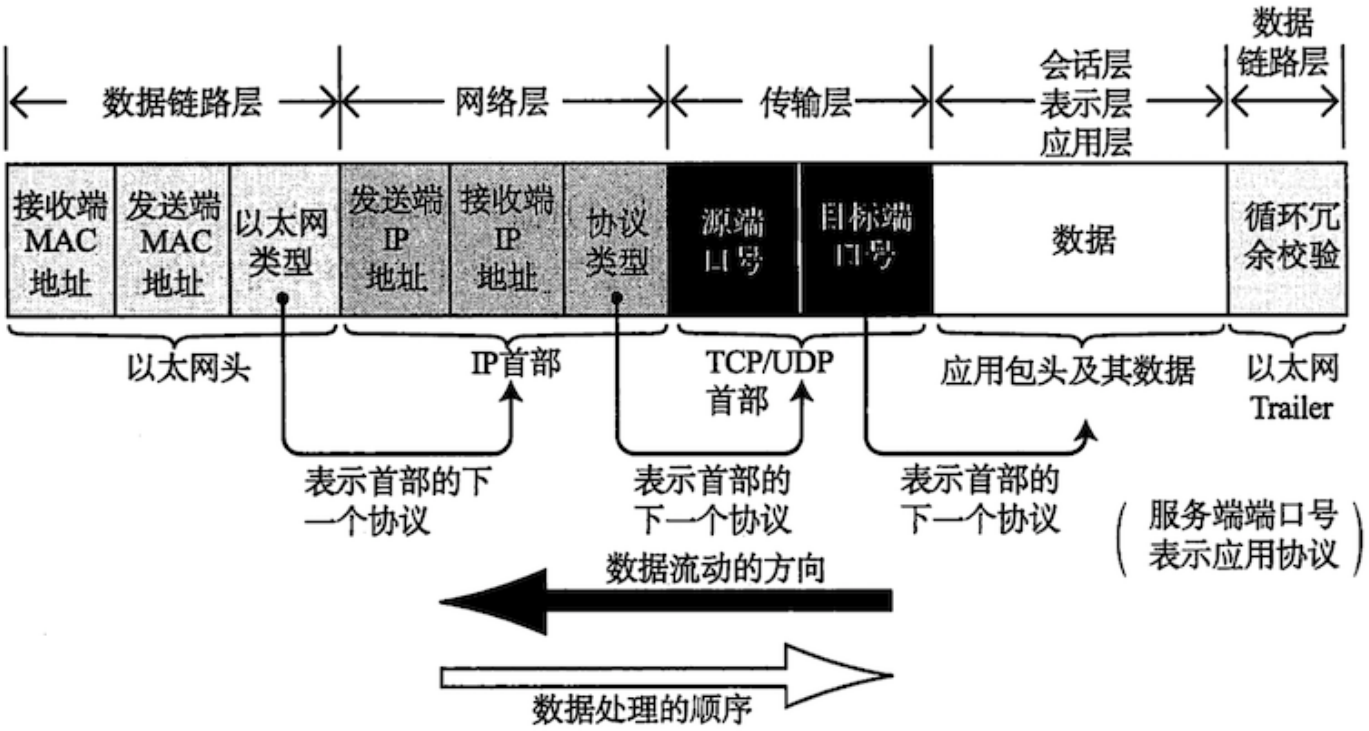
- 内核函数堆栈。
  write 发送数据，详细工作流程可以参考内核函数堆栈。

```
__dev_queue_xmit(struct sk_buff * skb, struct net_device * sb_dev) (/root/linux-5.0.1/ne
dev_queue_xmit(struct sk_buff * skb) (/root/linux-5.0.1/net/core/dev.c:3897)
# 网络介质层
neigh_hh_output() (/root/linux-5.0.1/include/net/neighbour.h:498)
neigh_output() (/root/linux-5.0.1/include/net/neighbour.h:506)
ip_finish_output2(struct net * net, struct sock * sk, struct sk_buff * skb) (/root/linux
NF_HOOK_COND() (/root/linux-5.0.1/include/linux/netfilter.h:278)
ip_output(struct net * net, struct sock * sk, struct sk_buff * skb) (/root/linux-5.0.1/n
# 网络 ip 层。
__tcp_transmit_skb(struct sock * sk, struct sk_buff * skb, int clone_it, gfp_t gfp_mask,
tcp_transmit_skb() (/root/linux-5.0.1/net/ipv4/tcp_output.c:1176)
tcp_write_xmit(struct sock * sk, unsigned int mss_now, int nonagle, int push_one, gfp_t
__tcp_push_pending_frames(struct sock * sk, unsigned int cur_mss, int nonagle) (/root/li
tcp_push(struct sock * sk, int flags, int mss_now, int nonagle, int size_goal) (/root/li
tcp_sendmsg_locked(struct sock * sk, struct msghdr * msg, size_t size) (/root/linux-5.0.
tcp_sendmsg(struct sock * sk, struct msghdr * msg, size_t size) (/root/linux-5.0.1/net/i
# tcp 传输层。
sock_sendmsg_nosec() (/root/linux-5.0.1/net/socket.c:622)
sock_sendmsg(struct socket * sock, struct msghdr * msg) (/root/linux-5.0.1/net/socket.c
sock_write_iter(struct kiocb * iocb, struct iov_iter * from) (/root/linux-5.0.1/net/sock
# socket 层（应用层）。
call_write_iter() (/root/linux-5.0.1/include/linux/fs.h:1863)
new_sync_write() (/root/linux-5.0.1/fs/read_write.c:474)
__vfs_write(struct file * file, const char * p, size_t count, loff_t * pos) (/root/linux
vfs_write(struct file * file, const char * buf, size_t count, loff_t * pos) (/root/linux
# vfs 虚拟文件系统管理层（应用层）。
ksys_write(unsigned int fd, const char * buf, size_t count) (/root/linux-5.0.1/fs/read_w
do_syscall_64(unsigned long nr, struct pt_regs * regs) (/root/linux-5.0.1/arch/x86/entry
entry_SYSCALL_64() (/root/linux-5.0.1/arch/x86/entry/entry_64.S:175)
# 系统调用层（应用层）。
...
```

> 参考：vscode + gdb 远程调试 linux (EPOLL) 内核源码



- 通信分层数据包封装格式。



> 图片来源：《图解 TCP_IP》

2. VFS 层

2.1. 文件与socket

socket 是 Linux 一种 **特殊文件**，socket 在创建时（ `sock_alloc_file` ）会关联对应的文件处理，所以我们在 TCP 通信过程中，发送数据，用户层调用 `write` 接口，在内核里实际是调用了 `sock_write_iter` 接口。

> 详细参考：《[内核源码] 网络协议栈 - socket (tcp)》 - 4.1 文件部分

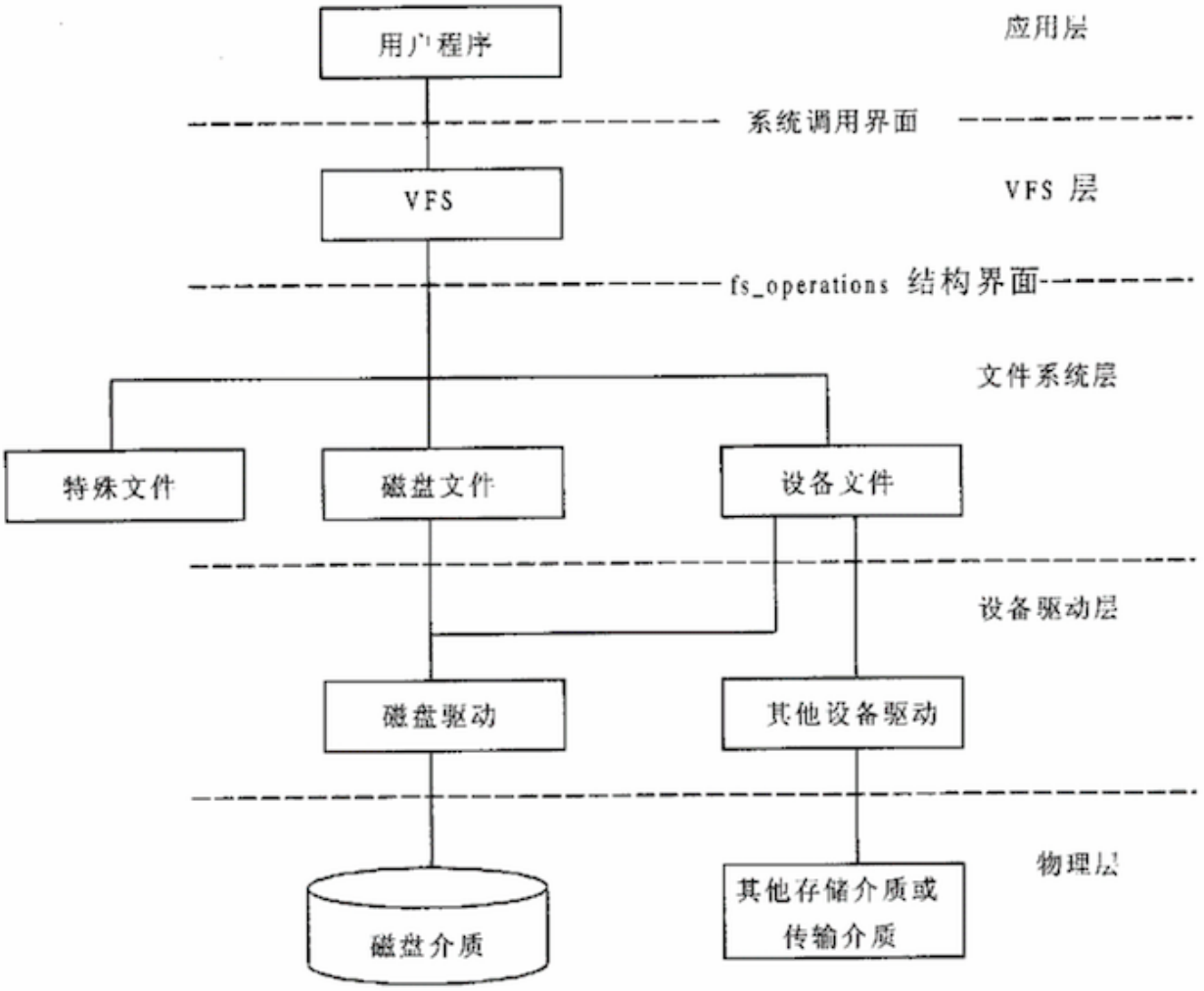


图 5.3　Linux 文件系统的层次结构

> 图片来源：《Linux 内核源代码情景分析》- 第五章 - 文件系统

```
/* ./include/linux/fs.h */
struct file_operations {
    struct module *owner;
    ...
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ...
} __randomize_layout;

/* ./net/socket.c */
static const struct file_operations socket_file_ops = {
    .owner      =    THIS_MODULE,
    ...
    .read_iter  =    sock_read_iter,
    .write_iter =    sock_write_iter,
    ...
};

/* net/socket.c */
struct file *sock_alloc_file(struct socket *sock, int flags, const char *dname) {
    ...
    file = alloc_file_pseudo(SOCK_INODE(sock), sock_mnt, dname,
                O_RDWR | (flags & O_NONBLOCK),
                &socket_file_ops);
    ...
    /* 文件和socket相互建立联系。 */
    sock->file = file;
    file->private_data = sock;
    return file;
}
```

2.2. 系统调用

发送数据从用户层到内核，通过 fd 文件描述符，找到对应的文件，然后再找到与文件关联的对应的 socket，进行发送数据。

```
write --> fd --> file --> sock_sendmsg
```

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
        size_t, count) {
    return ksys_write(fd, buf, count);
}

ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count) {
    struct fd f = fdget_pos(fd);
    ...
    if (f.file) {
        ...
        ret = vfs_write(f.file, buf, count, &pos);
        ...
    }
    ...
}

ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t *pos)
    ...
    ret = __vfs_write(file, buf, count, pos);
    ...
}

ssize_t __vfs_write(struct file *file, const char __user *p, size_t count,
            loff_t *pos) {
    if (file->f_op->write)
        return file->f_op->write(file, p, count, pos);
    /* sock_write_iter */
    else if (file->f_op->write_iter)
        return new_sync_write(file, p, count, pos);
    else
        return -EINVAL;
}

static ssize_t new_sync_write(struct file *filp, const char __user *buf, size_t len, lo
    ...
    ret = call_write_iter(filp, &kiocb, &iter);
    ...
}

static inline ssize_t call_write_iter(struct file *file, struct kiocb *kio,
                      struct iov_iter *iter) {
    /* sock_write_iter */
    return file->f_op->write_iter(kio, iter);
}

static ssize_t sock_write_iter(struct kiocb *iocb, struct iov_iter *from) {
    struct file *file = iocb->ki_filp;
    struct socket *sock = file->private_data;
    struct msghdr msg = {
        .msg_iter = *from,
        .msg_iocb = iocb
```

```
    };
    ssize_t res;
    ...
    res = sock_sendmsg(sock, &msg);
    *from = msg.msg_iter;
    return res;
}

int sock_sendmsg(struct socket *sock, struct msghdr *msg) {
    int err = security_socket_sendmsg(sock, msg,
                      msg_data_left(msg));

    return err ?: sock_sendmsg_nosec(sock, msg);
}
```

3. socket 层

fd --> file --> socket --> sock --> tcp

- 接口调用。

```
/* net/socket.c */
static inline int sock_sendmsg_nosec(struct socket *sock, struct msghdr *msg) {
    /* inet_sendmsg */
    int ret = sock->ops->sendmsg(sock, msg, msg_data_left(msg));
    ...
}

/* net/ipv4/af_inet.c */
int inet_sendmsg(struct socket *sock, struct msghdr *msg, size_t size) {
    struct sock *sk = sock->sk;
    ...
    return sk->sk_prot->sendmsg(sk, msg, size);
}

/* net/ipv4/tcp.c */
int tcp_sendmsg(struct sock *sk, struct msghdr *msg, size_t size) {
    ...
    ret = tcp_sendmsg_locked(sk, msg, size);
    ...
}
```

- sock 操作对象。

```
socket.ops --> inetsw_array[socket.type].ops --> inet_stream_ops
```

- 结构关联。

```
/* ./net/ipv4/af_inet.c */
const struct proto_ops inet_stream_ops = {
    .family        = PF_INET,
    ...
    .sendmsg       = inet_sendmsg,
    ...
};

/* ./net/ipv4/tcp_ipv4.c */
struct proto tcp_prot = {
    .name             = "TCP",
    ...
    .sendmsg          = tcp_sendmsg,
    ...
};

/* af_inet.c */
static struct inet_protosw inetsw_array[] = {
    {
        .type     = SOCK_STREAM,
        .protocol = IPPROTO_TCP,
        .prot     = &tcp_prot,
        .ops      = &inet_stream_ops,
        .flags    = INET_PROTOSW_PERMANENT | INET_PROTOSW_ICSK,
    },
    ...
};

struct socket {
    ...
    short              type;
    ...
    struct file        *file;
    struct sock        *sk;
    const struct proto_ops  *ops;
};

/* inet_stream_ops 与 socket.ops 关联。 */
static int inet_create(struct net *net, struct socket *sock, int protocol, int kern) {
    ...
    struct inet_protosw *answer;
    struct proto *answer_prot;
    ...
    /* 查找对应的协议。 */
    list_for_each_entry_rcu(answer, &inetsw[sock->type], list) {
        ...
    }
    ...
    /* 关联。 */
    sock->ops = answer->ops;
    ...
```

```
}

/* tcp_prot 与 socket 关联。 */
struct sock *sk_alloc(struct net *net, int family, gfp_t priority,
                      struct proto *prot, int kern) {
    struct sock *sk;
    ...
    sk->sk_prot = sk->sk_prot_creator = prot;
    ...
}
```

> 详细参考：[内核源码] 网络协议栈 - socket (tcp)

4. TCP 层

4.1. sk_buff

socket 数据缓存，sk_buff 用于保存接收或者发送的数据报文信息，目的为了方便网络协议栈的各层间进行无缝传递数据。sk_buff 数据存储的两个区域：

- `data`： 连续数据区（数据拷贝）。
- `skb_shared_info`： 共享数据区。

> 详细请参考：《Linux 内核源码剖析 - TCP/IP 实现》- 第三章 - 套接口缓存

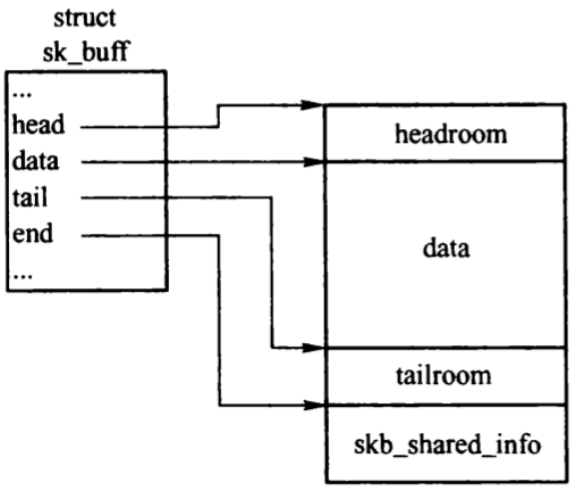


图 3-4　skb 的 head/end 指针和 rxsj data/tail 指针

> 图片来源：《Linux 内核源码剖析 - TCP/IP 实现》- 3.2.3

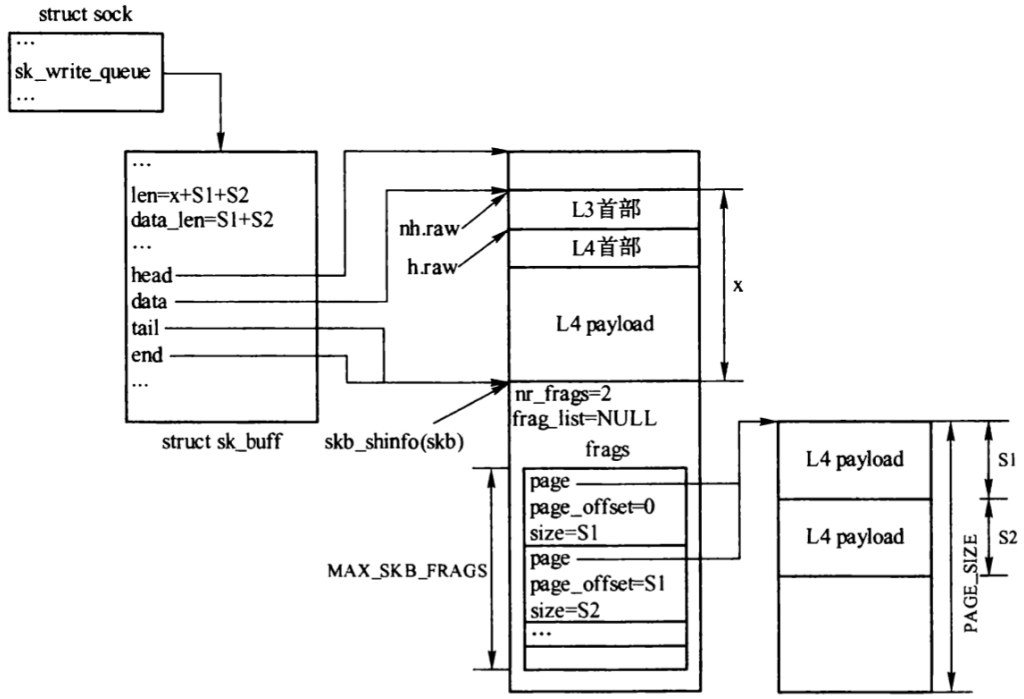


图 3-9　只启用聚合分散 I/O 分片的报文

> 图片来源：《Linux 内核源码剖析 - TCP/IP 实现》- 3.2.3

tcp 的数据输出，数据首先是从应用层在流入内核，内核会将应用层传入的数据进行拷贝，拷贝到 `sk_buff` 链表中进行发送，参考下图。

- `sk_write_queue` ： 发送队列的双向链表头。
- `sk_send_head` ： 指向发送队列中下一个要发送的数据包。

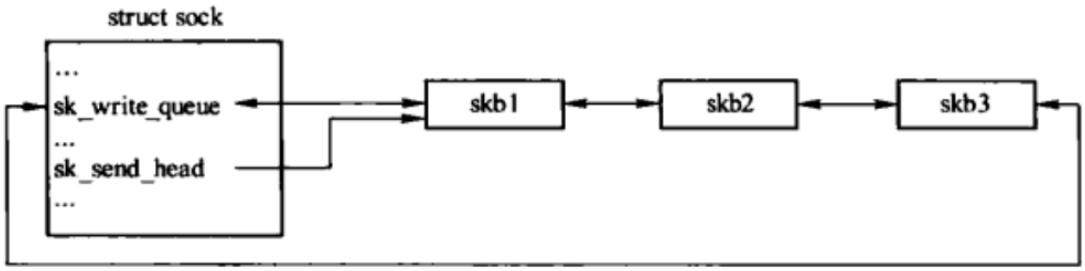


图 30-1　sk_write_queue 和 sk_send_head 之间的关系

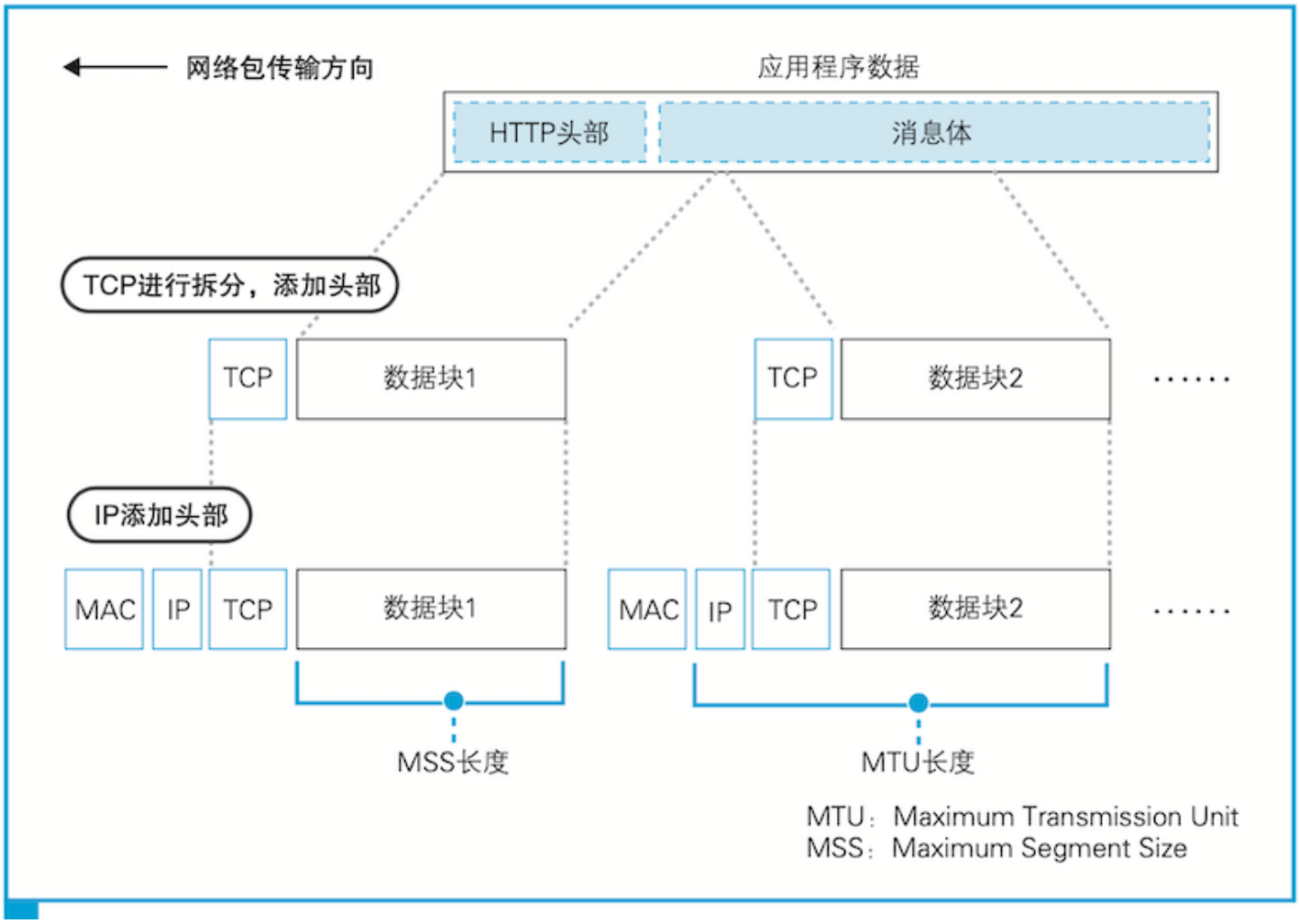
> 图片来源：《Linux 内核源码剖析 - TCP/IP 实现》- 30.1

4.2. mtu / mss

网络上传输的网络包大小是有限制的，理解 MTU 和 MSS 这两个限制概念。

- MTU：Maximum Transmission Unit。

- MSS：Max Segment Size。



图片来源：《网络是怎样连接的》

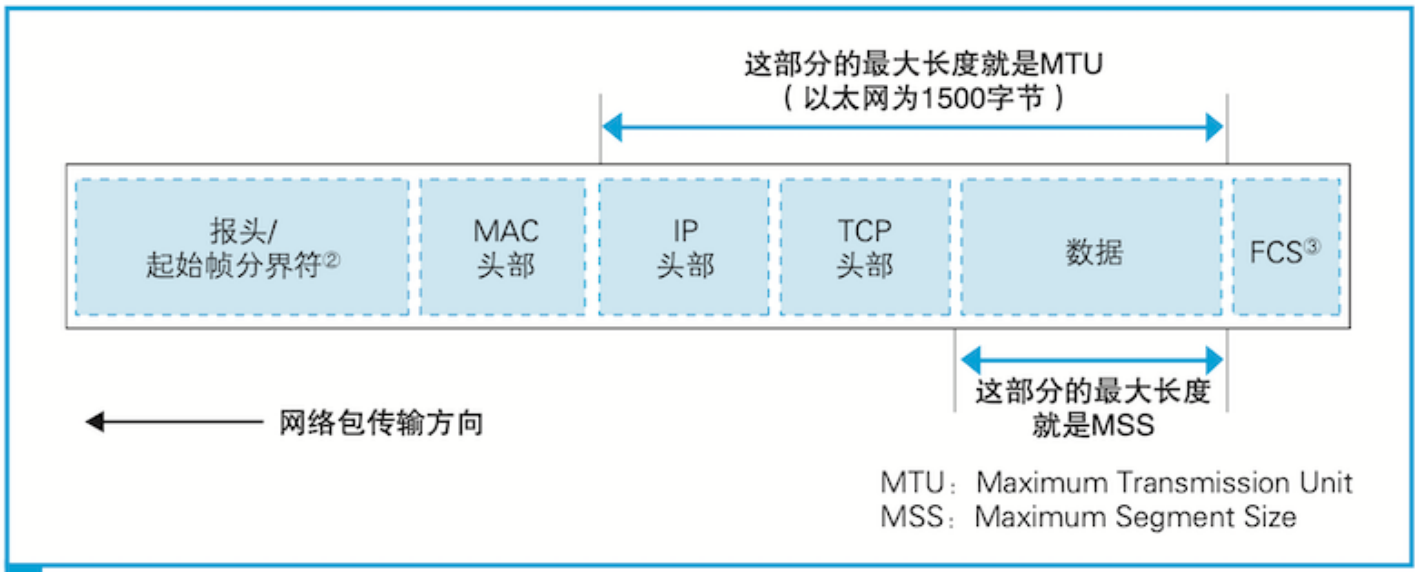


图 2.5　MTU 与 MSS

图片来源：《网络是怎样连接的》

4.3. 数据发送逻辑

`tcp_sendmsg_locked` 主要工作是要把用户层的数据填充到内核的发送队列进行发送。

> 源码注释参考：《Linux 内核源码剖析 - TCP/IP 实现》- 下册 - 第 30 章 TCP 的输出。

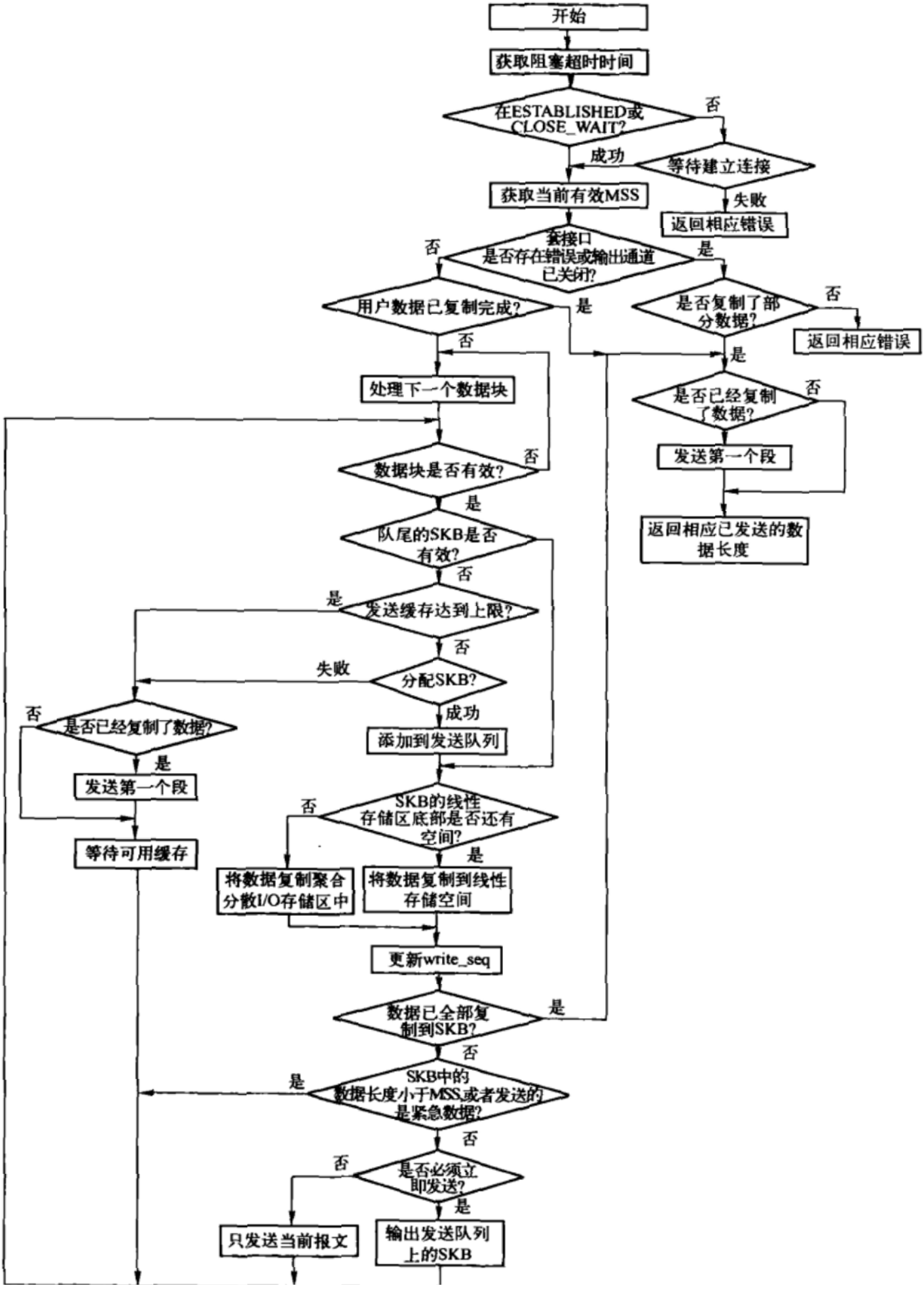

开始
获取阻塞超时时间
在ESTABLISHED或CLOSE_WAIT?
否
成功
等待建立连接
失败
返回相应错误
获取当前有效MSS
套接口是否存在错误或输出通道已关闭?
否
是
用户数据已复制完成?
是
是否复制了部分数据?
否
返回相应错误
是
否
处理下一个数据块
是否已经复制了数据?
否
发送第一个段
返回相应已发送的数据长度
数据块是否有效?
否
是
队尾的SKB是否有效?
否
是
发送缓存达到上限?
否
失败
分配SKB?
成功
否
是否已经复制了数据?
是
添加到发送队列
发送第一个段
SKB的线性存储区底部是否还有空间?
否
是
等待可用缓存
将数据复制聚合分散I/O存储区中
将数据复制到线性存储空间
更新write_seq
数据已全部复制到SKB?
是
否
是
SKB中的数据长度小于MSS或者发送的是紧急数据?
否
否
是否必须立即发送?
是
只发送当前报文
输出发送队列上的SKB

图 30-13　tcp_sendmsg()流程图

```
int tcp_sendmsg_locked(struct sock *sk, struct msghdr *msg, size_t size) {
    struct tcp_sock *tp = tcp_sk(sk);
    struct ubuf_info *uarg = NULL;
    struct sk_buff *skb;
    struct sockcm_cookie sockc;
    int flags, err, copied = 0;
    int mss_now = 0, size_goal, copied_syn = 0;
    bool process_backlog = false;
    bool zc = false;
    long timeo;
    ...
    /* 获取等待的时间，如果阻塞模式，获取超时时间，非阻塞为 0。 */
    timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);
    ...
    /* Wait for a connection to finish. One exception is TCP Fast Open
     * (passive side) where data is allowed to be sent before a connection
     * is fully established. */
    if (((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT)) &&
        !tcp_passive_fastopen(sk)) {
        /* 等待连接建立。 */
        err = sk_stream_wait_connect(sk, &timeo);
        if (err != 0)
            goto do_error;
    }
    ...
restart:
    /* 获取当前有效的 mss。
     * mtu: max transmission unit.
     * mss: max segment size. (mtu - (ip header size) - (tcp header size)).
     * GSO: Generic Segmentation Offload.
     * size_goal 表示数据报到达网络设备时，数据段的最大长度，该长度用来分割数据，
     * TCP 发送段时，每个 SKB 的大小不能超过该值。
     * 不支持 GSO 情况下， size_goal 就等于 MSS，如果支持 GSO，
     * 那么 size_goal 是 mss 的整数倍，数据报发送到网络设备后再由网络设备根据 MSS 进行分割。
     */
    mss_now = tcp_send_mss(sk, &size_goal, flags);
    ...
    /* 将 msg 数据拷贝到 skb，等待发送。 */
    while (msg_data_left(msg)) {
        int copy = 0;

        /* 从等待发送数据链表中，取最后一个 skb，将将要发送的数据填充到 skb，等待发送。 */
        skb = tcp_write_queue_tail(sk);
        if (skb)
            /* size_goal - skb->len 判断 skb 是否已满，大于零说明 skb 还有剩余空间，
             * 还能往 skb 追加填充数据，组成一个 mss 的数据包，发往 ip 层。 */
            copy = size_goal - skb->len;

        /* 如果当前 skb 空间不足，那么要重新创建一个 sk_buffer 装载数据。
           或者被设置了 eor 标记不能合并。*/
        if (copy <= 0 || !tcp_skb_can_collapse_to(skb)) {
```

```
            bool first_skb;
            int linear;

new_segment:
            /* 如果发送队列的总大小（sk_wmem_queued）>= 发送缓存上限（sk_sndbuf）
             * 或者发送缓冲区中尚未发送的数据量，超过了用户的设置值，那么进入等待状态。*/
            if (!sk_stream_memory_free(sk))
                goto wait_for_sndbuf;

            /* 重新分配一个 sk_buffer 结构。 */
            skb = sk_stream_alloc_skb(sk, 0, sk->sk_allocation,
                    tcp_rtx_and_write_queues_empty(sk));
            ...
            /* 将 skb 添加进发送队列尾部。 */
            skb_entail(sk, skb);
            /* skb 数据缓冲区大小是 size_goal。 */
            copy = size_goal;
            ...
        }

        /* Try to append data to the end of skb. */
        if (copy > msg_data_left(msg))
            copy = msg_data_left(msg);

        /* skb 的线性存储区底部是否还有空间。 */
        if (skb_availroom(skb) > 0 && !zc) {
            /* We have some space in skb head. Superb! */
            copy = min_t(int, copy, skb_availroom(skb));
            /* 将数据拷贝到连续的数据区域。*/
            err = skb_add_data_nocache(sk, skb, &msg->msg_iter, copy);
            if (err)
                goto do_fault;
        } else if (!zc) {
            bool merge = true;
            int i = skb_shinfo(skb)->nr_frags;
            struct page_frag *pfrag = sk_page_frag(sk);
            ...
            copy = min_t(int, copy, pfrag->size - pfrag->offset);
            ...
            /* 如果 skb 的线性存储区底部已经没有空间了，
             * 将数据拷贝到 skb 的 struct skb_shared_info 结构指向的不需要连续的页面区域。 */
            err = skb_copy_to_page_nocache(sk, &msg->msg_iter, skb,
                                pfrag->page,
                                pfrag->offset,
                                copy);
            ...
            pfrag->offset += copy;
        } else {
            /* zero copy. */
        }
```

```
        /* 如果复制的数据长度为零（或者第一次拷贝），那么取消 PSH 标志。 */
        if (!copied)
            TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_PSH;

        /* 更新发送队列的最后一个序号 write_seq。 */
        tp->write_seq += copy;
        /* 更新 skb 的结束序号。 */
        TCP_SKB_CB(skb)->end_seq += copy;
        /* 初始化 gso 分段数 gso_segs. */
        tcp_skb_pcount_set(skb, 0);

        copied += copy;
        if (!msg_data_left(msg)) {
            if (unlikely(flags & MSG_EOR))
                /* #define MSG_EOR 0x80 -- End of record */
                TCP_SKB_CB(skb)->eor = 1;
            /* 用户层数据已经拷贝完毕，进行发送。 */
            goto out;
        }

        /* 如果当前 skb 还可以填充数据，或者发送的是带外数据，或者使用 tcp repair 选项，
         * 那么继续拷贝数据，先不发送。*/
        if (skb->len < size_goal || (flags & MSG_OOB) || unlikely(tp->repair))
            continue;

        /* 检查是否必须立即发送。 */
        if (forced_push(tp)) {
            tcp_mark_push(tp, skb);
            /* 积累的数据包数量太多了，需要发送出去。*/
            __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
        } else if (skb == tcp_send_head(sk))
            /* 如果是第一个网络包，那么只发送当前段。 */
            tcp_push_one(sk, mss_now);
        continue;
        ...
wait_for_sndbuf:
        /* 发送队列中段数据总长度已经达到了发送缓冲区的长度上限，那么设置 SOCK_NOSPACE。*/
        set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
wait_for_memory:
        /* 在进入睡眠等待前，如果已有数据从用户空间复制过来，那么通过 tcp_push 先发送出去。 */
        if (copied)
            tcp_push(sk, flags & ~MSG_MORE, mss_now,
                TCP_NAGLE_PUSH, size_goal);

        /* 进入睡眠，等待内存空闲信号唤醒。 */
        err = sk_stream_wait_memory(sk, &timeo);
        if (err != 0)
            goto do_error;

        /* 睡眠后 MSS 和 TSO 段长可能会发生变化，重新计算。 */
        mss_now = tcp_send_mss(sk, &size_goal, flags);
```

```
    }
    ...
out:
    /* 在连接状态下，在发送过程中，如果有正常的退出，或者由于错误退出，
     * 但是已经有复制数据了，都会进入发送环节。 */
    if (copied) {
        /* 如果已经有数据复制到发送队列了，就尝试立即发送。 */
        tcp_tx_timestamp(sk, sockc.tsflags);
        /* 是否能立即发送数据要看是否启用了 Nagle 算法。 */
        tcp_push(sk, flags, mss_now, tp->nonagle, size_goal);
    }
    return copied;
    ...
}
```

5. 参考

- 《图解 TCP_IP》
- 《网络是怎样连接的》
- 《Linux 内核源代码情景分析》
- 《Linux 内核源码剖析 - TCP/IP 实现》
- vscode + gdb 远程调试 linux (EPOLL) 内核源码
- [内核源码] 网络协议栈 - socket (tcp)

- Linux socket 数据发送类函数实现(四)
- Linux内核中sk_buff结构详解
- sk_buff 结构体 以及 完全解释
- sk_buff 整理笔记（一、数据结构）
- Linux网络系统原理笔记
- TCP发送源码学习(1)--tcp_sendmsg
- Linux操作系统学习笔记（二十二）网络通信之发包
- TCP数据发送之TSO/GSO
- linux tcp GSO和TSO实现
- 浅析TCP协议报文生成过程
- Linux Kernel TCP/IP Stack|Linux网络硬核系列
- TCP的发送系列 — tcp_sendmsg()的实现（一）