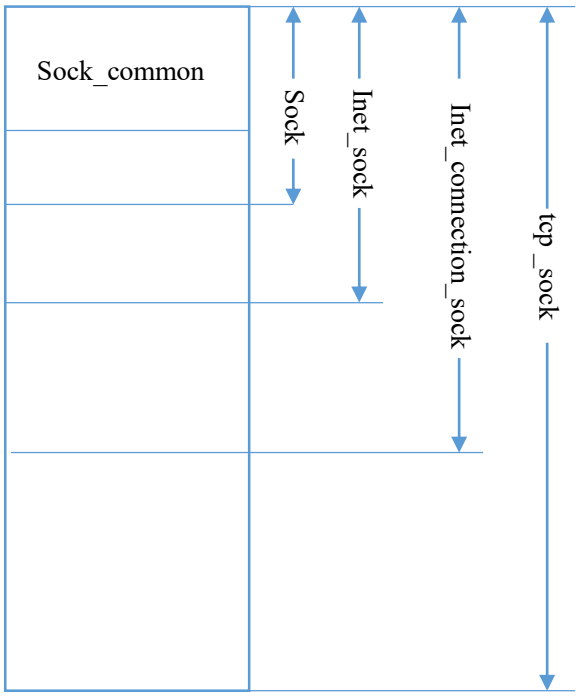
各个传输控制块概述

IPv4 协议族中的传输控制块 socket 结构类型包括：sock_common, sock, inet_sock, inet_connection_sock, tcp_sock, udp_sock, raw_sock, request_sock, inet_request_sock, tcp_request_sock, inet_timewait_sock, tcp_timewait_sock。

- Sock_common: 是网络层对 socket 的最简表示，是由 sock 和 inet_timewait_sock 两个结构体前面的相同部分构成，因此只出现在这两个结构体中。
- Sock:是通用的传输控制块，和具体的协议无关，是各个协议族传输层协议的公共信息，因此不能直接作为传输层控制块使用，不同协议族的传输层协议会对其进行拓展。比如 Ipv4 协议族的 inet_sock 就是在 sock 的基础上进行拓展的。
- Inet_sock:是 Ipv4 协议族传输层的基础，是 TCP、UDP、RAW 协议传输控制块的共同信息（比如本地和外地的 IP 地址、端口、IP 头部等）。
- Inet_connection_sock:是 ipv4 面向连接的传输控制块，是 TCP 协议控制块的基础，在 inet_sock 上加上了有关连接的特性，比如重传计时器、拥塞控制等。
- Tcp_sock:是 TCP 协议传输控制块，包含了每条连接的属性（两个方向上的序列号、接收窗口、发送窗口等）。
- Inet_timewait_sock: 是支持面向连接的 TCP_TIME_WAIT 状态的描述，是 tcp_timewait_sock 的基础。
- Udp_sock:是 UDP 协议传输控制块，所需要的信息基本都在 inet_sock 中了。

tcp_sock 结构示意图：



Sock

```
struct sock {
    /*
     * Now struct inet_timewait_sock also uses sock_common, so please just
     * don't add nothing before this first member (__sk_common) --acme
     */
    struct sock_common    __sk_common;
#define sk_family        __sk_common.skc_family
#define sk_state         __sk_common.skc_state
#define sk_reuse         __sk_common.skc_reuse
#define sk_bound_dev_if          __sk_common.skc_bound_dev_if
#define sk_node          __sk_common.skc_node
#define sk_bind_node         __sk_common.skc_bind_node
#define sk_refcnt        __sk_common.skc_refcnt
#define sk_hash          __sk_common.skc_hash
#define sk_prot          __sk_common.skc_prot
    unsigned char     sk_shutdown : 2,/*占用 2bit，表示关闭读写那个通道，取值为 RCV_SHUTDOWN 关闭接收, SEND_SHUTDOWN 关闭发送, SHUTDOWN_MASK 都关闭*/
                      sk_no_check : 2,
                      sk_userlocks : 4;    /*用户是否通过 setsockopt 设置了 socket 选项，可以取值为 SOCK_SNDBUF_LOCK 设置了发送缓冲区大小，SOCK_RCVBUF_LOCK 设置接收缓冲区大小, SOCK_BINDADDR_LOCK 绑定了本地地址, SOCK_BIND_PORT_LOCK 绑定了本地端口*/
    unsigned char     sk_protocol;
    unsigned short    sk_type;    /*socket 类型，是 SOCK_STREAM, SOCK_DGRAM, SOCK_RAW*/
    int               sk_rcvbuf;
    socket_lock_t     sk_lock;    /*同步锁*/
    wait_queue_head_t *sk_sleep;    /*进程等待队列*/
    struct dst_entry  *sk_dst_cache;    /*目的路由项缓存，在发送报文时，如果没有在这里找对对应的项，才会去找路由表或路由缓存*/
    struct xfrm_policy  *sk_policy[2];
    rwlock_t          sk_dst_lock;    /*目的路由缓存的读写锁*/
```

```
        atomic_t        sk_rmem_alloc;      /*接收队列 sk_receive_queue 中所有 skb 的总长度
*/
        atomic_t        sk_wmem_alloc;     /*所有发送的 skb 的总长度*/
        atomic_t        sk_omem_alloc;
        struct sk_buff_head  sk_receive_queue;     /*接收队列，等待用户进程读取*/
        struct sk_buff_head  sk_write_queue;      /*发送队列，包括待发送数据和未确认数据，
sk_send_head 之后是待发送数据，之前是已发送未确认的数据*/
        struct sk_buff_head sk_async_wait_queue;
        int             sk_wmem_queued;     /*发送队列的总长度*/
        int             sk_forward_alloc;
        gfp_t               sk_allocation;   /*内存分配分方式*/
        int             sk_sndbuf;      /*发送缓冲区的上限，发送队列的总长度
(sk_wmem_alloc)应该小于该值*/
        int             sk_route_caps;
        int             sk_gso_type;
        int             sk_rcvlowat;   /*接收缓存区的下限*/
        unsigned long           sk_flags;
        unsigned long           sk_lingertime; /*关闭套接口前发送剩余数据的时间*/
        /*
          * The backlog queue is special, it is always used with
          * the per-socket spinlock held and requires low latency
          * access. Therefore we special case it's implementation.
          */
        struct {
            struct sk_buff *head;
            struct sk_buff *tail;
        } sk_backlog;
    /* 后备接收队列，当 sock 被上锁时，如果有报文传递到 sock，只能把报文放到
sk_backlog 中，只有用户进程读取 tcp 的数据时，再从该队列复制到用户空间*/
        struct sk_buff_head sk_error_queue;
        struct proto            *sk_prot_creator;
        rwlock_t        sk_callback_lock;
        int             sk_err,
                        sk_err_soft;
        unsigned short      sk_ack_backlog; //当前已建立的连接数
```

```
    unsigned short      sk_max_ack_backlog;   /*最多可以建立多少个连接*/
    __u32               sk_priority;   /*数据包的 Qos 级别*/
    struct ucred        sk_peercred;
    long            sk_rcvtimeo;   /*套接口层接收超时时间*/
    long            sk_sndtimeo;   /*套接口层发送超时时间*/

    struct sk_filter            *sk_filter;
    void            *sk_protinfo;
    struct timer_list   sk_timer; /*配合 TCP 状态，来实现连接定时器、FIN_WAIT_2 定时器，keepalive 定时器*/
    struct timeval      sk_stamp;    /* 未启用 SOCK_RCVTSTAMP 选项时，记录接收报文的时间；如果启用了该选项，在 SKB 的 tstamp 中记录*/
    struct socket       *sk_socket;
    void            *sk_user_data;
    struct page         *sk_sndmsg_page;   /*最近一次分配 skb 的页面*/
    struct sk_buff      *sk_send_head;   /*发送队列 sk_write_queue 中下一个待发送的数据，如果为 NULL 表示没有数据可发了，发送队列上全是已发送未确认的 skb*/
    __u32               sk_sndmsg_off;    /*数据末尾在 sk_sndmsg_page 中的偏移，这两个变量结合起来就可以尝试直接在这个页上追加新的 skb，不过放不下，就分配新的页面，往新页面复制数据，然后跟新 sk_sndmgs_page 和 sk_sndmsg_off，在 tcp_sndmsg()中用到*/
    int             sk_write_pending;    /*有数据等待写入套接口*/
    void            *sk_security;
    void            (*sk_state_change)(struct sock *sk);    /*当 sock 的状态发生变化时，唤醒等待的进程 */
    void            (*sk_data_ready)(struct sock *sk, int bytes);    /*当有数据到达时，唤醒或者发信号给那些等待读数据的进程 */
    void            (*sk_write_space)(struct sock *sk);
    void            (*sk_error_report)(struct sock *sk);
    int             (*sk_backlog_rcv)(struct sock *sk,
                                      struct sk_buff *skb);
    void                (*sk_destruct)(struct sock *sk);       /* 销毁这个 sock，只有 sock 的引用计数为 0，才会真正释放。在 sk_free()中调用，ipv4 对应的函数是 inet_sock_destruct */
};
```

Inet_sock

```
struct inet_sock {
    /* sk and pinet6 has to be the first two members of inet_sock */
    struct sock         sk;
#if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
    struct ipv6_pinfo   *pinet6; / *如果支持 Ipv6，是指向 ipv6 传输控制块的指针*/
#endif
    /* Socket demultiplex comparisons on incoming packets. */
    __be32              daddr; /*目的 IP 地址*/
    __be32              rcv_saddr; /* 源 IP 地址，接收数据时使用*/
    __be16              dport;  /*目的端口*/
    __u16               num;  /* 主机字节序存储的本地端口*/
    __be32              saddr;  /* 源 IP 地址，发送数据时使用*/
    __s16               uc_ttl;  /*单播 TTL*/
    __u16               cmsg_flags;
    struct ip_options   *opt;
    __be16              sport; /*网络字节序存储的本地端口*/
    __u16               id;   /*给 IP 头部的 id 域*/
    __u8                tos;  /*给 IP 头部的 tos 域*/
    __u8                mc_ttl;  /*多播 ttl*/
    __u8                pmtudisc;  /*是否启用路径 mtu 发现，如果不启用，允许分片；
启用，不分片*/
    __u8                recverr:1,
                        is_icsk:1, /*是否为 inet_connection_sock*/
                        freebind:1,
                        hdrincl:1,
                        mc_loop:1;  /*多播是否发向回路*/
    int                 mc_index;
    __be32              mc_addr; /*组播源地址*/
    struct ip_mc_socklist   *mc_list;
    struct {
        unsigned int        flags;
        unsigned int        fragsize;
        struct ip_options   *opt;
```

```
        struct rtable       *rt;
        int                 length; /* Total length of all frames */
        __be32              addr;
        struct flowi        fl;
    } cork;
};
```