TCP 高级实验

王鹿鸣, 刘保证

2016 年 5 月 29 日

目录

CHAPTER 1

准备部分

Contents

1.1 用户层 TCP

用户层的 TCP 编程模型大致如下，对于服务端，调用 listen 监听端口，之后接受客户端的请求，然后就可以收发数据了。结束时，关闭 socket。

```
// Server
socket(...,SOCK_STREAM,0);
bind(...,&server_address, ...);
listen(...);
accept(..., &client_address, ...);
recv(..., &clientaddr, ...);
close(...);
```

对于客户端，则调用 connect 连接服务端，之后便可以收发数据。最后关闭 socket。

```
socket(...,SOCK_STREAM,0);
connect();
send(...,&server_address,...);
```

那么根据我们的需求，我们着重照顾连接的建立、关闭和封包的收发过程。

1.2 探寻 tcp_prot，地图 get~

一般游戏的主角手中，都会有一张万能的地图。为了搞定 TCP，我们自然也是需要一张地图的，要不连该去找那个函数看都不知道。很有幸，在`tcp_ipv4.c`中，`tcp_prot`定义了`tcp`的各个接口。

tcp_prot的类型为struct proto，是这个结构体是为了抽象各种不同的协议的差异性而存在的。类似面向对象中所说的接口 (Interface) 的概念。这里，我们仅保留我们关系的部分。

```
struct proto tcp_prot = {
        .name                   = "TCP",
        .owner                  = THIS_MODULE,
        .close                  = tcp_close,
        .connect                = tcp_v4_connect,
        .disconnect             = tcp_disconnect,
        .accept                 = inet_csk_accept,
        .destroy                = tcp_v4_destroy_sock,
        .shutdown               = tcp_shutdown,
        .setsockopt             = tcp_setsockopt,
        .getsockopt             = tcp_getsockopt,
        .recvmsg                = tcp_recvmsg,
        .sendmsg                = tcp_sendmsg,
        .sendpage               = tcp_sendpage,
        .backlog_rcv            = tcp_v4_do_rcv,
        .get_port               = inet_csk_get_port,
        .twsk_prot              = &tcp_timewait_sock_ops,
        .rsk_prot               = &tcp_request_sock_ops,
};
```

通过名字，我大致筛选出来了这些函数，初步判断这些函数与实验所关心的功能相关。对着这张“地图”，就可以顺藤摸瓜，找出些路径了。

先根据参考书《Linux 内核源码剖析——TCP/IP 实现》中给出的流程图，找出所有和需求相关的部分。

首先找三次握手相关的部分: 从客户端的角度, 发起连接需要调用tcp_v4_connect，该函数会进一步调用tcp_connect，在这个函数中，会调用tcp_send_syn_data 发送 SYN 报文, 并设定超时计时器。第二次握手相关的接收代码在tcp_rcv_state_process中，该函数实现了除ESTABLISHED和TIME_WAIT之外所有状态下的接收处理。tcp_send_ack函数实现了发送 ACK 报文。从服务端的角度，则还需实现listen调用和 accept调用。二者都是服务端建立连接所需要的部分。

封包的封装发送部分，所对应的函数是tcp_sendmsg，实现对数据的复制、切割和发送。TCP 的重传接口为tcp_retransmit_skb，这里尚有疑问，因为这个函数是负责处理重传的，而不是判断是否应当重传的。所以并不明确到底是否该重新实现这一部分。

TCP 封包的接收在tcp_rcv_established函数中，根据目前有限的资料看，TCP 的滑动窗口机制应该在这一部分，更细节的内容待确认。

目前，待重新实现的函数列表是:

- tcp_transmit_skb
- tcp_rcv_state_process
- tcp_connect
- tcp_rcv_synsent_state_process
- tcp_rcv_established

- tcp_send_ack
- tcp_sendmsg
- tcp_retransmit_skb（存疑）
- tcp_rcv_established
- accept 和 listen（待详细调查）
- 更多需要等进一步仔细阅读后再做决定

CHAPTER 2

网络子系统相关核心数据结构

Contents

2.1 网络子系统数据结构架构

2.2 sock 底层数据结构

2.2.1 sock_common

2.2.2 sock

sock 结构是比较通用的网络层描述块，构成传输控制块的基础，与具体的协议族无关。它描述了各协议族的公共信息，因此不能直接作为传输层控制块来使用。不同协议族的传输层在使用该结构的时候都会对其进行拓展，来适合各自的传输特性，例如，inet_sock 结构由 sock 结构及其它特性组成，构成了 IPV4 协议族传输控制块的基础。结构如下：

```
/**
  * struct sock - network layer representation of sockets
  * @__sk_common: shared layout with inet_timewait_sock
  * @sk_shutdown: mask of %SEND_SHUTDOWN and/or %RCV_SHUTDOWN
  * @sk_userlocks: %SO_SNDBUF and %SO_RCVBUF settings
  * @sk_lock:   synchronizer
  * @sk_rcvbuf: size of receive buffer in bytes
  * @sk_wq: sock wait queue and async head
  * @sk_rx_dst: receive input route used by early demux
  * @sk_dst_cache: destination cache
  * @sk_policy: flow policy
  * @sk_receive_queue: incoming packets
  * @sk_wmem_alloc: transmit queue bytes committed
  * @sk_write_queue: Packet sending queue
  * @sk_omem_alloc: "o" is "option" or "other"
  * @sk_wmem_queued: persistent queue size
  * @sk_forward_alloc: space allocated forward
  * @sk_napi_id: id of the last napi context to receive data for sk
  * @sk_ll_usec: usecs to busypoll when there is no data
  * @sk_allocation: allocation mode
  * @sk_pacing_rate: Pacing rate (if supported by transport/packet scheduler)
  * @sk_max_pacing_rate: Maximum pacing rate (%SO_MAX_PACING_RATE)
  * @sk_sndbuf: size of send buffer in bytes
  * @sk_no_check_tx: %SO_NO_CHECK setting, set checksum in TX packets
  * @sk_no_check_rx: allow zero checksum in RX packets
  * @sk_route_caps: route capabilities (e.g. %NETIF_F_TSO)
  * @sk_route_nocaps: forbidden route capabilities (e.g NETIF_F_GSO_MASK)
  * @sk_gso_type: GSO type (e.g. %SKB_GSO_TCPV4)
  * @sk_gso_max_size: Maximum GSO segment size to build
  * @sk_gso_max_segs: Maximum number of GSO segments
  * @sk_lingertime: %SO_LINGER l_linger setting
  * @sk_backlog: always used with the per-socket spinlock held
  * @sk_callback_lock: used with the callbacks in the end of this struct
  * @sk_error_queue: rarely used
  * @sk_prot_creator: sk_prot of original sock creator (see ipv6_setsockopt,
  *           IPV6_ADDRFORM for instance)
  * @sk_err: last error
  * @sk_err_soft: errors that don't cause failure but are the cause of a
  *           persistent failure not just 'timed out'
  * @sk_drops: raw/udp drops counter
  * @sk_ack_backlog: current listen backlog
```

```
  * @sk_max_ack_backlog: listen backlog set in listen()
  * @sk_priority: %SO_PRIORITY setting
  * @sk_cgrp_prioidx: socket group's priority map index
  * @sk_type: socket type (%SOCK_STREAM, etc)
  * @sk_protocol: which protocol this socket belongs in this network family
  * @sk_peer_pid: &struct pid for this socket's peer
  * @sk_peer_cred: %SO_PEERCRED setting
  * @sk_rcvlowat: %SO_RCVLOWAT setting
  * @sk_rcvtimeo: %SO_RCVTIMEO setting
  * @sk_sndtimeo: %SO_SNDTIMEO setting
  * @sk_txhash: computed flow hash for use on transmit
  * @sk_filter: socket filtering instructions
  * @sk_timer: sock cleanup timer
  * @sk_stamp: time stamp of last packet received
  * @sk_tsflags: SO_TIMESTAMPING socket options
  * @sk_tskey: counter to disambiguate concurrent tstamp requests
  * @sk_socket: Identd and reporting IO signals
  * @sk_user_data: RPC layer private data
  * @sk_frag: cached page frag
  * @sk_peek_off: current peek_offset value
  * @sk_send_head: front of stuff to transmit
  * @sk_security: used by security modules
  * @sk_mark: generic packet mark
  * @sk_classid: this socket's cgroup classid
  * @sk_cgrp: this socket's cgroup-specific proto data
  * @sk_write_pending: a write to stream socket waits to start
  * @sk_state_change: callback to indicate change in the state of the sock
  * @sk_data_ready: callback to indicate there is data to be processed
  * @sk_write_space: callback to indicate there is bf sending space available
  * @sk_error_report: callback to indicate errors (e.g. %MSG_ERRQUEUE)
  * @sk_backlog_rcv: callback to process the backlog
  * @sk_destruct: called at sock freeing time, i.e. when all refcnt == 0
 */
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_node         __sk_common.skc_node
#define sk_nulls_node       __sk_common.skc_nulls_node
#define sk_refcnt       __sk_common.skc_refcnt
#define sk_tx_queue_mapping __sk_common.skc_tx_queue_mapping

#define sk_dontcopy_begin   __sk_common.skc_dontcopy_begin
#define sk_dontcopy_end     __sk_common.skc_dontcopy_end
#define sk_hash         __sk_common.skc_hash
#define sk_portpair     __sk_common.skc_portpair
#define sk_num          __sk_common.skc_num
#define sk_dport        __sk_common.skc_dport
#define sk_addrpair     __sk_common.skc_addrpair
#define sk_daddr        __sk_common.skc_daddr
#define sk_rcv_saddr        __sk_common.skc_rcv_saddr
#define sk_family       __sk_common.skc_family
#define sk_state        __sk_common.skc_state
#define sk_reuse        __sk_common.skc_reuse
#define sk_reuseport        __sk_common.skc_reuseport
#define sk_ipv6only     __sk_common.skc_ipv6only
```

```
100 #define sk_net_refcnt       __sk_common.skc_net_refcnt
101 #define sk_bound_dev_if     __sk_common.skc_bound_dev_if
102 #define sk_bind_node        __sk_common.skc_bind_node
103 #define sk_prot         __sk_common.skc_prot
104 #define sk_net          __sk_common.skc_net
105 #define sk_v6_daddr     __sk_common.skc_v6_daddr
106 #define sk_v6_rcv_saddr __sk_common.skc_v6_rcv_saddr
107 #define sk_cookie       __sk_common.skc_cookie
108 #define sk_incoming_cpu     __sk_common.skc_incoming_cpu
109 #define sk_flags        __sk_common.skc_flags
110 #define sk_rxhash       __sk_common.skc_rxhash
111
112     socket_lock_t       sk_lock;
113     struct sk_buff_head sk_receive_queue;
114     /*
115      * The backlog queue is special, it is always used with
116      * the per-socket spinlock held and requires low latency
117      * access. Therefore we special case it's implementation.
118      * Note : rmem_alloc is in this structure to fill a hole
119      * on 64bit arches, not because its logically part of
120      * backlog.
121      */
122     struct {
123         atomic_t    rmem_alloc;
124         int     len;
125         struct sk_buff  *head;
126         struct sk_buff  *tail;
127     } sk_backlog;
128 #define sk_rmem_alloc sk_backlog.rmem_alloc
129     int         sk_forward_alloc;
130
131     __u32           sk_txhash;
132 #ifdef CONFIG_NET_RX_BUSY_POLL
133     unsigned int        sk_napi_id;
134     unsigned int        sk_ll_usec;
135 #endif
136     atomic_t        sk_drops;
137     int         sk_rcvbuf;
138
139     struct sk_filter __rcu  *sk_filter;
140     union {
141         struct socket_wq __rcu  *sk_wq;
142         struct socket_wq    *sk_wq_raw;
143     };
144 #ifdef CONFIG_XFRM
145     struct xfrm_policy __rcu *sk_policy[2];
146 #endif
147     struct dst_entry    *sk_rx_dst;
148     struct dst_entry __rcu  *sk_dst_cache;
149     /* Note: 32bit hole on 64bit arches */
150     atomic_t        sk_wmem_alloc;
151     atomic_t        sk_omem_alloc;
152     int         sk_sndbuf;
153     struct sk_buff_head sk_write_queue;
154     kmemcheck_bitfield_begin(flags);
155     unsigned int        sk_shutdown  : 2,
156                 sk_no_check_tx : 1,
157                 sk_no_check_rx : 1,
158                 sk_userlocks : 4,
```

```
                sk_protocol   : 8,
                sk_type       : 16;
#define SK_PROTOCOL_MAX U8_MAX
    kmemcheck_bitfield_end(flags);
    int         sk_wmem_queued;
    gfp_t           sk_allocation;
    u32         sk_pacing_rate; /* bytes per second */
    u32         sk_max_pacing_rate;
    netdev_features_t   sk_route_caps;
    netdev_features_t   sk_route_nocaps;
    int         sk_gso_type;
    unsigned int        sk_gso_max_size;
    u16         sk_gso_max_segs;
    int         sk_rcvlowat;
    unsigned long           sk_lingertime;
    struct sk_buff_head sk_error_queue;
    struct proto        *sk_prot_creator;
    rwlock_t        sk_callback_lock;
    int         sk_err,
                sk_err_soft;
    u32         sk_ack_backlog;
    u32         sk_max_ack_backlog;
    __u32           sk_priority;
#if IS_ENABLED(CONFIG_CGROUP_NET_PRIO)
    __u32           sk_cgrp_prioidx;
#endif
    struct pid      *sk_peer_pid;
    const struct cred   *sk_peer_cred;
    long            sk_rcvtimeo;
    long            sk_sndtimeo;
    struct timer_list   sk_timer;
    ktime_t         sk_stamp;
    u16         sk_tsflags;
    u32         sk_tskey;
    struct socket       *sk_socket;
    void            *sk_user_data;
    struct page_frag    sk_frag;
    struct sk_buff      *sk_send_head;
    __s32           sk_peek_off;
    int         sk_write_pending;
#ifdef CONFIG_SECURITY
    void            *sk_security;
#endif
    __u32           sk_mark;
#ifdef CONFIG_CGROUP_NET_CLASSID
    u32         sk_classid;
#endif
    struct cg_proto     *sk_cgrp;
    void            (*sk_state_change)(struct sock *sk);
    void            (*sk_data_ready)(struct sock *sk);
    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);
};
```

2.2.3 request_sock

该结构位于/include/net/request_sock.h中。该结构用于表示一个简单的 TCP 连接请求。

```
/* struct request_sock - mini sock to represent a connection request
 */
struct request_sock {
        struct sock_common              __req_common;
#define rsk_refcnt                      __req_common.skc_refcnt
#define rsk_hash                        __req_common.skc_hash
#define rsk_listener                    __req_common.skc_listener
#define rsk_window_clamp                __req_common.skc_window_clamp
#define rsk_rcv_wnd                     __req_common.skc_rcv_wnd

        struct request_sock             *dl_next;
        u16                             mss;
        u8                              num_retrans; /* number of retransmits */
        u8                              cookie_ts:1; /* syncookie: encode tcpopts in timestamp */
        u8                              num_timeout:7; /* number of timeouts */
        u32                             ts_recent;
        struct timer_list               rsk_timer;
        const struct request_sock_ops   *rsk_ops;
        struct sock                     *sk;
        u32                             *saved_syn;
        u32                             secid;
        u32                             peer_secid;
};
```

2.2.4 sk_buff

struct sk_buff这一结构体在各层协议中都会被用到。该结构体存储了网络数据报的所有信息。包括各层的头部以及 payload，以及必要的各层实现相关的信息。

该结构体的定义较长，需要一点一点分析。结构体的开头为

```
  union {
    struct {
      /* These two members must be first. */
      struct sk_buff          *next;
      struct sk_buff          *prev;

      union {
        ktime_t         tstamp;
        struct skb_mstamp skb_mstamp;
      };
    };
    struct rb_node  rbnode; /* used in netem and tcp stack */
  };
```

可以看到，sk_buff可以被组织成两种数据结构：双向链表和红黑树。且一个sk_buff不是在双向链表中，就是在红黑树中，因此，采用了 union 来节约空间。next 和 prev 两个域是用于双向链表的结构体，而 rbnode 是红黑树相关的结构。

包的到达/发送时间存放在union {ktime_t tstamp;struct skb_mstamp skb_mstamp;};中，之所以这里有两种不同的时间戳类型，是因为有时候调用ktime_get()的成本太高。因

此,内核开发者希望能够在 TCP 协议栈中实现一个轻量级的微秒级的时间戳。struct skb_mstamp正是结合了local_clock()和 jiffies二者，而实现的一个轻量级的工具。当然，根据内核邮件列表中的说法，并不是任何时候都可以用该工具替换调ktime_get()的。因此，在struct sk_buff结构体中，采用union的方式同时保留了这二者。

在定义完数据结构相关的一些部分后，又定义了如下的结构体

```
  /* 拥有该 sk_buff 的套接字的指针 */
  struct sock             *sk;
  /* 与该包关联的网络设备 */
  struct net_device       *dev;
  /* 控制用的缓冲区，用于存放各层的私有数据 */
  char                    cb[48] __aligned(8);
  /* 存放了目的地项的引用计数 */
  unsigned long           _skb_refdst;
  /* 析构函数 */
  void                    (*destructor)(struct sk_buff *skb);
#ifdef CONFIG_XFRM
  /* xfrm 加密通道 */
  struct  sec_path        *sp;
#endif
#if IS_ENABLED(CONFIG_BRIDGE_NETFILTER)
  /* 保存和 bridge 相关的信息 */
  struct nf_bridge_info   *nf_bridge;
#endif
```

其中的char cb[48]比较有意思，各层都使用这个 buffer 来存放自己私有的变量。这里值得注意的是，如果想要跨层传递数据，则需要使用 skb_clone()。XFRM 则是 Linux 在 2.6 版本中引入的一个安全方面的扩展。

之后，又定义了一些长度相关的字段。len代表 buffer 中的数据长度（含各协议的头部），以及分片长度。而len代表分片中的数据的长度。mac_len是 MAC 层头部的长度。hdr_len是一个克隆出来的可写的头部的长度。

```
unsigned int            len,
                        data_len;
__u16                   mac_len,
                        hdr_len;
```

kmemcheck 是内核中的一套内存检测工具。kmemcheck_bitfield_begin 和kmemcheck_bitfield_begi以用于说明一段内容的起始和终止位置。其代码定义如下：

```
#define kmemcheck_bitfield_begin(name)  \
        int name##_begin[0];

#define kmemcheck_bitfield_end(name)    \
        int name##_end[0];
```

通过定义，我们不难看出，这两个宏是用于在代码中产生两个对应于位域的起始地址和终止地址的符号的。当然，这两个宏是为 kmemcheck 的功能服务的。如果没有开启该功能的话，这两个宏的定义为空，也即不会产生任何作用。

```
/* Following fields are _not_ copied in __copy_skb_header()
* Note that queue_mapping is here mostly to fill a hole.
*/
kmemcheck_bitfield_begin(flags1);
__u16                   queue_mapping; /* 对于多队列设备的队列关系映射 */
__u8                    cloned:1, /* 是否被克隆 */
                        nohdr:1, /* 只引用了负载 */
                        fclone:2, /* skbuff 克隆的情况 */
                        /* peeked 表明该包已经被统计过了，无需再次统计 */
                        peeked:1,
                        head_frag:1,
                        xmit_more:1; /* 在队列中有更多的 SKB 在等待 */
/* one bit hole */
kmemcheck_bitfield_end(flags1);
```

在这段定义中，内核将一系列的标志位命名为了 flags1，利用那两个函数可以在生成的代码中插入 `flags1_begin`和`flags1_end`两个符号。这样，当有需要的时候，可以通过这两个符号找到这一段的起始地址和结束地址。

紧接着是一个包的头部，这一部分再次使用了类似上面的方法，用了两个零长度的数组 `headers_start`和`headers_end`来标明头部的起始和终止地址。

```
        /* 在 __copy_skb_header() 中，只需使用一个 memcpy() 即可将 headers_start/end
         * 之间的部分克隆一份。
         */
        /* private: */
        __u32                   headers_start[0];
        /* public: */

/* if you move pkt_type around you also must adapt those constants */
#ifdef __BIG_ENDIAN_BITFIELD
#define PKT_TYPE_MAX    (7 << 5)
#else
#define PKT_TYPE_MAX    7
#endif
#define PKT_TYPE_OFFSET()       offsetof(struct sk_buff, __pkt_type_offset)

        __u8                    __pkt_type_offset[0];
        /* 该包的类型 */
        __u8                    pkt_type:3;
        __u8                    pfmemalloc:1;
        /* 是否允许本地分片 (local fragmentation) */
        __u8                    ignore_df:1;
        /* 表明该 skb 和连接的关系 */
        __u8                    nfctinfo:3;
        /* netfilter 包追踪标记 */
        __u8                    nf_trace:1;
        /* 驱动（硬件）给出来的 checksum */
        __u8                    ip_summed:2;
        /* 允许该 socket 到队列的对应关系发生变更 */
        __u8                    ooo_okay:1;
        /* 表明哈希值字段 hash 是一个典型的 4 元组的通过传输端口的哈希 */
        __u8                    l4_hash:1;
        /* 表明哈希值字段 hash 是通过软件栈计算出来的 */
        __u8                    sw_hash:1;
        /* 表明 wifi_acked 是否被设置了 */
        __u8                    wifi_acked_valid:1;
```

```
36          /* 表明帧是否在 wifi 上被确认了 */
37          __u8                    wifi_acked:1;
38
39          /* 请求 NIC 将最后的 4 个字节作为以太网 FCS 来对待 */
40          __u8                    no_fcs:1;
41          /* Indicates the inner headers are valid in the skbuff. */
42          __u8                    encapsulation:1;
43          __u8                    encap_hdr_csum:1;
44          __u8                    csum_valid:1;
45          __u8                    csum_complete_sw:1;
46          __u8                    csum_level:2;
47          __u8                    csum_bad:1;
48
49  #ifdef CONFIG_IPV6_NDISC_NODETYPE
50          __u8                    ndisc_nodetype:2; /* 路由类型（来自链路层） */
51  #endif
52          /* 标明该 skbuff 是否被 ipvs 拥有 */
53          __u8                    ipvs_property:1;
54          __u8                    inner_protocol_type:1;
55          __u8                    remcsum_offload:1;
56          /* 3 or 5 bit hole */
57
58  #ifdef CONFIG_NET_SCHED
59          __u16                   tc_index;       /* traffic control index */
60  #ifdef CONFIG_NET_CLS_ACT
61          __u16                   tc_verd;        /* traffic control verdict */
62  #endif
63  #endif
64
65          union {
66                  __wsum          csum; /* 校验码 */
67                  struct {
68                          /* 从 skb->head 开始到应当计算校验码的起始位置的偏移 */
69                          __u16   csum_start;
70                          /* 从 csum_start 开始到存储校验码的位置的偏移 */
71                          __u16   csum_offset;
72                  };
73          };
74          __u32                   priority; /* 包队列的优先级 */
75          int                     skb_iif; /* 到达的设备的序号 */
76          __u32                   hash; /* 包的哈希值 */
77          __be16                  vlan_proto; /* vlan 包装协议 */
78          __u16                   vlan_tci; /* vlan tag 控制信息 */
79  #if defined(CONFIG_NET_RX_BUSY_POLL) || defined(CONFIG_XPS)
80          union {
81                  unsigned int    napi_id; /* 表明该 skb 来源的 NAPI 结构体的 id */
82                  unsigned int    sender_cpu;
83          };
84  #endif
85          union {
86  #ifdef CONFIG_NETWORK_SECMARK
87                  __u32           secmark; /* 安全标记 */
88  #endif
89  #ifdef CONFIG_NET_SWITCHDEV
90                  __u32           offload_fwd_mark; /* fwding offload mark */
91  #endif
92          };
93
94          union {
```

```
95                  __u32           mark; /* 通用的包的标记位 */
96                  __u32           reserved_tailroom;
97          };
98
99          union {
100                 __be16          inner_protocol; /* 协议（封装好的） */
101                 __u8            inner_ipproto;
102         };
103
104         /* 已封装的内部传输层头部 */
105         __u16                   inner_transport_header;
106         /* 已封装的内部网络层头部 */
107         __u16                   inner_network_header;
108         /* 已封装的内部链路层头部 */
109         __u16                   inner_mac_header;
110
111         /* 驱动（硬件）给出的包的协议类型 */
112         __be16                  protocol;
113         /* 传输层头部 */
114         __u16                   transport_header;
115         /* 网络层头部 */
116         __u16                   network_header;
117         /* 数据链路层头部 */
118         __u16                   mac_header;
119
120         /* private: */
121         __u32                   headers_end[0];
```

最后是一组是管理相关的字段。其中，head和end 代表被分配的内存的起始位置和终止位置。而data和tail 则是实际数据的起始和终止位置。

```
1   /* These elements must be at the end, see alloc_skb() for details.  */
2           sk_buff_data_t          tail;
3           sk_buff_data_t          end;
4           unsigned char           *head,
5                                   *data;
6           unsigned int            truesize;
7           atomic_t                users;
```

users是引用计数，所以是个原子的。truesize是数据报的真实大小。

2.3 inet 层相关数据结构

inet_request_sock

2.3.1 inet_connection_sock_af_ops

该结构位于/include/net/inet_connect_sock.h中, 其后面的 af 表示 address of function 即函数地址, ops 表示 operations，即操作。

该结构封装了一组与传输层有关的操作集,包括向网络层发送的接口、传输层的setsockopt接口等。

```
struct inet_connection_sock_af_ops {
    int     (*queue_xmit)(struct sock *sk, struct sk_buff *skb, struct flowi *fl);
    void        (*send_check)(struct sock *sk, struct sk_buff *skb);
    int     (*rebuild_header)(struct sock *sk);
    void        (*sk_rx_dst_set)(struct sock *sk, const struct sk_buff *skb);
    int     (*conn_request)(struct sock *sk, struct sk_buff *skb);
    struct sock *(*syn_recv_sock)(const struct sock *sk, struct sk_buff *skb,
                      struct request_sock *req,
                      struct dst_entry *dst,
                      struct request_sock *req_unhash,
                      bool *own_req);
    u16     net_header_len;
    u16     net_frag_header_len;
    u16     sockaddr_len;
    int     (*setsockopt)(struct sock *sk, int level, int optname,
                  char __user *optval, unsigned int optlen);
    int     (*getsockopt)(struct sock *sk, int level, int optname,
                  char __user *optval, int __user *optlen);
#ifdef CONFIG_COMPAT
    int     (*compat_setsockopt)(struct sock *sk,
                int level, int optname,
                char __user *optval, unsigned int optlen);
    int     (*compat_getsockopt)(struct sock *sk,
                int level, int optname,
                char __user *optval, int __user *optlen);
#endif
    void        (*addr2sockaddr)(struct sock *sk, struct sockaddr *);
    int     (*bind_conflict)(const struct sock *sk,
                     const struct inet_bind_bucket *tb, bool relax);
    void        (*mtu_reduced)(struct sock *sk);
};
```

2.3.2 inet_connect_sock

该结构位于/include/net/inet_connect_sock.h中，它是所有面向传输控制块的表示。其在inet_sock的基础上，增加了有关连接，确认，重传等成员。

```
/** inet_connection_sock - INET connection oriented sock
 *
 * @icsk_accept_queue:     FIFO of established children
 * @icsk_bind_hash:    Bind node
 * @icsk_timeout:      Timeout
 * @icsk_retransmit_timer: Resend (no ack)
 * @icsk_rto:          Retransmit timeout
 * @icsk_pmtu_cookie       Last pmtu seen by socket
 * @icsk_ca_ops        Pluggable congestion control hook
 * @icsk_af_ops        Operations which are AF_INET{4,6} specific
 * @icsk_ca_state:     Congestion control state
 * @icsk_retransmits:      Number of unrecovered [RTO] timeouts
 * @icsk_pending:      Scheduled timer event
 * @icsk_backoff:      Backoff
 * @icsk_syn_retries:      Number of allowed SYN (or equivalent) retries
 * @icsk_probes_out:       unanswered 0 window probes
 * @icsk_ext_hdr_len:      Network protocol overhead (IP/IPv6 options)
 * @icsk_ack:          Delayed ACK control data
 * @icsk_mtup;         MTU probing control data
 */
```

```
struct inet_connection_sock {
    /* inet_sock has to be the first member! */
    struct inet_sock      icsk_inet;
    struct request_sock_queue icsk_accept_queue;
    struct inet_bind_bucket   *icsk_bind_hash;
    unsigned long         icsk_timeout;
    struct timer_list     icsk_retransmit_timer;
    struct timer_list     icsk_delack_timer;
    __u32             icsk_rto;
    __u32             icsk_pmtu_cookie;
    const struct tcp_congestion_ops *icsk_ca_ops;
    const struct inet_connection_sock_af_ops *icsk_af_ops;
    unsigned int          (*icsk_sync_mss)(struct sock *sk, u32 pmtu);
    __u8              icsk_ca_state:6,
                  icsk_ca_setsockopt:1,
                  icsk_ca_dst_locked:1;
    __u8              icsk_retransmits;
    __u8              icsk_pending;
    __u8              icsk_backoff;
    __u8              icsk_syn_retries;
    __u8              icsk_probes_out;
    __u16             icsk_ext_hdr_len;
    struct {
        __u8          pending;   /* ACK is pending             */
        __u8          quick;     /* Scheduled number of quick acks     */
        __u8          pingpong;  /* The session is interactive         */
        __u8          blocked;   /* Delayed ACK was blocked by socket lock */
        __u32         ato;       /* Predicted tick of soft clock       */
        unsigned long     timeout;   /* Currently scheduled timeout        */
        __u32         lrcvtime;  /* timestamp of last received data packet */
        __u16         last_seg_size; /* Size of last incoming segment      */
        __u16         rcv_mss;   /* MSS used for delayed ACK decisions     */
    } icsk_ack;
    struct {
        int       enabled;

        /* Range of MTUs to search */
        int       search_high;
        int       search_low;

        /* Information on the current probe. */
        int       probe_size;

        u32       probe_timestamp;
    } icsk_mtup;
    u32           icsk_user_timeout;

    u64           icsk_ca_priv[64 / sizeof(u64)];
#define ICSK_CA_PRIV_SIZE      (8 * sizeof(u64))
};
```

2.3.3 sockaddr 和 sockaddr_in

sockaddr用于描述一个地址。

```
/* include/linux/socket.h */
struct sockaddr {
        sa_family_t     sa_family;      /* 地址所属的协议族，AF_xxx       */
```

```
        char            sa_data[14];    /* 在协议下的地址 */
};
```

可以看出sockaddr是一个较为通用的描述方法。可以支持任意的网络层协议。那么具体到我们的情况，就是 IP 网络。下面是 IP 网络下，该结构体的定义。

```
/* include/uapi/linux/in.h
 * 该结构体用于描述一个 Internet (IP) 套接字的地址
 */
struct sockaddr_in {
  __kernel_sa_family_t  sin_family;     /* 这里和 sockaddr 是对应的，填写 IP 网络 */
  __be16                sin_port;       /* 端口号                                 */
  struct in_addr        sin_addr;       /* Internet 地址                          */

  /* 填充位，为了将 sockaddr_in 填充到和 sockaddr 一样长 */
  unsigned char         __pad[__SOCK_SIZE__ - sizeof(short int) -
                        sizeof(unsigned short int) - sizeof(struct in_addr)];
};
```

sockaddr的使用方法是在需要的地方直接强制转型成相应网络的结构体。因此，需要让二者一样大。这就是为何sockaddr_in要加填充位的原因。

2.3.4 ip_options

```
/** struct ip_options - IP Options
 *
 * @faddr - 保存的第一跳地址
 * @nexthop - 保存在 LSRR 和 SSRR 的下一跳地址
 * @is_strictroute - 严格的源路由
 * @srr_is_hit - 包目标地址命中
 * @is_changed - IP 校验和不合法
 * @rr_needaddr - 需要记录出口设备的地址
 * @ts_needtime - 需要记录时间戳
 * @ts_needaddr - 需要记录出口设备的地址
 */
struct ip_options {
        __be32          faddr;
        __be32          nexthop;
        unsigned char   optlen;
        unsigned char   srr;
        unsigned char   rr;
        unsigned char   ts;
        unsigned char   is_strictroute:1,
                        srr_is_hit:1,
                        is_changed:1,
                        rr_needaddr:1,
                        ts_needtime:1,
                        ts_needaddr:1;
        unsigned char   router_alert;
        unsigned char   cipso;
        unsigned char   __pad2;
        unsigned char   __data[0];
};

struct ip_options_rcu {
        struct rcu_head rcu;
        struct ip_options opt;
```

```
};
```

2.4 路由相关数据结构

2.4.1 dst_entry

该结构位于/include/net/dst.h中。

最终生成的 IP 数据报的路由称为目的入口 (dst_entry)，目的入口反映了相邻的外部主机在本地主机内部的一种“映象”。它是与协议无关的目的路由缓存相关的数据结构，保护了路由缓存链接在一起的数据结构成员变量、垃圾回收相关的成员变量、邻居项相关的成员、二层缓存头相关的成员、输入/输出函数指针以用于命中路由缓存的数据包进行后续的数据处理等。

```
/* Each dst_entry has reference count and sits in some parent list(s).
 * When it is removed from parent list, it is "freed" (dst_free).
 * After this it enters dead state (dst->obsolete > 0) and if its refcnt
 * is zero, it can be destroyed immediately, otherwise it is added
 * to gc list and garbage collector periodically checks the refcnt.
 */
struct dst_entry {
        struct rcu_head                 rcu_head;
        struct dst_entry        *child;
        struct net_device       *dev;
        struct  dst_ops                 *ops;
        unsigned long                _metrics;
        unsigned long           expires;
        struct dst_entry        *path;
        struct dst_entry        *from;
#ifdef CONFIG_XFRM
        struct xfrm_state        *xfrm;
#else
        void                        *__pad1;
#endif
        int                        (*input)(struct sk_buff *);
        int                        (*output)(struct net *net, struct sock *sk, struct sk_buff *skb);

        unsigned short                flags;
#define DST_HOST                0x0001
#define DST_NOXFRM               0x0002
#define DST_NOPOLICY              0x0004
#define DST_NOHASH               0x0008
#define DST_NOCACHE               0x0010
#define DST_NOCOUNT               0x0020
#define DST_FAKE_RTABLE                0x0040
#define DST_XFRM_TUNNEL                0x0080
#define DST_XFRM_QUEUE              0x0100
#define DST_METADATA               0x0200

        unsigned short                pending_confirm;

        short                        error;

        /* A non-zero value of dst->obsolete forces by-hand validation
         * of the route entry.  Positive values are set by the generic
         * dst layer to indicate that the entry has been forcefully
```

```
         * destroyed.
         *
         * Negative values are used by the implementation layer code to
         * force invocation of the dst_ops->check() method.
         */
        short                           obsolete;
#define DST_OBSOLETE_NONE       0
#define DST_OBSOLETE_DEAD       2
#define DST_OBSOLETE_FORCE_CHK      -1
#define DST_OBSOLETE_KILL       -2
        unsigned short                  header_len;     /* more space at head required */
        unsigned short                  trailer_len;    /* space to reserve at tail */
#ifdef CONFIG_IP_ROUTE_CLASSID
        __u32                           tclassid;
#else
        __u32                           __pad2;
#endif

#ifdef CONFIG_64BIT
        struct lwtunnel_state   *lwtstate;
        /*
         * Align __refcnt to a 64 bytes alignment
         * (L1_CACHE_SIZE would be too much)
         */
        long                            __pad_to_align_refcnt[1];
#endif
        /*
         * __refcnt wants to be on a different cache line from
         * input/output/ops or performance tanks badly
         */
        atomic_t                __refcnt;       /* client references    */
        int                             __use;
        unsigned long                   lastuse;
#ifndef CONFIG_64BIT
        struct lwtunnel_state   *lwtstate;
#endif
        union {
                struct dst_entry        *next;
                struct rtable __rcu     *rt_next;
                struct rt6_info         *rt6_next;
                struct dn_route __rcu   *dn_next;
        };
};
```

2.4.2 rtable

该结构位于/include/net/route.h中。

这是 ipv4 路由缓存相关结构体,保护了该路由缓存查找的匹配条件,即struct flowi类型的变量、目的 ip、源 ip、下一跳网关地址、路由类型等。当然了，还有最重要的，保护了一个协议无关的dst_entry变量，通过该 union 能够很好的实现dst_entry与 rtable 的转换，而dst_entry中又包含邻居项相关的信息，实现了路由缓存与邻居子系统的关联。

```
struct rtable {
        struct dst_entry        dst;

```

```
        int                     rt_genid;
        unsigned int            rt_flags;
        __u16                   rt_type;
        __u8                    rt_is_input;
        __u8                    rt_uses_gateway;

        int                     rt_iif;

        /* Info on neighbour */
        __be32                  rt_gateway;

        /* Miscellaneous cached information */
        u32                     rt_pmtu;

        u32                     rt_table_id;

        struct list_head        rt_uncached;
        struct uncached_list    *rt_uncached_list;
};
```

2.4.3 flowi

该数据结构位于/include/net/flow.h中，它是与路由查找相关的数据结构。

```
struct flowi {
        union {
                struct flowi_common     __fl_common;
                struct flowi4             ip4;
                struct flowi6             ip6;
                struct flowidn            dn;
        } u;
#define flowi_oif       u.__fl_common.flowic_oif
#define flowi_iif       u.__fl_common.flowic_iif
#define flowi_mark      u.__fl_common.flowic_mark
#define flowi_tos       u.__fl_common.flowic_tos
#define flowi_scope     u.__fl_common.flowic_scope
#define flowi_proto     u.__fl_common.flowic_proto
#define flowi_flags     u.__fl_common.flowic_flags
#define flowi_secid     u.__fl_common.flowic_secid
#define flowi_tun_key   u.__fl_common.flowic_tun_key
} __attribute__((__aligned__(BITS_PER_LONG/8)));
```

2.5 TCP 层相关数据结构

2.5.1 tcphdr

该数据结构位于/include/uapi/linux/tcp.h 中。

```
    struct tcphdr {
        __be16  source;
        __be16  dest;
        __be32  seq;
        __be32  ack_seq;
    #if defined(__LITTLE_ENDIAN_BITFIELD)
        __u16   res1:4,
            doff:4,
```

```
            fin:1,
            syn:1,
            rst:1,
            psh:1,
            ack:1,
            urg:1,
            ece:1,
            cwr:1;
    #elif defined(__BIG_ENDIAN_BITFIELD)
        __u16   doff:4,
            res1:4,
            cwr:1,
            ece:1,
            urg:1,
            ack:1,
            psh:1,
            rst:1,
            syn:1,
            fin:1;
    #else
    #error  "Adjust your <asm/byteorder.h> defines"
    #endif
        __be16  window;
        __sum16 check;
        __be16  urg_ptr;
    };
```

2.5.2 tcp_options_received

该结构位于/include/linux/tcp.h中，其主要表述 TCP 头部的选项字段。

```
struct tcp_options_received {
/*  PAWS/RTTM data  */
    long    ts_recent_stamp;  /* Time we stored ts_recent (for aging)        */
    u32 ts_recent;            /* Time stamp to echo next                     */
    u32 rcv_tsval;            /* Time stamp value                            */
    u32 rcv_tsecr;            /* Time stamp echo reply                       */
    u16     saw_tstamp : 1,   /* Saw TIMESTAMP on last packet                */
        tstamp_ok : 1,        /* TIMESTAMP seen on SYN packet                */
        dsack : 1,            /* D-SACK is scheduled                         */
        wscale_ok : 1,        /* Wscale seen on SYN packet                   */
        sack_ok : 4,          /* SACK seen on SYN packet                     */
        snd_wscale : 4,       /* Window scaling received from sender         */
        rcv_wscale : 4;       /* Window scaling to send to receiver          */
    u8  num_sacks;            /* Number of SACK blocks                       */
    u16 user_mss;             /* mss requested by user in ioctl              */
    u16 mss_clamp;            /* Maximal mss, negotiated at connection setup */
};
```

2.5.3 tcp_sock

该数据结构位于/include/linux/tcp.h中。

该数据结构时 TCP 协议的控制块，它在inet_connection_sock结构的基础上扩展了**滑动窗口协议、拥塞控制算法**等一些 TCP 的专有属性。

```
struct tcp_sock {
    /* inet_connection_sock has to be the first member of tcp_sock */
    struct inet_connection_sock inet_conn;
    u16 tcp_header_len; /* Bytes of tcp header to send      */
    u16 gso_segs;   /* Max number of segs per GSO packet    */

/*
 *  Header prediction flags
 *  0x5?10 << 16 + snd_wnd in net byte order
 */
    __be32  pred_flags;

/*
 *  RFC793 variables by their proper names. This means you can
 *  read the code and the spec side by side (and laugh ...)
 *  See RFC793 and RFC1122. The RFC writes these in capitals.
 */
    u64 bytes_received; /* RFC4898 tcpEStatsAppHCThruOctetsReceived
                * sum(delta(rcv_nxt)), or how many bytes
                * were acked.
                */
    u32 segs_in;    /* RFC4898 tcpEStatsPerfSegsIn
                * total number of segments in.
                */
    u32 rcv_nxt;    /* What we want to receive next     */
    u32 copied_seq; /* Head of yet unread data      */
    u32 rcv_wup;    /* rcv_nxt on last window update sent   */
    u32 snd_nxt;    /* Next sequence we send        */
    u32 segs_out;   /* RFC4898 tcpEStatsPerfSegsOut
                * The total number of segments sent.
                */
    u64 bytes_acked;    /* RFC4898 tcpEStatsAppHCThruOctetsAcked
                * sum(delta(snd_una)), or how many bytes
                * were acked.
                */
    struct u64_stats_sync syncp; /* protects 64bit vars (cf tcp_get_info()) */

    u32 snd_una;    /* First byte we want an ack for    */
    u32 snd_sml;    /* Last byte of the most recently transmitted small packet */
    u32 rcv_tstamp; /* timestamp of last received ACK (for keepalives) */
    u32 lsndtime;   /* timestamp of last sent data packet (for restart window) */
    u32 last_oow_ack_time;  /* timestamp of last out-of-window ACK */

    u32 tsoffset;   /* timestamp offset */

    struct list_head tsq_node; /* anchor in tsq_tasklet.head list */
    unsigned long   tsq_flags;

    /* Data for direct copy to user */
    struct {
        struct sk_buff_head prequeue;
        struct task_struct  *task;
        struct msghdr       *msg;
        int         memory;
        int         len;
    } ucopy;

    u32 snd_wl1;    /* Sequence for window update       */
```

```
59      u32 snd_wnd;    /* The window we expect to receive  */
60      u32 max_window; /* Maximal window ever seen from peer   */
61      u32 mss_cache;  /* Cached effective mss, not including SACKS */
62
63      u32 window_clamp;   /* Maximal window to advertise      */
64      u32 rcv_ssthresh;   /* Current window clamp         */
65
66      /* Information of the most recently (s)acked skb */
67      struct tcp_rack {
68          struct skb_mstamp mstamp; /* (Re)sent time of the skb */
69          u8 advanced; /* mstamp advanced since last lost marking */
70          u8 reord;    /* reordering detected */
71      } rack;
72      u16 advmss;     /* Advertised MSS           */
73      u8  unused;
74      u8  nonagle     : 4,/* Disable Nagle algorithm?             */
75          thin_lto    : 1,/* Use linear timeouts for thin streams */
76          thin_dupack : 1,/* Fast retransmit on first dupack      */
77          repair      : 1,
78          frto        : 1;/* F-RTO (RFC5682) activated in CA_Loss */
79      u8  repair_queue;
80      u8  do_early_retrans:1,/* Enable RFC5827 early-retransmit  */
81          syn_data:1, /* SYN includes data */
82          syn_fastopen:1, /* SYN includes Fast Open option */
83          syn_fastopen_exp:1,/* SYN includes Fast Open exp. option */
84          syn_data_acked:1,/* data in SYN is acked by SYN-ACK */
85          save_syn:1, /* Save headers of SYN packet */
86          is_cwnd_limited:1;/* forward progress limited by snd_cwnd? */
87      u32 tlp_high_seq;   /* snd_nxt at the time of TLP retransmit. */
88
89  /* RTT measurement */
90      u32 srtt_us;    /* smoothed round trip time << 3 in usecs */
91      u32 mdev_us;    /* medium deviation         */
92      u32 mdev_max_us;    /* maximal mdev for the last rtt period */
93      u32 rttvar_us;  /* smoothed mdev_max            */
94      u32 rtt_seq;    /* sequence number to update rttvar */
95      struct rtt_meas {
96          u32 rtt, ts;    /* RTT in usec and sampling time in jiffies. */
97      } rtt_min[3];
98
99      u32 packets_out;    /* Packets which are "in flight"    */
100     u32 retrans_out;    /* Retransmitted packets out        */
101     u32 max_packets_out;  /* max packets_out in last window */
102     u32 max_packets_seq;  /* right edge of max_packets_out flight */
103
104     u16 urg_data;   /* Saved octet of OOB data and control flags */
105     u8  ecn_flags;  /* ECN status bits.         */
106     u8  keepalive_probes; /* num of allowed keep alive probes   */
107     u32 reordering; /* Packet reordering metric.        */
108     u32 snd_up;     /* Urgent pointer       */
109
110 /*
111  *      Options received (usually on last packet, some only on SYN packets).
112  */
113     struct tcp_options_received rx_opt;
114
115 /*
116  *  Slow start and congestion control (see also Nagle, and Karn & Partridge)
117  */
```

```
118     u32 snd_ssthresh;   /* Slow start size threshold        */
119     u32 snd_cwnd;   /* Sending congestion window        */
120     u32 snd_cwnd_cnt;   /* Linear increase counter      */
121     u32 snd_cwnd_clamp; /* Do not allow snd_cwnd to grow above this */
122     u32 snd_cwnd_used;
123     u32 snd_cwnd_stamp;
124     u32 prior_cwnd; /* Congestion window at start of Recovery. */
125     u32 prr_delivered;  /* Number of newly delivered packets to
126                  * receiver in Recovery. */
127     u32 prr_out;    /* Total number of pkts sent during Recovery. */
128
129     u32 rcv_wnd;    /* Current receiver window      */
130     u32 write_seq;  /* Tail(+1) of data held in tcp send buffer */
131     u32 notsent_lowat;  /* TCP_NOTSENT_LOWAT */
132     u32 pushed_seq; /* Last pushed seq, required to talk to windows */
133     u32 lost_out;   /* Lost packets         */
134     u32 sacked_out; /* SACK'd packets           */
135     u32 fackets_out;    /* FACK'd packets           */
136
137     /* from STCP, retrans queue hinting */
138     struct sk_buff* lost_skb_hint;
139     struct sk_buff *retransmit_skb_hint;
140
141     /* OOO segments go in this list. Note that socket lock must be held,
142      * as we do not use sk_buff_head lock.
143      */
144     struct sk_buff_head out_of_order_queue;
145
146     /* SACKs data, these 2 need to be together (see tcp_options_write) */
147     struct tcp_sack_block duplicate_sack[1]; /* D-SACK block */
148     struct tcp_sack_block selective_acks[4]; /* The SACKS themselves*/
149
150     struct tcp_sack_block recv_sack_cache[4];
151
152     struct sk_buff *highest_sack;   /* skb just after the highest
153                      * skb with SACKed bit set
154                      * (validity guaranteed only if
155                      * sacked_out > 0)
156                      */
157
158     int     lost_cnt_hint;
159     u32     retransmit_high;    /* L-bits may be on up to this seqno */
160
161     u32 prior_ssthresh; /* ssthresh saved at recovery start */
162     u32 high_seq;   /* snd_nxt at onset of congestion   */
163
164     u32 retrans_stamp;  /* Timestamp of the last retransmit,
165                  * also used in SYN-SENT to remember stamp of
166                  * the first SYN. */
167     u32 undo_marker;    /* snd_una upon a new recovery episode. */
168     int undo_retrans;   /* number of undoable retransmissions. */
169     u32 total_retrans;  /* Total retransmits for entire connection */
170
171     u32 urg_seq;    /* Seq of received urgent pointer */
172     unsigned int        keepalive_time;   /* time before keep alive takes place */
173     unsigned int        keepalive_intvl;  /* time interval between keep alive probes */
174
175     int         linger2;
176
```

```
177 /* Receiver side RTT estimation */
178     struct {
179         u32 rtt;
180         u32 seq;
181         u32 time;
182     } rcv_rtt_est;
183 
184 /* Receiver queue space */
185     struct {
186         int space;
187         u32 seq;
188         u32 time;
189     } rcvq_space;
190 
191 /* TCP-specific MTU probe information. */
192     struct {
193         u32       probe_seq_start;
194         u32       probe_seq_end;
195     } mtu_probe;
196     u32 mtu_info; /* We received an ICMP_FRAG_NEEDED / ICMPV6_PKT_TOOBIG
197                * while socket was owned by user.
198                */
199 
200 #ifdef CONFIG_TCP_MD5SIG
201 /* TCP AF-Specific parts; only used by MD5 Signature support so far */
202     const struct tcp_sock_af_ops    *af_specific;
203 
204 /* TCP MD5 Signature Option information */
205     struct tcp_md5sig_info  __rcu *md5sig_info;
206 #endif
207 
208 /* TCP fastopen related information */
209     struct tcp_fastopen_request *fastopen_req;
210     /* fastopen_rsk points to request_sock that resulted in this big
211      * socket. Used to retransmit SYNACKs etc.
212      */
213     struct request_sock *fastopen_rsk;
214     u32 *saved_syn;
215 };
```

2.5.4 tcp_request_sock

```
1  struct tcp_request_sock {
2      struct inet_request_sock    req;
3      const struct tcp_request_sock_ops *af_specific;
4      struct skb_mstamp       snt_synack;                         /* first SYNACK sent time */
5      bool                tfo_listener;
6      u32             txhash;
7      u32             rcv_isn;
8      u32             snt_isn;
9      u32             last_oow_ack_time;                              /* last SYNACK */
10     u32             rcv_nxt;                                            /* the ack # by SYNACK. F
11                                                                                 * FastOpen i
12                                                                                * after data-
13                                                                              */
14 };
```

2.5.5 TCP 协议控制块–tcp_sock

```
struct tcp_sock {
    /* inet_connection_sock has to be the first member of tcp_sock */
    struct inet_connection_sock inet_conn;
    u16 tcp_header_len; /* Bytes of tcp header to send      */
    u16 gso_segs;   /* Max number of segs per GSO packet    */

/*
 *  Header prediction flags
 *  0x5?10 << 16 + snd_wnd in net byte order
 */
    __be32  pred_flags;

/*
 *  RFC793 variables by their proper names. This means you can
 *  read the code and the spec side by side (and laugh ...)
 *  See RFC793 and RFC1122. The RFC writes these in capitals.
 */
    u64 bytes_received; /* RFC4898 tcpEStatsAppHCThruOctetsReceived
                * sum(delta(rcv_nxt)), or how many bytes
                * were acked.
                */
    u32 segs_in;    /* RFC4898 tcpEStatsPerfSegsIn
                * total number of segments in.
                */
    u32 rcv_nxt;    /* What we want to receive next     */
    u32 copied_seq; /* Head of yet unread data      */
    u32 rcv_wup;    /* rcv_nxt on last window update sent   */
    u32 snd_nxt;    /* Next sequence we send        */
    u32 segs_out;   /* RFC4898 tcpEStatsPerfSegsOut
                * The total number of segments sent.
                */
    u64 bytes_acked;    /* RFC4898 tcpEStatsAppHCThruOctetsAcked
                * sum(delta(snd_una)), or how many bytes
                * were acked.
                */
    struct u64_stats_sync syncp; /* protects 64bit vars (cf tcp_get_info()) */

    u32 snd_una;    /* First byte we want an ack for    */
    u32 snd_sml;    /* Last byte of the most recently transmitted small packet */
    u32 rcv_tstamp; /* timestamp of last received ACK (for keepalives) */
    u32 lsndtime;   /* timestamp of last sent data packet (for restart window) */
    u32 last_oow_ack_time;  /* timestamp of last out-of-window ACK */

    u32 tsoffset;   /* timestamp offset */

    struct list_head tsq_node; /* anchor in tsq_tasklet.head list */
    unsigned long   tsq_flags;

    /* Data for direct copy to user */
    struct {
        struct sk_buff_head prequeue;
        struct task_struct  *task;
        struct msghdr       *msg;
        int         memory;
        int         len;
    } ucopy;

```

```
58      u32 snd_wl1;    /* Sequence for window update       */
59      u32 snd_wnd;    /* The window we expect to receive  */
60      u32 max_window; /* Maximal window ever seen from peer   */
61      u32 mss_cache;  /* Cached effective mss, not including SACKS */
62
63      u32 window_clamp;   /* Maximal window to advertise      */
64      u32 rcv_ssthresh;   /* Current window clamp         */
65
66      /* Information of the most recently (s)acked skb */
67      struct tcp_rack {
68          struct skb_mstamp mstamp; /* (Re)sent time of the skb */
69          u8 advanced; /* mstamp advanced since last lost marking */
70          u8 reord;    /* reordering detected */
71      } rack;
72      u16 advmss;     /* Advertised MSS           */
73      u8  unused;
74      u8  nonagle     : 4,/* Disable Nagle algorithm?             */
75          thin_lto    : 1,/* Use linear timeouts for thin streams */
76          thin_dupack : 1,/* Fast retransmit on first dupack      */
77          repair      : 1,
78          frto        : 1;/* F-RTO (RFC5682) activated in CA_Loss */
79      u8  repair_queue;
80      u8  do_early_retrans:1,/* Enable RFC5827 early-retransmit  */
81          syn_data:1, /* SYN includes data */
82          syn_fastopen:1, /* SYN includes Fast Open option */
83          syn_fastopen_exp:1,/* SYN includes Fast Open exp. option */
84          syn_data_acked:1,/* data in SYN is acked by SYN-ACK */
85          save_syn:1, /* Save headers of SYN packet */
86          is_cwnd_limited:1;/* forward progress limited by snd_cwnd? */
87      u32 tlp_high_seq;   /* snd_nxt at the time of TLP retransmit. */
88
89  /* RTT measurement */
90      u32 srtt_us;    /* smoothed round trip time << 3 in usecs */
91      u32 mdev_us;    /* medium deviation         */
92      u32 mdev_max_us;    /* maximal mdev for the last rtt period */
93      u32 rttvar_us;  /* smoothed mdev_max            */
94      u32 rtt_seq;    /* sequence number to update rttvar */
95      struct rtt_meas {
96          u32 rtt, ts;    /* RTT in usec and sampling time in jiffies. */
97      } rtt_min[3];
98
99      u32 packets_out;    /* Packets which are "in flight"    */
100     u32 retrans_out;    /* Retransmitted packets out        */
101     u32 max_packets_out;  /* max packets_out in last window */
102     u32 max_packets_seq;  /* right edge of max_packets_out flight */
103
104     u16 urg_data;   /* Saved octet of OOB data and control flags */
105     u8  ecn_flags;  /* ECN status bits.         */
106     u8  keepalive_probes; /* num of allowed keep alive probes   */
107     u32 reordering; /* Packet reordering metric.        */
108     u32 snd_up;     /* Urgent pointer       */
109
110 /*
111  *      Options received (usually on last packet, some only on SYN packets).
112  */
113     struct tcp_options_received rx_opt;
114
115 /*
```

```
116  *  Slow start and congestion control (see also Nagle, and Karn & Partridge)
117  */
118     u32 snd_ssthresh;   /* Slow start size threshold        */
119     u32 snd_cwnd;   /* Sending congestion window        */
120     u32 snd_cwnd_cnt;   /* Linear increase counter      */
121     u32 snd_cwnd_clamp; /* Do not allow snd_cwnd to grow above this */
122     u32 snd_cwnd_used;
123     u32 snd_cwnd_stamp;
124     u32 prior_cwnd; /* Congestion window at start of Recovery. */
125     u32 prr_delivered;  /* Number of newly delivered packets to
126                  * receiver in Recovery. */
127     u32 prr_out;    /* Total number of pkts sent during Recovery. */
128
129     u32 rcv_wnd;    /* Current receiver window      */
130     u32 write_seq;  /* Tail(+1) of data held in tcp send buffer */
131     u32 notsent_lowat;  /* TCP_NOTSENT_LOWAT */
132     u32 pushed_seq; /* Last pushed seq, required to talk to windows */
133     u32 lost_out;   /* Lost packets         */
134     u32 sacked_out; /* SACK'd packets           */
135     u32 fackets_out;    /* FACK'd packets           */
136
137     /* from STCP, retrans queue hinting */
138     struct sk_buff* lost_skb_hint;
139     struct sk_buff *retransmit_skb_hint;
140
141     /* OOO segments go in this list. Note that socket lock must be held,
142      * as we do not use sk_buff_head lock.
143      */
144     struct sk_buff_head out_of_order_queue;
145
146     /* SACKs data, these 2 need to be together (see tcp_options_write) */
147     struct tcp_sack_block duplicate_sack[1]; /* D-SACK block */
148     struct tcp_sack_block selective_acks[4]; /* The SACKS themselves*/
149
150     struct tcp_sack_block recv_sack_cache[4];
151
152     struct sk_buff *highest_sack;   /* skb just after the highest
153                      * skb with SACKed bit set
154                      * (validity guaranteed only if
155                      * sacked_out > 0)
156                      */
157
158     int     lost_cnt_hint;
159     u32     retransmit_high;    /* L-bits may be on up to this seqno */
160
161     u32 prior_ssthresh; /* ssthresh saved at recovery start */
162     u32 high_seq;   /* snd_nxt at onset of congestion   */
163
164     u32 retrans_stamp;  /* Timestamp of the last retransmit,
165                  * also used in SYN-SENT to remember stamp of
166                  * the first SYN. */
167     u32 undo_marker;    /* snd_una upon a new recovery episode. */
168     int undo_retrans;   /* number of undoable retransmissions. */
169     u32 total_retrans;  /* Total retransmits for entire connection */
170
171     u32 urg_seq;    /* Seq of received urgent pointer */
172     unsigned int        keepalive_time;   /* time before keep alive takes place */
173     unsigned int        keepalive_intvl;  /* time interval between keep alive probes */
174
```

```
175     int         linger2;
176
177 /* Receiver side RTT estimation */
178     struct {
179         u32 rtt;
180         u32 seq;
181         u32 time;
182     } rcv_rtt_est;
183
184 /* Receiver queue space */
185     struct {
186         int space;
187         u32 seq;
188         u32 time;
189     } rcvq_space;
190
191 /* TCP-specific MTU probe information. */
192     struct {
193         u32       probe_seq_start;
194         u32       probe_seq_end;
195     } mtu_probe;
196     u32 mtu_info; /* We received an ICMP_FRAG_NEEDED / ICMPV6_PKT_TOOBIG
197                * while socket was owned by user.
198                */
199
200 #ifdef CONFIG_TCP_MD5SIG
201 /* TCP AF-Specific parts; only used by MD5 Signature support so far */
202     const struct tcp_sock_af_ops    *af_specific;
203
204 /* TCP MD5 Signature Option information */
205     struct tcp_md5sig_info  __rcu *md5sig_info;
206 #endif
207
208 /* TCP fastopen related information */
209     struct tcp_fastopen_request *fastopen_req;
210     /* fastopen_rsk points to request_sock that resulted in this big
211      * socket. Used to retransmit SYNACKs etc.
212      */
213     struct request_sock *fastopen_rsk;
214     u32 *saved_syn;
215 };
```

2.5.6 tcp_skb_cb

在2.2.4中，我们分析过cb。在这一节中，我们将看到 TCP 层具体是如何使用这个控制缓冲区 (Control Buffer) 的。

2.5.6.1 TCP_SKB_CB

该宏用于访问给定的sk_buff的控制缓冲区的变量。在后续的章节中，可以在很多函数中看到它的身影。该宏的定义如下：

```
1 #define TCP_SKB_CB(__skb)       ((struct tcp_skb_cb *)&((__skb)->cb[0]))
```

可以看到，该宏实际上是将cb的指针强制转型成tcp_skb_cb 结构体的指针。也就是说，TCP 对于控制缓冲区的使用，可以从tcp_skb_cb 的定义分析出来

2.5.6.2 tcp_skb_cb 结构体

tcp_skb_cb结构体用于将每个 TCP 包中的控制信息传递给发送封包的代码。该结构体的定义如下：

```
struct tcp_skb_cb {
        __u32           seq;            /* 起始序号      */
        __u32           end_seq;        /* SEQ + FIN + SYN + datalen    */
        union {
                /* Note : tcp_tw_isn is used in input path only
                 *        (isn chosen by tcp_timewait_state_process())
                 *
                 *        tcp_gso_segs/size are used in write queue only,
                 *        cf tcp_skb_pcount()/tcp_skb_mss()
                 */
                __u32           tcp_tw_isn;
                struct {
                        u16     tcp_gso_segs;
                        u16     tcp_gso_size;
                };
        };
        __u8            tcp_flags;      /* TCP 头部的标志位 */

        __u8            sacked;         /* SACK/FACK 标志位          .    */
```

紧接着，定义了一些宏作为标志

```
#define TCPCB_SACKED_ACKED      0x01    /* SKB 被确认了                  */
#define TCPCB_SACKED_RETRANS    0x02    /* SKB 被重传了                  */
#define TCPCB_LOST              0x04    /* SKB 已丢失                    */
#define TCPCB_TAGBITS           0x07    /* 标志位掩码                    */
#define TCPCB_REPAIRED          0x10    /* SKB 被修复了 (no skb_mstamp)  */
#define TCPCB_EVER_RETRANS      0x80    /* SKB 曾经被重传过                */
#define TCPCB_RETRANS           (TCPCB_SACKED_RETRANS|TCPCB_EVER_RETRANS| \
                                TCPCB_REPAIRED)
```

接下来又继续定义 TCP 相关的位。

```
        __u8            ip_dsfield;     /* IPv4 tos or IPv6 dsfield     */
        /* 1 byte hole */
        __u32           ack_seq;        /* ACK 的序号                    */
        union {
                struct inet_skb_parm    h4;
#if IS_ENABLED(CONFIG_IPV6)
                struct inet6_skb_parm   h6;
#endif
        } header;       /* For incoming frames          */
};
```

CHAPTER 3

TCP 建立连接过程

Contents

3.1 TCP 主动打开-客户

3.1.1 基本流程

主动打开是通过 connect 系统调用来完成的。这一系统调用最终会调用传输层的`tcp_v4_connect`函数。

3.1.2 第一次握手——构造并发送 SYN 包

3.1.2.1 tcp_v4_connect

`tcp_v4_connect`的主要作用是进行一系列的判断，初始化传输控制块并调用相关函数发送 SYN 包。

```
1  /* 这个函数会初始化一个发送用的连接 */
2  int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
3  {
4          struct sockaddr_in *usin = (struct sockaddr_in *)uaddr;
5          struct inet_sock *inet = inet_sk(sk);
6          struct tcp_sock *tp = tcp_sk(sk);
7          __be16 orig_sport, orig_dport;
8          __be32 daddr, nexthop;
9          struct flowi4 *fl4;
10         struct rtable *rt;
11         int err;
12         struct ip_options_rcu *inet_opt;
13
14         /* 检验传入的结构体的大小是否满足要求 */
15         if (addr_len < sizeof(struct sockaddr_in))
16                 return -EINVAL;
17
18         /* 检验协议族是否正确 */
19         if (usin->sin_family != AF_INET)
20                 return -EAFNOSUPPORT;
21
22         /* 将下一跳地址和目的地址暂时设置为用户传入的 IP 地址 */
23         nexthop = daddr = usin->sin_addr.s_addr;
24         inet_opt = rcu_dereference_protected(inet->inet_opt,
25                                              sock_owned_by_user(sk));
26         /* 如果选择源地址路由，则将下一跳地址设置为 IP 选项中的 faddr */
```

```
27          if (inet_opt && inet_opt->opt.srr) {
28                  if (!daddr)
29                          return -EINVAL;
30                  nexthop = inet_opt->opt.faddr;
31          }
```

源地址路由是一种特殊的路由策略。一般路由都是通过目的地址来进行的。而有时也需要通过源地址来进行路由，例如在有多个网卡等情况下，可以根据源地址来决定走哪个网卡等等。

```
1           orig_sport = inet->inet_sport;
2           orig_dport = usin->sin_port;
3           fl4 = &inet->cork.fl.u.ip4;
4           /* 获取目标的路由缓存项 */
5           rt = ip_route_connect(fl4, nexthop, inet->inet_saddr,
6                                 RT_CONN_FLAGS(sk), sk->sk_bound_dev_if,
7                                 IPPROTO_TCP,
8                                 orig_sport, orig_dport, sk);
9           if (IS_ERR(rt)) {
10                  err = PTR_ERR(rt);
11                  if (err == -ENETUNREACH)
12                          IP_INC_STATS(sock_net(sk), IPSTATS_MIB_OUTNOROUTES);
13                  return err;
14          }
15
16          if (rt->rt_flags & (RTCF_MULTICAST | RTCF_BROADCAST)) {
17                  ip_rt_put(rt);
18                  return -ENETUNREACH;
19          }
20
21          /* 如果没有开启源路由功能，则采用查找到的缓存项 */
22          if (!inet_opt || !inet_opt->opt.srr)
23                  daddr = fl4->daddr;
24
25          /* 如果没有设置源地址，则设置为缓存项中的源地址 */
26          if (!inet->inet_saddr)
27                  inet->inet_saddr = fl4->saddr;
28          sk_rcv_saddr_set(sk, inet->inet_saddr);
29
30          /* 如果该传输控制块的时间戳已被使用过，则重置各状态 */
31          if (tp->rx_opt.ts_recent_stamp && inet->inet_daddr != daddr) {
32                  /* Reset inherited state */
33                  tp->rx_opt.ts_recent       = 0;
34                  tp->rx_opt.ts_recent_stamp = 0;
35                  if (likely(!tp->repair))
36                          tp->write_seq      = 0;
37          }
38
39          /* 在启用了 tw_recycle 的情况下，重设时间戳 */
40          if (tcp_death_row.sysctl_tw_recycle &&
41              !tp->rx_opt.ts_recent_stamp && fl4->daddr == daddr)
42                  tcp_fetch_timewait_stamp(sk, &rt->dst);
43
44          /* 设置传输控制块 */
45          inet->inet_dport = usin->sin_port;
46          sk_daddr_set(sk, daddr);
47
48          inet_csk(sk)->icsk_ext_hdr_len = 0;
```

```
49          if (inet_opt)
50                  inet_csk(sk)->icsk_ext_hdr_len = inet_opt->opt.optlen;
51
52          /* 设置 MSS 大小 */
53          tp->rx_opt.mss_clamp = TCP_MSS_DEFAULT;
54
55          /* Socket identity is still unknown (sport may be zero).
56           * However we set state to SYN-SENT and not releasing socket
57           * lock select source port, enter ourselves into the hash tables and
58           * complete initialization after this.
59           */
60          /* 将 TCP 的状态设置为 SYN-Sent */
61          tcp_set_state(sk, TCP_SYN_SENT);
62          err = inet_hash_connect(&tcp_death_row, sk);
63          if (err)
64                  goto failure;
65
66          sk_set_txhash(sk);
67
68          rt = ip_route_newports(fl4, rt, orig_sport, orig_dport,
69                                 inet->inet_sport, inet->inet_dport, sk);
70          if (IS_ERR(rt)) {
71                  err = PTR_ERR(rt);
72                  rt = NULL;
73                  goto failure;
74          }
75          /* 将目的地址提交到套接字  */
76          sk->sk_gso_type = SKB_GSO_TCPV4;
77          sk_setup_caps(sk, &rt->dst);
78
79          /* 如果没有设置序号，则计算初始序号 */
80          if (!tp->write_seq && likely(!tp->repair))
81                  tp->write_seq = secure_tcp_sequence_number(inet->inet_saddr,
82                                                             inet->inet_daddr,
83                                                             inet->inet_sport,
84                                                             usin->sin_port);
85
86          /* 计算 IP 首部的 id 域的值 */
87          inet->inet_id = tp->write_seq ^ jiffies;
88
89          /* 调用 tcp_connect 构造并发送 SYN 包 */
90          err = tcp_connect(sk);
91
92          rt = NULL;
93          if (err)
94                  goto failure;
95
96          return 0;
```

总结起来，tcp_v4_connect是在根据用户提供的目的地址，设置好了传输控制块，为传输做好准备。如果在这一过程中出现错误，则会跳到错误处理代码

```
1   failure:
2           /* 将状态设定为 TCP_CLOSE，释放端口，并返回错误值。
3            */
4           tcp_set_state(sk, TCP_CLOSE);
5           ip_rt_put(rt);
6           sk->sk_route_caps = 0;
```

```
7           inet->inet_dport = 0;
8           return err;
```

3.1.2.2 tcp_ connect

上面的tcp_v4_connect会进行一系列的判断，之后真正构造 SYN 包的部分被放置在了tcp_connect中。接下来，我们来分析这个函数。

```
1   /* 该函数用于构造并发送 SYN 包 */
2   int tcp_connect(struct sock *sk)
3   {
4           struct tcp_sock *tp = tcp_sk(sk);
5           struct sk_buff *buff;
6           int err;
7
8           /* 初始化 tcp 连接 */
9           tcp_connect_init(sk);
10
11          if (unlikely(tp->repair)) {
12                  /* 如果 repair 位被置 1，那么结束 TCP 连接 */
13                  tcp_finish_connect(sk, NULL);
14                  return 0;
15          }
16
17          /* 分配一个 sk_buff */
18          buff = sk_stream_alloc_skb(sk, 0, sk->sk_allocation, true);
19          if (unlikely(!buff))
20                  return -ENOBUFS;
21
22          /* 初始化 skb，并自增 write_seq 的值 */
23          tcp_init_nondata_skb(buff, tp->write_seq++, TCPHDR_SYN);
24          /* 设置时间戳 */
25          tp->retrans_stamp = tcp_time_stamp;
26          /* 将当前的 sk_buff 添加到发送队列中 */
27          tcp_connect_queue_skb(sk, buff);
28          /* ECN 支持 */
29          tcp_ecn_send_syn(sk, buff);
30
31          /* 发送 SYN 包，这里同时还考虑了 Fast Open 的情况 */
32          err = tp->fastopen_req ? tcp_send_syn_data(sk, buff) :
33                tcp_transmit_skb(sk, buff, 1, sk->sk_allocation);
34          if (err == -ECONNREFUSED)
35                  return err;
36
37          /* We change tp->snd_nxt after the tcp_transmit_skb() call
38           * in order to make this packet get counted in tcpOutSegs.
39           */
40          tp->snd_nxt = tp->write_seq;
41          tp->pushed_seq = tp->write_seq;
42          TCP_INC_STATS(sock_net(sk), TCP_MIB_ACTIVEOPENS);
43
44          /* 设定超时重传定时器 */
45          inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
46                                    inet_csk(sk)->icsk_rto, TCP_RTO_MAX);
47          return 0;
48  }
```

3.1.3 第二次握手——接收 SYN+ACK 包

tcp_rcv_state_process实现了 TCP 状态机相对较为核心的一个部分。该函数可以处理除 ESTABLISHED 和 TIME_WAIT 状态以外的情况下的接收过程。这里，我们仅关系主动连接情况下的处理。在3.1.2.1中，我们分析源码时得出，客户端发送 SYN 包后，会将状态机设置为TCP_SYN_SENT状态。因此，我们仅在这里分析该状态下的代码。

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct inet_connection_sock *icsk = inet_csk(sk);
        const struct tcphdr *th = tcp_hdr(skb);
        struct request_sock *req;
        int queued = 0;
        bool acceptable;

        tp->rx_opt.saw_tstamp = 0;

        switch (sk->sk_state) {
        case TCP_CLOSE:
                /* CLOSE 状态的处理代码 */

        case TCP_LISTEN:
                /* LISTEN 状态的处理代码 */

        case TCP_SYN_SENT:
                /* 处理接收到的数据段 */
                queued = tcp_rcv_synsent_state_process(sk, skb, th);
                if (queued >= 0)
                        return queued;

                /* 处理紧急数据并检测是否有数据需要发送 */
                tcp_urg(sk, skb, th);
                __kfree_skb(skb);
                tcp_data_snd_check(sk);
                return 0;
        }

        /* 处理其他情况的代码 */
}
```

具体的处理代码被放在了tcp_rcv_synsent_state_process中，通过命名就可以看出，该函数是专门用于处理 SYN_SENT 状态下收到的数据的。

```
static int tcp_rcv_synsent_state_process(struct sock *sk, struct sk_buff *skb,
                                         const struct tcphdr *th)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        struct tcp_fastopen_cookie foc = { .len = -1 };
        int saved_clamp = tp->rx_opt.mss_clamp;

        /* 解析 TCP 选项，并保存在传输控制块中 */
        tcp_parse_options(skb, &tp->rx_opt, 0, &foc);
        if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr)
                tp->rx_opt.rcv_tsecr -= tp->tsoffset;
```

接下来的部分，就是按照 TCP 协议的标准来实现相应的行为。注释中出现的 RFC793 即是描述 TCP 协议的 RFC 原文中的文本。

```
        if (th->ack) {
                /* rfc793:
                 * "If the state is SYN-SENT then
                 *    first check the ACK bit
                 *      If the ACK bit is set
                 *        If SEG.ACK =< ISS, or SEG.ACK > SND.NXT, send
                 *        a reset (unless the RST bit is set, if so drop
                 *        the segment and return)"
                 * ISS 代表初始发送序号 (Initial Send Sequence number)
                 */
                if (!after(TCP_SKB_CB(skb)->ack_seq, tp->snd_una) ||
                    after(TCP_SKB_CB(skb)->ack_seq, tp->snd_nxt))
                        goto reset_and_undo;

                if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr &&
                    !between(tp->rx_opt.rcv_tsecr, tp->retrans_stamp,
                             tcp_time_stamp)) {
                        NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PAWSACTIVEREJECTED);
                        goto reset_and_undo;
                }
```

上面的一段根据 RFC 在判断 ACK 的值是否在初始发送序号和下一个序号之间，如果不再，则发送一个重置。

```
                /* 此时, ACK 已经被接受了
                 *
                 * "If the RST bit is set
                 *    If the ACK was acceptable then signal the user "error:
                 *    connection reset", drop the segment, enter CLOSED state,
                 *    delete TCB, and return."
                 */

                if (th->rst) {
                        tcp_reset(sk);
                        goto discard;
                }
```

接下来，判断了收到的包的 RST 位，如果设置了 RST，则丢弃该分组，并进入 CLOSED 状态。

```
                /* rfc793:
                 *   "fifth, if neither of the SYN or RST bits is set then
                 *    drop the segment and return."
                 *
                 *    See note below!
                 *                                        --ANK(990513)
                 */
                if (!th->syn)
                        goto discard_and_undo;
```

之后，根据 RFC 的说法，如果既没有设置 SYN 位，也没有设置 RST 位，那么就将分组丢弃掉。前面已经判断了 RST 位了，因此，这里判断一下 SYN 位。

接下来就准备进入到 ESTABLISHED 状态了。

```
                /* rfc793:
                 *   "If the SYN bit is on ...
                 *    are acceptable then ...
                 *    (our SYN has been ACKed), change the connection
                 *    state to ESTABLISHED..."
                 */

                tcp_ecn_rcv_synack(tp, th);

                /* 初始化与窗口有关的参数 */
                tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);
                tcp_ack(sk, skb, FLAG_SLOWPATH);

                /* Ok.. it's good. Set up sequence numbers and
                 * move to established.
                 */
                tp->rcv_nxt = TCP_SKB_CB(skb)->seq + 1;
                tp->rcv_wup = TCP_SKB_CB(skb)->seq + 1;
```

对于 SYN 和 SYN/ACK 段是不进行窗口放大的。关于 RFC1323 窗口放大相关的内容，我们会在 3.1.6.1中详细讨论。接下来手动设定了窗口缩放相关的参数，使得缩放不生效。

```
                /* RFC1323: The window in SYN & SYN/ACK segments is
                 * never scaled.
                 */
                tp->snd_wnd = ntohs(th->window);

                if (!tp->rx_opt.wscale_ok) {
                        tp->rx_opt.snd_wscale = tp->rx_opt.rcv_wscale = 0;
                        tp->window_clamp = min(tp->window_clamp, 65535U);
                }
```

根据时间戳选项，设定相关字段及 TCP 头部长度。

```
                if (tp->rx_opt.saw_tstamp) {
                        tp->rx_opt.tstamp_ok       = 1;
                        tp->tcp_header_len =
                                sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED;
                        tp->advmss          -= TCPOLEN_TSTAMP_ALIGNED;
                        tcp_store_ts_recent(tp);
                } else {
                        tp->tcp_header_len = sizeof(struct tcphdr);
                }
```

之后会根据设定开启 FACK 机制。FACK 是在 SACK 机制上发展来的。SACK 用于准确地获知哪些包丢失了需要重传。开启 SACK 后，可以让发送端只重传丢失的包。而当重传的包比较多时，会进一步导致网络繁忙，FASK 用来做重传过程中的拥塞控制。

```
                if (tcp_is_sack(tp) && sysctl_tcp_fack)
                        tcp_enable_fack(tp);
```

最后初始化 MTU、MSS 等参数并完成 TCP 连接过程。

```
                tcp_mtup_init(sk);
                tcp_sync_mss(sk, icsk->icsk_pmtu_cookie);
                tcp_initialize_rcv_mss(sk);

                /* Remember, tcp_poll() does not lock socket!
                 * Change state from SYN-SENT only after copied_seq
                 * is initialized. */
                tp->copied_seq = tp->rcv_nxt;

                smp_mb();
                tcp_finish_connect(sk, skb);
```

此后，开始处理一些特殊情况。Fast Open 启用的情况下，SYN 包也会带有数据。这里调用 `tcp_rcv_fastopen_synack`函数处理 SYN 包附带的数据。

```
                if ((tp->syn_fastopen || tp->syn_data) &&
                    tcp_rcv_fastopen_synack(sk, skb, &foc))
                        return -1;

                /* 根据情况进入延迟确认模式 */
                if (sk->sk_write_pending ||
                    icsk->icsk_accept_queue.rskq_defer_accept ||
                    icsk->icsk_ack.pingpong) {
                        /* Save one ACK. Data will be ready after
                         * several ticks, if write_pending is set.
                         *
                         * It may be deleted, but with this feature tcpdumps
                         * look so _wonderfully_ clever, that I was not able
                         * to stand against the temptation 8)     --ANK
                         */
                        inet_csk_schedule_ack(sk);
                        icsk->icsk_ack.lrcvtime = tcp_time_stamp;
                        tcp_enter_quickack_mode(sk);
                        inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
                                                  TCP_DELACK_MAX, TCP_RTO_MAX);

discard:
                        __kfree_skb(skb);
                        return 0;
                } else {
                        /* 回复 ACK 包 */
                        tcp_send_ack(sk);
                }
                return -1;
        }
```

最后是一些异常情况的处理

```
        /* 进入该分支意味着包中不包含 ACK */

        if (th->rst) {
                /* rfc793:
                 * "If the RST bit is set
                 *
                 *      Otherwise (no ACK) drop the segment and return."
                 * 如果收到了 RST 包，则直接丢弃并返回。
                 */

```

```
11                  goto discard_and_undo;
12          }
13
14          /* PAWS 检查 */
15          if (tp->rx_opt.ts_recent_stamp && tp->rx_opt.saw_tstamp &&
16              tcp_paws_reject(&tp->rx_opt, 0))
17                  goto discard_and_undo;
18
19          /* 仅有 SYN 而无 ACK 的处理 */
20          if (th->syn) {
21                  /* We see SYN without ACK. It is attempt of
22                   * simultaneous connect with crossed SYNs.
23                   * Particularly, it can be connect to self.
24                   */
25                  tcp_set_state(sk, TCP_SYN_RECV);
26
27                  /* 下面的处理和前面几乎一样 */
28                  if (tp->rx_opt.saw_tstamp) {
29                          tp->rx_opt.tstamp_ok = 1;
30                          tcp_store_ts_recent(tp);
31                          tp->tcp_header_len =
32                                  sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED;
33                  } else {
34                          tp->tcp_header_len = sizeof(struct tcphdr);
35                  }
36
37                  tp->rcv_nxt = TCP_SKB_CB(skb)->seq + 1;
38                  tp->copied_seq = tp->rcv_nxt;
39                  tp->rcv_wup = TCP_SKB_CB(skb)->seq + 1;
40
41                  /* RFC1323: The window in SYN & SYN/ACK segments is
42                   * never scaled.
43                   */
44                  tp->snd_wnd    = ntohs(th->window);
45                  tp->snd_wl1    = TCP_SKB_CB(skb)->seq;
46                  tp->max_window = tp->snd_wnd;
47
48                  tcp_ecn_rcv_syn(tp, th);
49
50                  tcp_mtup_init(sk);
51                  tcp_sync_mss(sk, icsk->icsk_pmtu_cookie);
52                  tcp_initialize_rcv_mss(sk);
53
54                  tcp_send_synack(sk);
55  #if 0
56                  /* Note, we could accept data and URG from this segment.
57                   * There are no obstacles to make this (except that we must
58                   * either change tcp_recvmsg() to prevent it from returning data
59                   * before 3WHS completes per RFC793, or employ TCP Fast Open).
60                   *
61                   * However, if we ignore data in ACKless segments sometimes,
62                   * we have no reasons to accept it sometimes.
63                   * Also, seems the code doing it in step6 of tcp_rcv_state_process
64                   * is not flawless. So, discard packet for sanity.
65                   * Uncomment this return to process the data.
66                   */
67                  return -1;
68  #else
```

```
69                  goto discard;
70  #endif
71          }
72          /* "fifth, if neither of the SYN or RST bits is set then
73           * drop the segment and return."
74           */
75
76  discard_and_undo:
77          tcp_clear_options(&tp->rx_opt);
78          tp->rx_opt.mss_clamp = saved_clamp;
79          goto discard;
80
81  reset_and_undo:
82          tcp_clear_options(&tp->rx_opt);
83          tp->rx_opt.mss_clamp = saved_clamp;
84          return 1;
85  }
```

3.1.4 第三次握手——发送 ACK 包

3.1.4.1 tcp_send_ack

在3.1.3分析的代码的最后，我们看到它调用了`tcp_send_ack()`来发送 ACK 包，从而实现第三次握手。

```
1   /* 该函数用于发送 ACK，并更新窗口的大小 */
2   void tcp_send_ack(struct sock *sk)
3   {
4           struct sk_buff *buff;
5
6           /* 如果当前的套接字已经被关闭了，那么直接返回。 */
7           if (sk->sk_state == TCP_CLOSE)
8                   return;
9
10          tcp_ca_event(sk, CA_EVENT_NON_DELAYED_ACK);
11
12          /* We are not putting this on the write queue, so
13           * tcp_transmit_skb() will set the ownership to this
14           * sock.
15           * 为数据包分配空间
16           */
17          buff = alloc_skb(MAX_TCP_HEADER, sk_gfp_atomic(sk, GFP_ATOMIC));
18          if (!buff) {
19                  inet_csk_schedule_ack(sk);
20                  inet_csk(sk)->icsk_ack.ato = TCP_ATO_MIN;
21                  inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
22                                            TCP_DELACK_MAX, TCP_RTO_MAX);
23                  return;
24          }
25
26          /* 初始化 ACK 包 */
27          skb_reserve(buff, MAX_TCP_HEADER);
28          tcp_init_nondata_skb(buff, tcp_acceptable_seq(sk), TCPHDR_ACK);
29
30          /* We do not want pure acks influencing TCP Small Queues or fq/pacing
31           * too much.
32           * SKB_TRUESIZE(max(1 .. 66, MAX_TCP_HEADER)) is unfortunately ~784
```

```
33              * We also avoid tcp_wfree() overhead (cache line miss accessing
34              * tp->tsq_flags) by using regular sock_wfree()
35              */
36             skb_set_tcp_pure_ack(buff);
37
38             /* 添加时间戳并发送 ACK 包 */
39             skb_mstamp_get(&buff->skb_mstamp);
40             tcp_transmit_skb(sk, buff, 0, sk_gfp_atomic(sk, GFP_ATOMIC));
41     }
```

3.1.5 tcp_transmit_skb

```
1   /* 为 SYN 包计算 TCP 选项，这个函数中计算出来的还不是最终的格式。
2    */
3   static unsigned int tcp_syn_options(struct sock *sk, struct sk_buff *skb,
4                                       struct tcp_out_options *opts,
5                                       struct tcp_md5sig_key **md5);
6   /* 为已经建立连接的套接字计算 TCP 选项，这个函数中计算出来的还不是最终的格式。
7    */
8   static unsigned int tcp_established_options(struct sock *sk, struct sk_buff *skb,
9                                               struct tcp_out_options *opts,
10                                              struct tcp_md5sig_key **md5);
11  /* 在 skb 中为头部流出空间。
12   */
13  skb_push(skb, tcp_header_size);
14  /* 判断 skb 是否为一个纯 ACK。这里把实现也放出来了。可以看到，纯 ACK 的包最显著
15   * 的特点是其长度。Linux 里通过判断长度直接快速判断出 skb 是否为一个纯 ACK。
16   */
17  static inline bool skb_is_tcp_pure_ack(const struct sk_buff *skb)
18  {
19          return skb->truesize == 2;
20  }
21  /* 重置传输层的 header 的指针？
22   */
23  static inline void skb_reset_transport_header(struct sk_buff *skb)
24  {
25          skb->transport_header = skb->data - skb->head;
26  }
27  /* 选择发送窗口的大小
28   */
29  tcp_select_window(sk);
30  tcp_urg_mode(tp);
31  before(tcb->seq, tp->snd_up);
32  tcp_options_write((__be32 *)(th + 1), tp, &opts);
33  tcp_event_ack_sent(sk, tcp_skb_pcount(skb));
34  tcp_event_data_sent(tp, sk);
35  queue_xmit();
36  tcp_enter_cwr(sk);
37  net_xmit_eval(err);
```

3.1.6 tcp_select_window(struct sk_buff *skb)

这个函数的作用是选择一个新的窗口大小以用于更新tcp_sock。返回的结果根据RFC1323 进行了缩放。

3.1.6.1 RFC1323——高性能 TCP 扩展 (TCP Extensions for High Performance)

这个 RFC 主要是在考虑高带宽高延迟网络下如何提升 TCP 的性能。就好像一个又粗又长的管道，如果想要管道的利用率高，就要尽可能地把管道填满。但是 TCP 能够同时发送的东西的上限是受到发送窗口的限制的。超过了窗口大小，就必须等待 ACK 确认才可以继续发送。

然而，在 TCP 头部中，只有 16 位的一个域用于说明窗口大小。也就是说，窗口大小最大只能达到 $2^{16} = 64K$ 。为了解决这一问题，RFC1323 新增了一个 TCP 选项，用于放大窗口的大小。该选项的值代表将原窗口大小放大 2 的幂倍。

个人认为这个设计很有好。采用 2 的幂来缩放可以很大程度地扩展窗口的大小，因为 2 的幂增长得很快。而且可以通过位移运算来实现缩放，性能上也很好。

3.1.6.2 代码分析

```
1  static u16 tcp_select_window(struct sock *sk)
2  {
3          struct tcp_sock *tp = tcp_sk(sk);
4          u32 old_win = tp->rcv_wnd;
5          u32 cur_win = tcp_receive_window(tp);
6          u32 new_win = __tcp_select_window(sk);
7          /* old_win 是接收方窗口的大小。
8           * cur_win 当前的接收窗口大小。
9           * new_win 是新选择出来的窗口大小。
10          */
11
12         /* 当新窗口的大小小于当前窗口的大小时，不能缩减窗口大小。
13          * 这是 IEEE 强烈不建议的一种行为。
14          */
15         if (new_win < cur_win) {
16                 /* Danger Will Robinson!
17                  * Don't update rcv_wup/rcv_wnd here or else
18                  * we will not be able to advertise a zero
19                  * window in time.  --DaveM
20                  *
21                  * Relax Will Robinson.
22                  */
23                 if (new_win == 0)
24                         NET_INC_STATS(sock_net(sk),
25                                       LINUX_MIB_TCPWANTZEROWINDOWADV);
26                 /* 当计算出来的新窗口小于当前窗口时，将新窗口设置为大于 cur_win
27                  * 的 1<<tp->rx_opt.rcv_wscale 的整数倍。
28                  */
29                 new_win = ALIGN(cur_win, 1 << tp->rx_opt.rcv_wscale);
30         }
31         /* 将当前的接收窗口设置为新的窗口大小。*/
32         tp->rcv_wnd = new_win;
33         tp->rcv_wup = tp->rcv_nxt;
34
35         /* 判断当前窗口未越界。*/
36         if (!tp->rx_opt.rcv_wscale && sysctl_tcp_workaround_signed_windows)
37                 new_win = min(new_win, MAX_TCP_WINDOW);
38         else
```

```
39                  new_win = min(new_win, (65535U << tp->rx_opt.rcv_wscale));
40
41          /* RFC1323 缩放窗口大小。这里之所以是右移，是因为此时的 new_win 是
42           * 窗口的真正大小。所以返回时需要返回正常的可以放在 16 位整型中的窗口大小。
43           * 所以需要右移。
44           */
45          new_win >>= tp->rx_opt.rcv_wscale;
46
47          /* If we advertise zero window, disable fast path. */
48          if (new_win == 0) {
49                  tp->pred_flags = 0;
50                  if (old_win)
51                          NET_INC_STATS(sock_net(sk),
52                                        LINUX_MIB_TCPTOZEROWINDOWADV);
53          } else if (old_win == 0) {
54                  NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPFROMZEROWINDOWADV);
55          }
56
57          return new_win;
58  }
```

在这个过程中，还调用了`__tcp_select_window(sk)`来计算新的窗口大小。该函数会尝试增加窗口的大小，但是有两个限制条件：

1. 窗口不能收缩 (RFC793)
2. 每个 socket 所能使用的内存是有限制的。

RFC 1122 中说：

> "the suggested [SWS] avoidance algorithm for the receiver is to keep RECV.NEXT + RCV.WIN fixed until: RCV.BUFF - RCV.USER - RCV.WINDOW >= min(1/2 RCV.BUFF, MSS)"
>
> 推荐的用于接收方的糊涂窗口综合症的避免算法是保持 recv.next+rcv.win 不变，直到：RCV.BUFF - RCV.USER - RCV.WINDOW >= min(1/2 RCV.BUFF, MSS)

换句话说，就是除非缓存的大小多出来至少一个 MSS 那么多字节，否则不要增长窗口右边界的大小。

然而，根据 Linux 注释中的说法，被推荐的这个算法会破坏头预测 (header prediction)，因为头预测会假定`th->window`不变。严格地说，保持`th->window`固定不变会违背接收方的用于防止糊涂窗口综合症的准则。在这种规则下，一个单字节的包的流会引发窗口的右边界总是提前一个字节。当然，如果发送方实现了预防糊涂窗口综合症的方法，那么就不会出现问题。

Linux 的 TCP 部分的作者们参考了 BSD 的实现方法。BSD 在这方面的做法是是，如果空闲空间小于最大可用空间的 $\frac{1}{4}$，且空闲空间小于 mss 的 $\frac{1}{2}$，那么就把窗口设置为 0。否则，只是单纯地阻止窗口缩小，或者阻止窗口大于最大可表示的范围 (the largest representable value)。BSD 的方法似乎“意外地”使得窗口基本上都是 MSS 的整倍数。且很多情况下窗口大小都是固定不变的。因此，Linux 采用强制窗口为 MSS 的整倍数，以获得相似的行为。

```
u32 __tcp_select_window(struct sock *sk)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        int mss = icsk->icsk_ack.rcv_mss;
        int free_space = tcp_space(sk);
        int allowed_space = tcp_full_space(sk);
        int full_space = min_t(int, tp->window_clamp, allowed_space);
        int window;

        /* 如果 mss 超过了总共的空间大小，那么把 mss 限制在允许的空间范围内。 */
        if (mss > full_space)
                mss = full_space;

        if (free_space < (full_space >> 1)) {
                /* 当空闲空间小于允许空间的一半时。 */
                icsk->icsk_ack.quick = 0;

                if (tcp_under_memory_pressure(sk))
                        tp->rcv_ssthresh = min(tp->rcv_ssthresh,
                                               4U * tp->advmss);

                /* free_space 有可能成为新的窗口的大小，因此，需要考虑
                 * 窗口扩展的影响。
                 */
                free_space = round_down(free_space, 1 << tp->rx_opt.rcv_wscale);

                /* 如果空闲空间小于 mss 的大小，或者低于最大允许空间的的 1/16，那么，
                 * 返回 0 窗口。否则，tcp_clamp_window() 会增长接收缓存到 tcp_rmem[2]。
                 * 新进入的数据会由于内醋限制而被丢弃。对于较大的窗口，单纯地探测 mss 的
                 * 大小以宣告 0 窗口有些太晚了（可能会超过限制）。
                 */
                if (free_space < (allowed_space >> 4) || free_space < mss)
                        return 0;
        }

        if (free_space > tp->rcv_ssthresh)
                free_space = tp->rcv_ssthresh;

        /* 这里处理一个例外情况，就是如果开启了窗口缩放，那么就没法对齐 mss 了。
         * 所以就保持窗口是对齐 2 的幂的。
         */
        window = tp->rcv_wnd;
        if (tp->rx_opt.rcv_wscale) {
                window = free_space;

                /* Advertise enough space so that it won't get scaled away.
                 * Import case: prevent zero window announcement if
                 * 1<<rcv_wscale > mss.
                 */
                if (((window >> tp->rx_opt.rcv_wscale) << tp->rx_opt.rcv_wscale) != window)
                        window = (((window >> tp->rx_opt.rcv_wscale) + 1)
                                  << tp->rx_opt.rcv_wscale);
        } else {
                /* 如果内存条件允许，那么就把窗口设置为 mss 的整倍数。
                 * 或者如果 free_space > 当前窗口大小加上全部允许的空间的一半，
                 * 那么，就将窗口大小设置为 free_space
                 */
```

```
59                  if (window <= free_space - mss || window > free_space)
60                          window = (free_space / mss) * mss;
61                  else if (mss == full_space &&
62                           free_space > window + (full_space >> 1))
63                          window = free_space;
64          }
65
66          return window;
67  }
```

3.2 TCP 被动打开-服务器

3.2.1 基本流程

tcp 想要被动打开，就必须得先进行 listen 调用。而对于一台主机，它如果想要作为服务器，它会在什么时候进行 listen 调用呢？不难想到，它在启动某个需要 TCP 连接的高级应用程序的时候，就会执行 listen 调用。经过 listen 调用之后，系统内部其实创建了一个监听套接字，专门负责监听是否有数据发来，而不会负责传输数据。

当客户端的第一个 syn 包到达服务器时，其实 linux 内核并不会创建 sock 结构体，而是创建一个轻量级的`request_sock`结构体，里面能唯一确定某个客户端发来的 syn 的信息，接着就发送 syn、ack 给客户端。

客户端一般就接着回 ack。这时，我们能从 ack 中，取出信息，在一堆`request_sock`匹配，看看是否之前有这个 ack 对应的 syn 发过来过。如果之前发过 syn，那么现在我们就能找到`request_sock`，也就是客户端 syn 时建立的`request_sock`。此时，我们内核才会为这条流创建 sock 结构体，毕竟，sock 结构体比`request_sock`大的多，犯不着三次握手都没建立起来我就建立一个大的结构体。当三次握手建立以后，内核就建立一个相对完整的 sock。所谓相对完整，其实也是不完整。因为如果写过 socket 程序，你就知道，所谓的真正完整，是建立 socket，而不是 sock(socket 结构体中有一个指针 sock *sk，显然 sock 只是 socket 的一个子集)。那么我们什么时候才会创建完整的 socket，或者换句话说，什么时候使得 sock 结构体和文件系统关联从而绑定一个 fd，用这个 fd 就可以用来传输数据呢？所谓 fd(file descriptor)，一般是 BSD Socket 的用法，用在 Unix/Linux 系统上。在 Unix/Linux 系统下，一个 socket 句柄，可以看做是一个文件，在 socket 上收发数据，相当于对一个文件进行读写，所以一个 socket 句柄，通常也用表示文件句柄的 fd 来表示。

如果你有 socket 编程经验，那么你一定能想到，那就是在 accept 系统调用时，返回了一个 fd，所以说，是你在 accept 时，你三次握手完成后建立的 sock 才绑定了一个 fd。

First Shake Hand of Server.png First Shake Hand of Server.bb

3.2.2 第一次握手：接受 SYN 段

3.2.2.1 第一次握手函数调用关系

3.2.2.2 tcp_v4_do_rcv

在进行第一次握手的时候，TCP 必然处于 LISTEN 状态。传输控制块接收处理的段都由`tcp_v4_do_rcv`来处理。该函数位于/net/ipv4/tcp_ipv4.c 中。该函数会根据不同的 TCP 状态进行不同的处理，这里我们只是讨论服务器第一次握手的函数处理过程。

```
/* The socket must have it's spinlock held when we get
 * here, unless it is a TCP_LISTEN socket.
 *
 * We have a potential double-lock case here, so even when
 * doing backlog processing we use the BH locking scheme.
 * This is because we cannot sleep with the original spinlock
 * held.
 */
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
    struct sock *rsk;

    /* 省略无关代码 */

    if (tcp_checksum_complete(skb))
        goto csum_err;

    if (sk->sk_state == TCP_LISTEN) {
        struct sock *nsk = tcp_v4_cookie_check(sk, skb);

        if (!nsk)
            goto discard;
        if (nsk != sk) {
            sock_rps_save_rxhash(nsk, skb);
            sk_mark_napi_id(nsk, skb);
            if (tcp_child_process(sk, nsk, skb)) {
                rsk = nsk;
                goto reset;
            }
            return 0;
        }
```

```
32      } else
33          sock_rps_save_rxhash(sk, skb);
34
35      if (tcp_rcv_state_process(sk, skb)) {
36          rsk = sk;
37          goto reset;
38      }
39      return 0;
40
41  reset:
42      tcp_v4_send_reset(rsk, skb);
43  discard:
44      kfree_skb(skb);
45      /* Be careful here. If this function gets more complicated and
46       * gcc suffers from register pressure on the x86, sk (in \%ebx)
47       * might be destroyed here. This current version compiles correctly,
48       * but you have been warned.
49       */
50      return 0;
51
52  csum_err:
53      TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_CSUMERRORS);
54      TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_INERRS);
55      goto discard;
56  }
```

首先，程序先基于伪首部累加和进行全包的校验和，判断包是否传输正确。

其次，程序会进行相应的 cookie 检查。

最后，程序会继续调用`tcp_rcv_state_process`函数处理接收到的 SYN 段。

3.2.2.3 tcp_v4_cookie_check

该函数如下：

```
1   static struct sock *tcp_v4_cookie_check(struct sock *sk, struct sk_buff *skb)
2   {
3   #ifdef CONFIG_SYN_COOKIES
4       const struct tcphdr *th = tcp_hdr(skb);
5
6       if (!th->syn)
7           sk = cookie_v4_check(sk, skb);
8   #endif
9       return sk;
10  }
```

一般情况下，当前 linux 内核都会定义`CONFIG_SYN_COOKIES`宏的，显然对于第一次握手的时候，接收到的确实是 syn 包，故而直接返回了 sk。

3.2.2.4 tcp_rcv_state_process

该函数位于`/net/ipv4/tcp_input.c`中。与第一次握手相关的代码如下：

```
1   /*
2    *  This function implements the receiving procedure of RFC 793 for
3    *  all states except ESTABLISHED and TIME_WAIT.
4    *  It's called from both tcp_v4_rcv and tcp_v6_rcv and should be
```

```
 *  address independent.
 */

int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct inet_connection_sock *icsk = inet_csk(sk);
    const struct tcphdr *th = tcp_hdr(skb);
    struct request_sock *req;
    int queued = 0;
    bool acceptable;

    tp->rx_opt.saw_tstamp = 0;  /*saw_tstamp 表示在最新的包上是否看到的时间戳选项 */

    switch (sk->sk_state) {
    /* 省略无关代码 */

                case TCP_LISTEN:
                    if (th->ack)
                        return 1;

                    if (th->rst)
                        goto discard;

                    if (th->syn) {
                        if (th->fin)
                            goto discard;
                        if (icsk->icsk_af_ops->conn_request(sk, skb) < 0)
                            return 1;

                        /* Now we have several options: In theory there is
                         * nothing else in the frame. KA9Q has an option to
                         * send data with the syn, BSD accepts data with the
                         * syn up to the [to be] advertised window and
                         * Solaris 2.1 gives you a protocol error. For now
                         * we just ignore it, that fits the spec precisely
                         * and avoids incompatibilities. It would be nice in
                         * future to drop through and process the data.
                         *
                         * Now that TTCP is starting to be used we ought to
                         * queue this data.
                         * But, this leaves one open to an easy denial of
                         * service attack, and SYN cookies can't defend
                         * against this problem. So, we drop the data
                         * in the interest of security over speed unless
                         * it's still in use.
                         */
                        kfree_skb(skb);
                        return 0;
                    }
                    goto discard;

                    /* 省略无关代码 */
discard:
        __kfree_skb(skb);
    }
    return 0;
}
```

显然，所接收到的包的 ack、rst、fin 字段都不为 1，故而这时开始进行连接检查，判

断是否可以允许连接。经过不断查找，我们发现icsk->icsk_af_ops->conn_request最终会掉用tcp_v4_conn_request进行处理。如果 syn 段合法，内核就会为该连接请求创建连接请求块，并且保存相应的信息。否则，就会返回 1, 原函数会发送 reset 给客户端表明连接请求失败。

当然，如果收到的包的 ack 字段为 1, 那么由于此时链接还未建立，故该包无效，返回 1, 并且调用该函数的函数会发送 reset 包给对方。如果收到的是 rst 字段或者既有 fin 又有 syn 的字段，那就直接销毁，并且释放内存。

3.2.2.5 tcp_v4_conn_request && tcp_conn_request

该函数位于/net/ipv4/tcp_ipv4/tcp_ipv4.c中，函数如下：

```
int tcp_v4_conn_request(struct sock *sk, struct sk_buff *skb)
{
    /* Never answer to SYNs send to broadcast or multicast */
    if (skb_rtable(skb)->rt_flags & (RTCF_BROADCAST | RTCF_MULTICAST))
        goto drop;

    return tcp_conn_request(&tcp_request_sock_ops,
                &tcp_request_sock_ipv4_ops, sk, skb);

drop:
    NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENDROPS);
    return 0;
}
```

如果一个 SYN 段是要被发送到广播地址和组播地址，则直接 drop 掉，然后返回 0。否则的话，就继续调用tcp_conn_request进行连接处理。

```
int tcp_conn_request(struct request_sock_ops *rsk_ops,
             const struct tcp_request_sock_ops *af_ops,
             struct sock *sk, struct sk_buff *skb)
{
    struct tcp_fastopen_cookie foc = { .len = -1 };                 //初始化 len 字段
    __u32 isn = TCP_SKB_CB(skb)->tcp_tw_isn;                         //tw???    isn: initial sequence n
    struct tcp_options_received tmp_opt;
    struct tcp_sock *tp = tcp_sk(sk);
    struct sock *fastopen_sk = NULL;
    struct dst_entry *dst = NULL;
    struct request_sock *req;
    bool want_cookie = false;                                                          //???
    struct flowi fl;                                                                          //路由查找

    /* TW buckets are converted to open requests without
     * limitations, they conserve resources and peer is
     * evidently real one.
     */
    if ((sysctl_tcp_syncookies == 2 ||
         inet_csk_reqsk_queue_is_full(sk)) && !isn) {
        want_cookie = tcp_syn_flood_action(sk, skb, rsk_ops->slab_name);
        if (!want_cookie)
            goto drop;
    }
```

首先,前面???如果 SYN 请求队列已满并且 isn 为 0, 然后通过函数 **tcp_syn_flood_action** 判断是否需要发送 syncookie。如果没有启用 syncookie 的话，就会返回 false，此时不能接收新的 SYN 请求，会将所收到的包丢掉。

```
    /* Accept backlog is full. If we have already queued enough
     * of warm entries in syn queue, drop request. It is better than
     * clogging syn queue with openreqs with exponentially increasing
     * timeout.
     */
    if (sk_acceptq_is_full(sk) && inet_csk_reqsk_queue_young(sk) > 1) {
        NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENOVERFLOWS);
        goto drop;
    }
```

warm entries

如果连接队列长度已经达到上限且 SYN 请求队列中至少有一个握手过程中没有重传过段，则丢弃当前请求。

```
    req = inet_reqsk_alloc(rsk_ops, sk, !want_cookie);
    if (!req)
        goto drop;
```

这时调用 reqsk_alloc() 分配一个连接请求块，用于保存连接请求信息，同时初始化在连接过程中用来发送 ACK/RST 段的操作集合，以便在建立连接过程中能方便地调用这些接口。

```
    tcp_rsk(req)->af_specific = af_ops;
```

这一步进行的是为了保护 BGP 会话。???

```
    tcp_rsk(req)->af_specific = af_ops;

    tcp_clear_options(&tmp_opt);
    tmp_opt.mss_clamp = af_ops->mss_clamp;
    tmp_opt.user_mss  = tp->rx_opt.user_mss;
    tcp_parse_options(skb, &tmp_opt, 0, want_cookie ? NULL : &foc);
```

之后,清除 TCP 选项后初始化 mss_vlamp 和 user_mss. 然后调用 tcp_parse_options 解析 SYN 段中的 TCP 选项，查看是否有相关的选项。

```
    if (want_cookie && !tmp_opt.saw_tstamp)
        tcp_clear_options(&tmp_opt);
```

如果启动了 syncookies，并且 TCP 段中没有存在时间戳（why，the reason?），则清除已经解析的 TCP 选项。

```
    tmp_opt.tstamp_ok = tmp_opt.saw_tstamp;
    tcp_openreq_init(req, &tmp_opt, skb, sk);
```

这时，根据收到的 SYN 段中的选项和序号来初始化连接请求块信息。

```
    /* Note: tcp_v6_init_req() might override ir_iif for link locals */
    inet_rsk(req)->ir_iif = sk->sk_bound_dev_if;

    af_ops->init_req(req, sk, skb);

    if (security_inet_conn_request(sk, skb, req))
        goto drop_and_free;
```

这一部分于 IPV6 以及安全检测有关，这里不进行详细讲解。安全检测失败的话，就会丢弃 SYN 段。

```
    if (!want_cookie && !isn) {
        /* VJ's idea. We save last timestamp seen
         * from the destination in peer table, when entering
         * state TIME-WAIT, and check against it before
         * accepting new connection request.
         *
         * If "isn" is not zero, this request hit alive
         * timewait bucket, so that all the necessary checks
         * are made in the function processing timewait state.
         */
        if (tcp_death_row.sysctl_tw_recycle) {
            bool strict;

            dst = af_ops->route_req(sk, &fl, req, &strict);

            if (dst && strict &&
                !tcp_peer_is_proven(req, dst, true,
                        tmp_opt.saw_tstamp)) {
                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PAWSPASSIVEREJECTED);
                goto drop_and_release;
            }
        }
        /* Kill the following clause, if you dislike this way. */
        else if (!sysctl_tcp_syncookies &&
             (sysctl_max_syn_backlog - inet_csk_reqsk_queue_len(sk) <
              (sysctl_max_syn_backlog >> 2)) &&
             !tcp_peer_is_proven(req, dst, false,
                         tmp_opt.saw_tstamp)) {
            /* Without syncookies last quarter of
             * backlog is filled with destinations,
             * proven to be alive.
             * It means that we continue to communicate
             * to destinations, already remembered
             * to the moment of synflood.
             */
            pr_drop_req(req, ntohs(tcp_hdr(skb)->source),
                    rsk_ops->family);
            goto drop_and_release;
        }

        isn = af_ops->init_seq(skb);
    }
```

如果没有开启 syncookie 并且 isn 为 0 的话，在距中的第一个 if 从对段信息块中获取时间戳，在新的连接请求之前检测 **PAWS**。后边的表明在没有启动 syncookies 的情况下受到 synflood 攻击，丢弃收到的段。之后由源地址，源端口，目的地址以及目的端口计算出服务端初始序列号。

```
    if (!dst) {
        dst = af_ops->route_req(sk, &fl, req, NULL);
        if (!dst)
            goto drop_and_free;
    }

    tcp_ecn_create_request(req, skb, sk, dst);

    if (want_cookie) {
        isn = cookie_init_sequence(af_ops, sk, skb, &req->mss);
        req->cookie_ts = tmp_opt.tstamp_ok;
        if (!tmp_opt.tstamp_ok)
            inet_rsk(req)->ecn_ok = 0;
    }

    tcp_rsk(req)->snt_isn = isn;
    tcp_rsk(req)->txhash = net_tx_rndhash();
    tcp_openreq_init_rwin(req, sk, dst);
    if (!want_cookie) {
        tcp_reqsk_record_syn(sk, req, skb);
        fastopen_sk = tcp_try_fastopen(sk, skb, req, &foc, dst);
    }
    if (fastopen_sk) {
        af_ops->send_synack(fastopen_sk, dst, &fl, req,
                    &foc, false);
        /* Add the child socket directly into the accept queue */
        inet_csk_reqsk_queue_add(sk, req, fastopen_sk);
        sk->sk_data_ready(sk);
        bh_unlock_sock(fastopen_sk);
        sock_put(fastopen_sk);
    } else {
        tcp_rsk(req)->tfo_listener = false;
        if (!want_cookie)
            inet_csk_reqsk_queue_hash_add(sk, req, TCP_TIMEOUT_INIT);
        af_ops->send_synack(sk, dst, &fl, req,
                    &foc, !want_cookie);
        if (want_cookie)
            goto drop_and_free;
    }
    reqsk_put(req);
    return 0;

drop_and_release:
    dst_release(dst);
drop_and_free:
    reqsk_free(req);
drop:
    NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENDROPS);
    return 0;
```

暂时不懂,,,, 等等在分析。。。。。。

3.2.2.6 inet_csk_reqsk_queue_add

```
struct sock *inet_csk_reqsk_queue_add(struct sock *sk,
                      struct request_sock *req,
                      struct sock *child)
{
```

```
    struct request_sock_queue *queue = &inet_csk(sk)->icsk_accept_queue;

    spin_lock(&queue->rskq_lock);
    if (unlikely(sk->sk_state != TCP_LISTEN)) {
        inet_child_forget(sk, req, child);
        child = NULL;
    } else {
        req->sk = child;
        req->dl_next = NULL;
        if (queue->rskq_accept_head == NULL)
            queue->rskq_accept_head = req;
        else
            queue->rskq_accept_tail->dl_next = req;
        queue->rskq_accept_tail = req;
        sk_acceptq_added(sk);
    }
    spin_unlock(&queue->rskq_lock);
    return child;
}
```

这一个函数所进行的操作就是直接将请求挂在接收队列中。

3.2.2.7 inet_csk_reqsk_queue_hash_add

```
void inet_csk_reqsk_queue_hash_add(struct sock *sk, struct request_sock *req,
                   unsigned long timeout)
{
    reqsk_queue_hash_req(req, timeout);
    inet_csk_reqsk_queue_added(sk);
}
```

首先将连接请求块保存到父传输请求块的散列表中，并设置定时器超时时间。之后更新已存在的连接请求块数，并启动连接建立定时器。

3.2.3 第二次握手：发送 SYN+ACK 段

在第一次握手的最后调用了 af_ops->send_synack 函数，而该函数最终会调用 tcp_v4_send_synack 函数进行发送，故而这里我们这里就从这个函数进行分析。

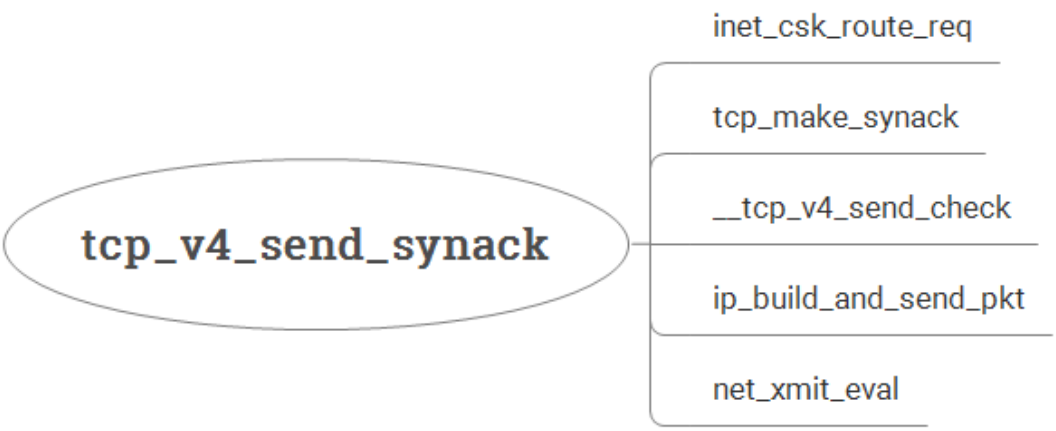
3.2.3.1 第二次函数调用关系

第二次握手的调用函数关系图如下：

3.2.3.2 tcp_v4_send_synack

```
/*
 *  Send a SYN-ACK after having received a SYN.
 *  This still operates on a request_sock only, not on a big
 *  socket.
 */
static int tcp_v4_send_synack(const struct sock *sk, struct dst_entry *dst,
                  struct flowi *fl,
                  struct request_sock *req,
                  struct tcp_fastopen_cookie *foc,
```

Second Shake Hand of Server.png Second Shake Hand of Server.bb




```
                    bool attach_req)
{
    const struct inet_request_sock *ireq = inet_rsk(req);
    struct flowi4 fl4;
    int err = -1;
    struct sk_buff *skb;

    /* First, grab a route. */
    if (!dst && (dst = inet_csk_route_req(sk, &fl4, req)) == NULL)
        return -1;

    skb = tcp_make_synack(sk, dst, req, foc, attach_req);

    if (skb) {
        __tcp_v4_send_check(skb, ireq->ir_loc_addr, ireq->ir_rmt_addr);

        err = ip_build_and_send_pkt(skb, sk, ireq->ir_loc_addr,
                    ireq->ir_rmt_addr,
                    ireq->opt);
        err = net_xmit_eval(err);
    }

    return err;
}
```

首先，如果传进来的 dst 为空或者根据连接请求块中的信息查询路由表，如果没有查到，那么就直接推出。

否则就跟据当前的传输控制块，路由信息，请求等信息构建 syn+ack 段。

如果构建成功的话，就生成 TCP 校验码，然后调用 ip_build_and_send_pkt 生成 IP 数据报并且发送出去。

net_xmit_eval 是什么，待考虑。

3.2.3.3 tcp_make_synack

该函数用来构造一个 SYN+ACK 段，并初始化 TCP 首部及 SKB 中的各字段项，填入相应的选项，如 MSS，SACK，窗口扩大银子，时间戳等。函数如下：

```
/**
 * tcp_make_synack - Prepare a SYN-ACK.
 * sk: listener socket
 * dst: dst entry attached to the SYNACK
 * req: request_sock pointer
 *
 * Allocate one skb and build a SYNACK packet.
 * @dst is consumed : Caller should not use it again.
 */
struct sk_buff *tcp_make_synack(const struct sock *sk, struct dst_entry *dst,
                struct request_sock *req,
                struct tcp_fastopen_cookie *foc,
                bool attach_req)
{
    struct inet_request_sock *ireq = inet_rsk(req);
    const struct tcp_sock *tp = tcp_sk(sk);
    struct tcp_md5sig_key *md5 = NULL;
    struct tcp_out_options opts;
    struct sk_buff *skb;
    int tcp_header_size;
    struct tcphdr *th;
    u16 user_mss;
    int mss;

    skb = alloc_skb(MAX_TCP_HEADER, GFP_ATOMIC);
    if (unlikely(!skb)) {
        dst_release(dst);
        return NULL;
    }
```

首先为将要发送的数据申请发送缓存，**unlikely 函数待分析???** ，如果没有申请到，那就会返回 NULL。

```
    /* Reserve space for headers. */
    skb_reserve(skb, MAX_TCP_HEADER);
```

为 MAC 层，IP 层，TCP 层首部预留必要的空间。

```
    if (attach_req) {
        skb_set_owner_w(skb, req_to_sk(req));
    } else {
        /* sk is a const pointer, because we want to express multiple
         * cpu might call us concurrently.
         * sk->sk_wmem_alloc in an atomic, we can promote to rw.
         */
        skb_set_owner_w(skb, (struct sock *)sk);
    }
    skb_dst_set(skb, dst);
```

根据attach_req来判断该执行如何执行相关操作 **to do in the future**。然后设置发送缓存的目的路由 **a little confused, why need this**。

```
    mss = dst_metric_advmss(dst);
    user_mss = READ_ONCE(tp->rx_opt.user_mss);
    if (user_mss && user_mss < mss)
        mss = user_mss;
```

根据每一个路由器上的 mss 以及自身的 mss 来得到最大的 mss。

```
    memset(&opts, 0, sizeof(opts));
#ifdef CONFIG_SYN_COOKIES
    if (unlikely(req->cookie_ts))
        skb->skb_mstamp.stamp_jiffies = cookie_init_timestamp(req);
    else
#endif
    skb_mstamp_get(&skb->skb_mstamp);
```

清除选项，并且设置相关时间戳 **to add in future**。

```
#ifdef CONFIG_TCP_MD5SIG
    rcu_read_lock();
    md5 = tcp_rsk(req)->af_specific->req_md5_lookup(sk, req_to_sk(req));
#endif
```

查看是否有 MD5 选项，有的话构造出相应的 md5.

```
    skb_set_hash(skb, tcp_rsk(req)->txhash, PKT_HASH_TYPE_L4);
    tcp_header_size = tcp_synack_options(req, mss, skb, &opts, md5, foc) +
              sizeof(*th);

    skb_push(skb, tcp_header_size);
    skb_reset_transport_header(skb);
```

得到 tcp 的头部大小，然后进行大小设置，并且重置传输层的头部。

```
    th = tcp_hdr(skb);
    memset(th, 0, sizeof(struct tcphdr));
    th->syn = 1;
    th->ack = 1;
    tcp_ecn_make_synack(req, th);
    th->source = htons(ireq->ir_num);
    th->dest = ireq->ir_rmt_port;
```

清空 tcp 头部，并设置 tcp 头部的各个字段。

```
    /* Setting of flags are superfluous here for callers (and ECE is
     * not even correctly set)
     */
    tcp_init_nondata_skb(skb, tcp_rsk(req)->snt_isn,
                 TCPHDR_SYN | TCPHDR_ACK);

    th->seq = htonl(TCP_SKB_CB(skb)->seq);
    /* XXX data is queued and acked as is. No buffer/window check */
    th->ack_seq = htonl(tcp_rsk(req)->rcv_nxt);

    /* RFC1323: The window in SYN & SYN/ACK segments is never scaled. */
    th->window = htons(min(req->rsk_rcv_wnd, 65535U));
    tcp_options_write((__be32 *)(th + 1), NULL, &opts);
    th->doff = (tcp_header_size >> 2);
    TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_OUTSEGS);
```

首先初始化不含数据的 tcp 报文，然后设置相关的序列号，确认序列号，窗口大小，选项字段，以及 TCP 数据偏移，之所以除以，是因为稿子短的单位时 32 位字，即以四个字节长的字为计算单位。

```
#ifdef CONFIG_TCP_MD5SIG
    /* Okay, we have all we need - do the md5 hash if needed */
    if (md5)
        tcp_rsk(req)->af_specific->calc_md5_hash(opts.hash_location,
                        md5, req_to_sk(req), skb);
    rcu_read_unlock();
#endif

    /* Do not fool tcpdump (if any), clean our debris */
    skb->tstamp.tv64 = 0;
    return skb;
}
```

最后判断是否需要 md5 哈希值，如果需要的话，就进行添加。**what does it mean in the middle?**，然后返回生成包含 SYN+ACK 段的 skb。

3.2.4 第三次握手：接收 ACK 段

在服务器第二次我受的最后启动了建立连接定时器，等待客户端最后一次握手的 ACK 段。

3.2.4.1 第三次握手函数调用关系图

3.2.4.2 tcp_v4_do_rcv

```
/* The socket must have it's spinlock held when we get
 * here, unless it is a TCP_LISTEN socket.
 *
 * We have a potential double-lock case here, so even when
 * doing backlog processing we use the BH locking scheme.
 * This is because we cannot sleep with the original spinlock
 * held.
 */
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
    struct sock *rsk;

    /** 省略无关代码 **/

    if (tcp_checksum_complete(skb))
        goto csum_err;

    if (sk->sk_state == TCP_LISTEN) {
        struct sock *nsk = tcp_v4_cookie_check(sk, skb);

        if (!nsk)
            goto discard;
        if (nsk != sk) {
            sock_rps_save_rxhash(nsk, skb);
            sk_mark_napi_id(nsk, skb);
            if (tcp_child_process(sk, nsk, skb)) {
                rsk = nsk;
                goto reset;
            }
            return 0;
        }
```

```
32      } else
33          sock_rps_save_rxhash(sk, skb);
34      /** 省略无关代码 **/
35  reset:
36      tcp_v4_send_reset(rsk, skb);
37  discard:
38      kfree_skb(skb);
39      /* Be careful here. If this function gets more complicated and
40       * gcc suffers from register pressure on the x86, sk (in %ebx)
41       * might be destroyed here. This current version compiles correctly,
42       * but you have been warned.
43       */
44      return 0;
45
46  csum_err:
47      TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_CSUMERRORS);
48      TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_INERRS);
49      goto discard;
50  }
```

在服务器最后一次握手的时候，其实传输控制块仍然处于 LISTEN 状态，但是这时候 cookie 检查得到的传输控制块已经不是侦听传输控制快了，故而会执行`tcp_child_process`来初始化子传输控制块。如果初始化失败的话（返回值非零），就会给客户端发送 RST 段进行复位。

3.2.4.3 tcp_v4_cookie_check

```
1   static struct sock *tcp_v4_cookie_check(struct sock *sk, struct sk_buff *skb)
2   {
3   #ifdef CONFIG_SYN_COOKIES
4       const struct tcphdr *th = tcp_hdr(skb);
5
6       if (!th->syn)
7           sk = cookie_v4_check(sk, skb);
8   #endif
9       return sk;
10  }
```

在现在的 Linux 内核中一般都会定义`CONFIG_SYN_COOKIES`宏，此时在第三次握手阶段，并不是 syn 包，内核就会执行`cookie_v4_check`。在这个函数中，服务器会将客户端的 ACK 序列号减去 1，得到 cookie 比较值，然后将客户端的 IP 地址，客户端端口，服务器 IP 地址和服务器端口，接收到的 TCP 序列好以及其它一些安全数值等要素进行 hash 运算后，与该 cookie 比较值比较，如果相等，则直接完成三次握手，此时不必查看该连接是否属于请求连接队列。

3.2.4.4 tcp_child_process

子传输控制块开始处理 TCP 段。

```
1   /*
2    * Queue segment on the new socket if the new socket is active,
3    * otherwise we just shortcircuit this and continue with
4    * the new socket.
```

```
 *
 * For the vast majority of cases child->sk_state will be TCP_SYN_RECV
 * when entering. But other states are possible due to a race condition
 * where after __inet_lookup_established() fails but before the listener
 * locked is obtained, other packets cause the same connection to
 * be created.
 */

int tcp_child_process(struct sock *parent, struct sock *child,
              struct sk_buff *skb)
{
    int ret = 0;
    int state = child->sk_state;

    tcp_sk(child)->segs_in += max_t(u16, 1, skb_shinfo(skb)->gso_segs);
    if (!sock_owned_by_user(child)) {
        ret = tcp_rcv_state_process(child, skb);
        /* Wakeup parent, send SIGIO */
        if (state == TCP_SYN_RECV && child->sk_state != state)
            parent->sk_data_ready(parent);
    } else {
        /* Alas, it is possible again, because we do lookup
         * in main socket hash table and lock on listening
         * socket does not protect us more.
         */
        __sk_add_backlog(child, skb);
    }

    bh_unlock_sock(child);
    sock_put(child);
    return ret;
}
```

该函数位于tcp_minisocks.c中。

首先，如果此时刚刚创建的新的子传输控制块没有被用户进程占用，则根据作为第三次握手的 ACK 段，调用tcp_rcv_state_process继续对子传输控制块做初始化。否则的话，只能将其加入后备队列中，等空闲时再进行处理。虽然这种情况出现的概率小，但是也是有可能发生的。

3.2.4.5 tcp_rcv_state_process

该函数用来处理 ESTABLISHED 和TIME_WAIT状态以外的 TCP 段,这里处理SYN_RECV状态。

```
    acceptable = tcp_ack(sk, skb, FLAG_SLOWPATH |
                         FLAG_UPDATE_TS_RECENT) > 0;
```

首先对收到的 ACK 段进行处理判断是否正确接收，如果正确接收就会发送返回非零值。

```
    switch (sk->sk_state) {
        case TCP_SYN_RECV:
            if (!acceptable)
                return 1;

```

```
6               if (!tp->srtt_us)
7                   tcp_synack_rtt_meas(sk, req);
8
9               /* Once we leave TCP_SYN_RECV, we no longer need req
10               * so release it.
11               */
12              if (req) {
13                  tp->total_retrans = req->num_retrans;
14                  reqsk_fastopen_remove(sk, req, false);
15              } else {
16                  /* Make sure socket is routed, for correct metrics. */
17                  icsk->icsk_af_ops->rebuild_header(sk);
18                  tcp_init_congestion_control(sk);
19
20                  tcp_mtup_init(sk);
21                  tp->copied_seq = tp->rcv_nxt;
22                  tcp_init_buffer_space(sk);
23              }
24              smp_mb();
25              tcp_set_state(sk, TCP_ESTABLISHED);
26              sk->sk_state_change(sk);
```

进行一系列的初始化，开启相应拥塞控制等，并且将 TCP 的状态置为 ESTABLISHED。

```
1               /* Note, that this wakeup is only for marginal crossed SYN case.
2                * Passively open sockets are not waked up, because
3                * sk->sk_sleep == NULL and sk->sk_socket == NULL.
4                */
5               if (sk->sk_socket)
6                   sk_wake_async(sk, SOCK_WAKE_IO, POLL_OUT);
```

发信号给那些将通过该套接口发送数据的进程，通知它们套接口目前已经可以发送数据了。

```
1               tp->snd_una = TCP_SKB_CB(skb)->ack_seq;
2               tp->snd_wnd = ntohs(th->window) << tp->rx_opt.snd_wscale;
3               tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);
4
5               if (tp->rx_opt.tstamp_ok)
6                   tp->advmss -= TCPOLEN_TSTAMP_ALIGNED;
```

初始化传输控制块的各个字段，对时间戳进行处理。

```
1               if (req) {
2                   /* Re-arm the timer because data may have been sent out.
3                    * This is similar to the regular data transmission case
4                    * when new data has just been ack'ed.
5                    *
6                    * (TFO) - we could try to be more aggressive and
7                    * retransmitting any data sooner based on when they
8                    * are sent out.
9                    */
10                  tcp_rearm_rto(sk);
11              } else
12                  tcp_init_metrics(sk);
```

为该套接口初始化路由。

```
            tcp_update_pacing_rate(sk);

            /* Prevent spurious tcp_cwnd_restart() on first data packet */
            tp->lsndtime = tcp_time_stamp;

            tcp_initialize_rcv_mss(sk);
            tcp_fast_path_on(tp);
            break;
```

更新最近一次的发送数据报的时间，初始化与路径 MTU 有关的成员，并计算有关 TCP 首部预测的标志。

```
    /* step 6: check the URG bit */
    tcp_urg(sk, skb, th);
```

检测带外数据标志位。

```
    /* step 7: process the segment text */
    switch (sk->sk_state) {
        /** 省略无关代码 **/
        case TCP_ESTABLISHED:
            tcp_data_queue(sk, skb);
            queued = 1;
            break;
    }
```

对已接收到的 TCP 段排队，在建立连接阶段一般不会收到 TCP 段。

```
    /* tcp_data could move socket to TIME-WAIT */
    if (sk->sk_state != TCP_CLOSE) {
        tcp_data_snd_check(sk);
        tcp_ack_snd_check(sk);
    }

    if (!queued) {
discard:
        __kfree_skb(skb);
    }
    return 0;
```

显然此时状态不为 CLOSE，故而就回去检测是否数据和 ACK 要发送。其次，根据 queue 标志来确定是否释放接收到的 TCP 段，如果接收到的 TCP 段已添加到接收队列中，则不释放。

CHAPTER 4

非核心函数分析

Contents

4.1 SKB

4.1.1 skb_transport_header

该函数位于/include/linux/skbuff.h中，其功能是根据 skb 得到其 tcphdr 的偏移。

```
1 static inline unsigned char *skb_transport_header(const struct sk_buff *skb)
2 {
3     return skb->head + skb->transport_header;
4 }
```

其中，skb->head指向缓存区的头部。skb->transport_heade为传输层相对于缓冲区头部的偏移。

4.2 Inet

4.2.1 inet_csk

该函数位于/include/net/inet_connection_sock.h中, 主要的目的就是进行类型转换，将一个 sock 类型的变量转为inet_connection_sock。**问题待解决。**

```
static inline struct inet_connection_sock *inet_csk(const struct sock *sk)
{
    return (struct inet_connection_sock *)sk;
}
```

4.3 TCP 层

4.3.1 tcp_hdr

该函数位于/include/linux/tcp.h中,主要目的就是将sk_buff类型的变量转化为tcphdr;

```
static inline struct tcphdr *tcp_hdr(const struct sk_buff *skb)
{
    return (struct tcphdr *)skb_transport_header(skb);
}
```

TCP_INC_STATS_BH

rcu_read_unlock 出现在tcp_make_synack中于 MD5 相关的部分。

net_xmit_eval 定时器??

4.3.2 tcp_init_nondata_skb

该函数提供了初始化不含数据的 skb 的功能。函数原型如下:

```
tcp_init_nondata_skb(struct sk_buff *skb, u32 seq, u8 flags);
```

skb 待初始化的sk_buff。

seq 序号

flags 标志位

```
static void tcp_init_nondata_skb(struct sk_buff *skb, u32 seq, u8 flags)
{
        /* 设置校验码 */
        skb->ip_summed = CHECKSUM_PARTIAL;
        skb->csum = 0;

        /* 设置标志位 */
        TCP_SKB_CB(skb)->tcp_flags = flags;
        TCP_SKB_CB(skb)->sacked = 0;

        tcp_skb_pcount_set(skb, 1);

        /* 设置起始序号 */
```

```
14          TCP_SKB_CB(skb)->seq = seq;
15          if (flags & (TCPHDR_SYN | TCPHDR_FIN))
16                  seq++;
17          TCP_SKB_CB(skb)->end_seq = seq;
18  }
```

4.3.3 before() 和 after()

在一些需要判断序号前后的地方出现了before()和after() 这两个函数。这两个函数的定义如下

```
1  /* include/net/tcp.h
2   * 比较两个无符号 32 位整数
3   */
4  static inline bool before(__u32 seq1, __u32 seq2)
5  {
6          return (__s32)(seq1-seq2) < 0;
7  }
8  #define after(seq2, seq1)       before(seq1, seq2)
```

可以看到，这两个函数实际上就是将两个数直接相减。之所以要单独弄个函数应该是为了避免强制转型造成影响。序号都是 32 位无符号整型。

4.4 辅助函数

有些小函数是用于辅助一些很底层的功能的，这里单独列出来。

4.4.1 分支预测优化

现代处理器均为流水线结构。而分支语句可能导致流水线断流。因此，很多处理器均有分支预测的功能。然而，分支预测失败所导致的惩罚也是相对高昂的。为了提升性能，Linux 的很多分支判断中都使用了 likely()和unlikely()这组宏定义来人工指示编译器，哪些分支出现的概率极高，以便编译器进行优化。

这里有两种定义，一种是开启了分支语句分析相关的选项时，内核会采用下面的一种定义

```
1  /* include/linux/compiler.h
2   * 采用 __builtin_constant_p(x) 来忽略常量表达式。
3   */
4  # ifndef likely
5  #  define likely(x)     (__builtin_constant_p(x) ? !!(x) : __branch_check__(x, 1))
6  # endif
7  # ifndef unlikely
8  #  define unlikely(x)   (__builtin_constant_p(x) ? !!(x) : __branch_check__(x, 0))
9  # endif
```

在该定义下，__branch_check__用于跟踪分支结果并更新统计数据。

如果不开启该选项，则定义得较为简单：

```
# define likely(x)      __builtin_expect(!!(x), 1)
# define unlikely(x)    __builtin_expect(!!(x), 0)
```

其中__builtin_expect是 GCC 的内置函数，用于指示编译器，该条件语句最可能的结果是什么。

4.4.2 字节序

CPU 分为大端和小端两种。而在网络传输的过程中，大小端的不一致会带来问题。因此，网络协议中对于字节序都有明确规定。一般采用大端序。

Linux 中，对于这一部分的支持放在了include/linux/byteorder/generic.h 中。而实现，则交由体系结构相关的代码来完成。

```
/* 下面的函数用于进行对 16 位整型或者 32 位整型在网络传输格式和本地格式之间的转换。
 */
ntohl(__u32 x)
ntohs(__u16 x)
htonl(__u32 x)
htons(__u16 x)
```

上面函数的命名规则是末尾的 l 代表 32 位，s 代表 16 位。n 代表 network，h 代表 host。根据命名规则，不难知道函数的用途。比如 htons 就是从本地的格式转换的网络传输用的格式，转换的是 16 位整数。

CHAPTER 5

附录: 基础知识

5.1 C 语言

5.1.1 结构体初始化

```
typedef struct{
        int  a;
        char ch;
}flag;
/*
        目的是将 a 初始化为 1,ch 初始化为'u'.
*/
/* 法一：分别初始化 */
flag tmp;
tmp.a=1;
tmp.ch='u';

/* 法二：点式初始化 */
flag tmp={.a=1,.ch='u'};                //注意两个变量之间使用 , 而不是;
/* 法三：*/
flag tmp={
                        a:1,
                        ch:'u'
                };
```

当然，我们也可以使用上述任何一种方法只对结构体中的某几个变量进行初始化。

5.1.2 位字段

在存储空间极为宝贵的情况下，有可能需要将多个对象保存在一个机器字中。而在 linux 开发的早期，那时确实空间极其宝贵。于是乎，那一帮黑客们就发明了各种各样的办法。一种常用的办法是使用类似于编译器符号表的单个二进制位标志集合，即定义一系列的 2 的指数次方的数，此方法确实有效。但是，仍然十分浪费空间。而且有可能很多位都利用不到。于是乎，他们提出了另一种新的思路即位字段。我们可以利用如下方式定义一个包含 3 位的变量。

```
struct {
        unsigned int a:1;
        unsigned int b:1;
        unsigned int c:1;
}flags;
```

字段可以不命名，无名字段，即只有一个冒号和宽度，起到填充作用。特殊宽度 0 可以用来强制在下一个字边界上对齐，一般位于结构体的尾部。

冒号后面表示相应字段的宽度 (二进制宽度)，即不一定非得是 1 位。字段被声明为unsigned int类型，以确保它们是无符号量。

当然我们需要注意，机器是分大端和小端存储的。因此，我们在选择外部定义数据的情况下爱，必须仔细考虑那一端优先的问题。同时，字段不是数组，并且没有地址，因此不能对它们使用&运算符。

5.2 操作系统

5.3 GNU C

5.3.1 __attribute__

GNU C 的一大特色就是__attribute__机制。__attribute__可以设置函数属性 (Function Attribute)、变量属性 (Variable Attribute) 和类型属性 (Type Attribute)。

__attribute__的书写特征是: __attribute__前后都有两个下划线，并切后面会紧跟一对原括弧，括弧里面是相应的__attribute__参数。

__attribute__的语法格式为: __attribute__((attribute-list))

__attribute__的位置约束为:放于声明的尾部";"之前。参考博客:http://www.cnblogs.com/astwish/p/
关键字__attribute__也可以对结构体 (struct) 或共用体 (union) 进行属性设置。大致有六个参数值可以被设定，即: aligned, packed, transparent_union, unused, deprecated 和 may_alias 。

在使用__attribute__参数时，你也可以在参数的前后都加上两个下划线，例如，使用__aligned__而不是 aligned ，这样，你就可以在相应的头文件里使用它而不用关心头文件里是否有重名的宏定义。aligned (alignment) 该属性设定一个指定大小的对齐格式 (以字节为单位)。

5.3.1.1 设置变量属性

下面的声明将强制编译器确保 (尽它所能) 变量类型为int32_t的变量在分配空间时采用 8 字节对齐方式。

```
typedef int int32_t __attribute__ ((aligned(8)));
```

5.3.1.2 设置类型属性

下面的声明将强制编译器确保 (尽它所能) 变量类型为 struct S 的变量在分配空间时采用 8 字节对齐方式。

```
struct S {
        short b[3];
} __attribute__ ((aligned (8)));
```

如上所述，你可以手动指定对齐的格式，同样，你也可以使用默认的对齐方式。如果 aligned 后面不紧跟一个指定的数字值，那么编译器将依据你的目标机器情况使用最大最有益的对齐方式。例如：

```
struct S {
        short b[3];
} __attribute__ ((aligned));
```

这里，如果 sizeof（short）的大小为 2（byte），那么，S 的大小就为 6 。取一个 2 的次方值，使得该值大于等于 6 ，则该值为 8，所以编译器将设置 S 类型的对齐方式为 8 字节。aligned 属性使被设置的对象占用更多的空间，相反的，使用 packed 可以减小对象占用的空间。需要注意的是，attribute 属性的效力与你的连接器也有关，如果你的连接器最大只支持 16 字节对齐，那么你此时定义 32 字节对齐也是无济于事的。

5.3.1.3 设置函数属性