<https://blog.csdn.net/wangpengqi/article/details/9156083>

sk\_buff是Linux网络协议栈最重要的数据结构之一，该数据结构贯穿于整个数据包处理的流程。由于协议采用分层结构，上层向下层传递数据时需要增加包头，下层向上层数据时又需要去掉包头。sk\_buff中保存了L2，L3，L4层的头指针，这样在层传递时只需要对数据缓冲区改变头部信息，并调整sk\_buff中的指针，而不需要拷贝数据，这样大大减少了内存拷贝的需要。

**[cpp]** [view plain](https://blog.csdn.net/wangpengqi/article/details/9156083) [copy](https://blog.csdn.net/wangpengqi/article/details/9156083)

1. /\*\*
2. \*  struct sk\_buff - socket buffer
3. \*  @next: Next buffer in list
4. \*  @prev: Previous buffer in list
5. \*  @tstamp: Time we arrived
6. \*  @sk: Socket we are owned by
7. \*  @dev: Device we arrived on/are leaving by
8. \*  @cb: Control buffer. Free for use by every layer. Put private vars here
9. \*  @\_skb\_refdst: destination entry (with norefcount bit)
10. \*  @sp: the security path, used for xfrm
11. \*  @len: Length of actual data
12. \*  @data\_len: Data length
13. \*  @mac\_len: Length of link layer header
14. \*  @hdr\_len: writable header length of cloned skb
15. \*  @csum: Checksum (must include start/offset pair)
16. \*  @csum\_start: Offset from skb->head where checksumming should start
17. \*  @csum\_offset: Offset from csum\_start where checksum should be stored
18. \*  @priority: Packet queueing priority
19. \*  @local\_df: allow local fragmentation
20. \*  @cloned: Head may be cloned (check refcnt to be sure)
21. \*  @ip\_summed: Driver fed us an IP checksum
22. \*  @nohdr: Payload reference only, must not modify header
23. \*  @nfctinfo: Relationship of this skb to the connection
24. \*  @pkt\_type: Packet class
25. \*  @fclone: skbuff clone status
26. \*  @ipvs\_property: skbuff is owned by ipvs
27. \*  @peeked: this packet has been seen already, so stats have been
28. \*      done for it, don't do them again
29. \*  @nf\_trace: netfilter packet trace flag
30. \*  @protocol: Packet protocol from driver
31. \*  @destructor: Destruct function
32. \*  @nfct: Associated connection, if any
33. \*  @nfct\_reasm: netfilter conntrack re-assembly pointer
34. \*  @nf\_bridge: Saved data about a bridged frame - see br\_netfilter.c
35. \*  @skb\_iif: ifindex of device we arrived on
36. \*  @tc\_index: Traffic control index
37. \*  @tc\_verd: traffic control verdict
38. \*  @rxhash: the packet hash computed on receive
39. \*  @queue\_mapping: Queue mapping for multiqueue devices
40. \*  @ndisc\_nodetype: router type (from link layer)
41. \*  @ooo\_okay: allow the mapping of a socket to a queue to be changed
42. \*  @l4\_rxhash: indicate rxhash is a canonical 4-tuple hash over transport
43. \*      ports.
44. \*  @wifi\_acked\_valid: wifi\_acked was set
45. \*  @wifi\_acked: whether frame was acked on wifi or not
46. \*  @no\_fcs:  Request NIC to treat last 4 bytes as Ethernet FCS
47. \*  @dma\_cookie: a cookie to one of several possible DMA operations
48. \*      done by skb DMA functions
49. \*  @secmark: security marking
50. \*  @mark: Generic packet mark
51. \*  @dropcount: total number of sk\_receive\_queue overflows
52. \*  @vlan\_tci: vlan tag control information
53. \*  @inner\_transport\_header: Inner transport layer header (encapsulation)
54. \*  @inner\_network\_header: Network layer header (encapsulation)
55. \*  @transport\_header: Transport layer header
56. \*  @network\_header: Network layer header
57. \*  @mac\_header: Link layer header
58. \*  @tail: Tail pointer
59. \*  @end: End pointer
60. \*  @head: Head of buffer
61. \*  @data: Data head pointer
62. \*  @truesize: Buffer size
63. \*  @users: User count - see {datagram,tcp}.c
64. \*/
66. **struct** sk\_buff {
67. /\* These two members must be first. \*/
68. **struct** sk\_buff      \*next;
69. **struct** sk\_buff      \*prev;
71. ktime\_t         tstamp;
73. **struct** sock     \*sk;
74. **struct** net\_device   \*dev;
76. /\*
77. \* This is the control buffer. It is free to use for every
78. \* layer. Please put your private variables there. If you
79. \* want to keep them across layers you have to do a skb\_clone()
80. \* first. This is owned by whoever has the skb queued ATM.
81. \*/
82. **char**            cb[48] \_\_aligned(8);
84. unsigned **long**       \_skb\_refdst;
85. #ifdef CONFIG\_XFRM
86. **struct**  sec\_path    \*sp;
87. #endif
88. unsigned **int**        len,
89. data\_len;
90. \_\_u16           mac\_len,
91. hdr\_len;
92. **union** {
93. \_\_wsum      csum;
94. **struct** {
95. \_\_u16   csum\_start;
96. \_\_u16   csum\_offset;
97. };
98. };
99. \_\_u32           priority;
100. kmemcheck\_bitfield\_begin(flags1);
101. \_\_u8            local\_df:1,
102. cloned:1,
103. ip\_summed:2,
104. nohdr:1,
105. nfctinfo:3;
106. \_\_u8            pkt\_type:3,
107. fclone:2,
108. ipvs\_property:1,
109. peeked:1,
110. nf\_trace:1;
111. kmemcheck\_bitfield\_end(flags1);
112. \_\_be16          protocol;
114. **void**            (\*destructor)(**struct** sk\_buff \*skb);
115. #if defined(CONFIG\_NF\_CONNTRACK) || defined(CONFIG\_NF\_CONNTRACK\_MODULE)
116. **struct** nf\_conntrack \*nfct;
117. #endif
118. #ifdef NET\_SKBUFF\_NF\_DEFRAG\_NEEDED
119. **struct** sk\_buff      \*nfct\_reasm;
120. #endif
121. #ifdef CONFIG\_BRIDGE\_NETFILTER
122. **struct** nf\_bridge\_info   \*nf\_bridge;
123. #endif
125. **int**         skb\_iif;
127. \_\_u32           rxhash;
129. \_\_u16           vlan\_tci;
131. #ifdef CONFIG\_NET\_SCHED
132. \_\_u16           tc\_index;   /\* traffic control index \*/
133. #ifdef CONFIG\_NET\_CLS\_ACT
134. \_\_u16           tc\_verd;    /\* traffic control verdict \*/
135. #endif
136. #endif
138. \_\_u16           queue\_mapping;
139. kmemcheck\_bitfield\_begin(flags2);
140. #ifdef CONFIG\_IPV6\_NDISC\_NODETYPE
141. \_\_u8            ndisc\_nodetype:2;
142. #endif
143. \_\_u8            pfmemalloc:1;
144. \_\_u8            ooo\_okay:1;
145. \_\_u8            l4\_rxhash:1;
146. \_\_u8            wifi\_acked\_valid:1;
147. \_\_u8            wifi\_acked:1;
148. \_\_u8            no\_fcs:1;
149. \_\_u8            head\_frag:1;
150. /\* Encapsulation protocol and NIC drivers should use
151. \* this flag to indicate to each other if the skb contains
152. \* encapsulated packet or not and maybe use the inner packet
153. \* headers if needed
154. \*/
155. \_\_u8            encapsulation:1;
156. /\* 7/9 bit hole (depending on ndisc\_nodetype presence) \*/
157. kmemcheck\_bitfield\_end(flags2);
159. #ifdef CONFIG\_NET\_DMA
160. dma\_cookie\_t        dma\_cookie;
161. #endif
162. #ifdef CONFIG\_NETWORK\_SECMARK
163. \_\_u32           secmark;
164. #endif
165. **union** {
166. \_\_u32       mark;
167. \_\_u32       dropcount;
168. \_\_u32       reserved\_tailroom;
169. };
171. sk\_buff\_data\_t      inner\_transport\_header;
172. sk\_buff\_data\_t      inner\_network\_header;
173. sk\_buff\_data\_t      transport\_header;
174. sk\_buff\_data\_t      network\_header;
175. sk\_buff\_data\_t      mac\_header;
176. /\* These elements must be at the end, see alloc\_skb() for details.  \*/
177. sk\_buff\_data\_t      tail;
178. sk\_buff\_data\_t      end;
179. unsigned **char**       \*head,
180. \*data;
181. unsigned **int**        truesize;
182. atomic\_t        users;
183. };

struct sk\_buff {//介绍   
    struct sk\_buff \*next \*prev;//双向链表指针  
    ktime\_t tstamp ;//时间撮  
    struct sock \*sk;   //对应于传输层，标示属于哪个socket ？  
    struct net\_device \*dev;    //数据来自或者发送自哪个设备  
    char cb[48];//控制信息buffer，在每个层都可以用，并且目前为止足够大  
    int len;      实际总长度  
    int data\_len; 数据的长度 //也许是paged的data   
    \_\_u16 mac\_len; 数据链路层头的长度  
    \_\_u16 hdr\_len; writable header length of cloned skb   
      
     sk\_buff\_data\_t   transport\_header;   传输层头指针  
    sk\_buff\_data\_t   network\_header;    网络层头指针  
    sk\_buff\_data\_t   mac\_header;        数据链路层头  
  
    unsigned char \*head; //buffer 头  
    unsigned char \*data; 数据头  
    sk\_buff\_data\_t tail; 数据结尾  
    sk\_buff\_data\_t end;  buffer 结尾  
    unsigned int truesize; //bufffer 大小  
  
    cloned 是不是cloned  
    mark 数据包mark  
    destructor 销毁函数指针   
    pkt\_type : 根据二层头确定的包信息  
    \_\_be16 protocol ： 三层协议 IP ARP 等，用于和全局数组qtype\_base中的数据对比，该数组可以通过dev\_add\_pack()注册.  
}  
  
由于该结构将用于各个层，内核提供了一系列的sk\_buff的操作函数  
skb\_put()  减小tailroom，buffer下后扩展  
skb\_push() 减小headroom，buffer向上扩张   
skb\_trim() cut buffer到一个长度  
skb\_pull   从数据头cut一定长度的数据   
skb\_reserve 增大headroom，减少tailroom，只能用于buffer为空时  
skb\_headroom headroom的大小  
skb\_tailroom tailroom的太小  
  
alloc\_skb() 分配一个sk\_buff结构及buffer区域  
kfree\_slb() reference 减一，并且free skb和buffer如果不再有引用  
  
dev\_alloc\_skb() 方便接收数据的sk\_buff的分配函数  
dev\_free\_skb()    
  
skb\_shinfo() 获得和sk\_buff 一块分配的struct skb\_shared\_info   
  
skb\_clone() //复制sk\_buff ,但是buffer不变   
pskb\_copy()  //拷贝sk\_buff和私有的头部，常用于需要修改sk\_buff的头部时  
skb\_copy() //完全拷贝  
  
skb\_queue\_head\_init()  
skb\_queue\_head()  
skb\_queue\_tail()  
skb\_dequeue\_head()  
skb\_dequeue\_tail()  
skb\_queue\_purge() //list 清空

skb\_queue\_walk() //遍历list用

在Linux2.6中，struct sk\_buf承担了socket的输入输出的传输缓存的任务。

首先，还是先看struct socket的定义

|  |
| --- |
| /\*\*  \* struct socket - general BSD socket  \* @state: socket state (%SS\_CONNECTED, etc)  \* @type: socket type (%SOCK\_STREAM, etc)  \* @flags: socket flags (%SOCK\_ASYNC\_NOSPACE, etc)  \* @ops: protocol specific socket operations  \* @file: File back pointer for gc  \* @sk: internal networking protocol agnostic socket representation  \* @wq: wait queue for several uses  \*/ struct socket {     socket\_state        state;      kmemcheck\_bitfield\_begin(type);     short            type;     kmemcheck\_bitfield\_end(type);      unsigned long        flags;      struct socket\_wq    \*wq;      struct file        \*file;     struct sock        \*sk;     const struct proto\_ops    \*ops; }; |

代码中的注释对于每一个变量说的都很清楚——看到这里，我先感叹一下，linux2.6的结构体的注释比老版本要清楚的多。到目前为止，我所看到的关键的结构体，都有清晰的注释。我们可以看出struct socket中的sock变量，是socket变量的工作核心。

那么现在跳转到struct sock的定义处。由于struct sock的定义过长，所以只展示一部分。

|  |
| --- |
| 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;     /\* skip some codes \*/     int sk\_rcvbuf;     /\* skip some codes \*/     int sk\_sndbuf;     struct sk\_buff\_head    sk\_receive\_queue;     struct sk\_buff\_head    sk\_write\_queue;  }； |

其中，sk\_rcvbuf和sk\_sendbuf分别是接收和发送缓存的字节数。

而struct sk\_buff\_head的定义如下：

|  |
| --- |
| struct sk\_buff\_head {     /\* These two members must be first. \*/     struct sk\_buff    \*next;     struct sk\_buff    \*prev;      \_\_u32        qlen;     spinlock\_t    lock; }; |

可以看出socket的接收和发送缓存是使用一个双链表将sk\_buff组织起来的。