sk_buff 详解

杨毅(yangyi@baidu.com) 2008-12-31

概述

本文对 Linux 内核网络子系统中的核心数据结构 sk_buff 进行分析，以 2.6.21-7 的内核来分析，其余版本可能存在差别。

本文档将回答以下几个与 sk_buff 有关的问题：

1. 几个长度有关的成员变量：skb->len, skb->data_len, skb->truesize 之间的关系，还包含 skb_headlen()，skb_pagelen()等，分别在何种环境下使用？
2. 几个引用计数的区别：skb->users, skb->cloned, skb_shared_info->dataref；
3. 几个指针的关系和移动：head/data/tail/end, h.raw, nh.raw, mac.raw；
4. 与 skb 共享复制有关的几个操作有什么区别？
5. skb 分配，释放的实现细节；网络子系统中会在哪些地方分配 skb，有哪些区别？
6. skb 的数据区分为哪几部分？为什么需要这么多种类，分别都应用在何种场景？互相之间的转化关系如何？

struct sk_buff 成员变量

如下变量的区别需要注意：

```
struct net_device  *dev;
int                iif;
```

dev 和 iif

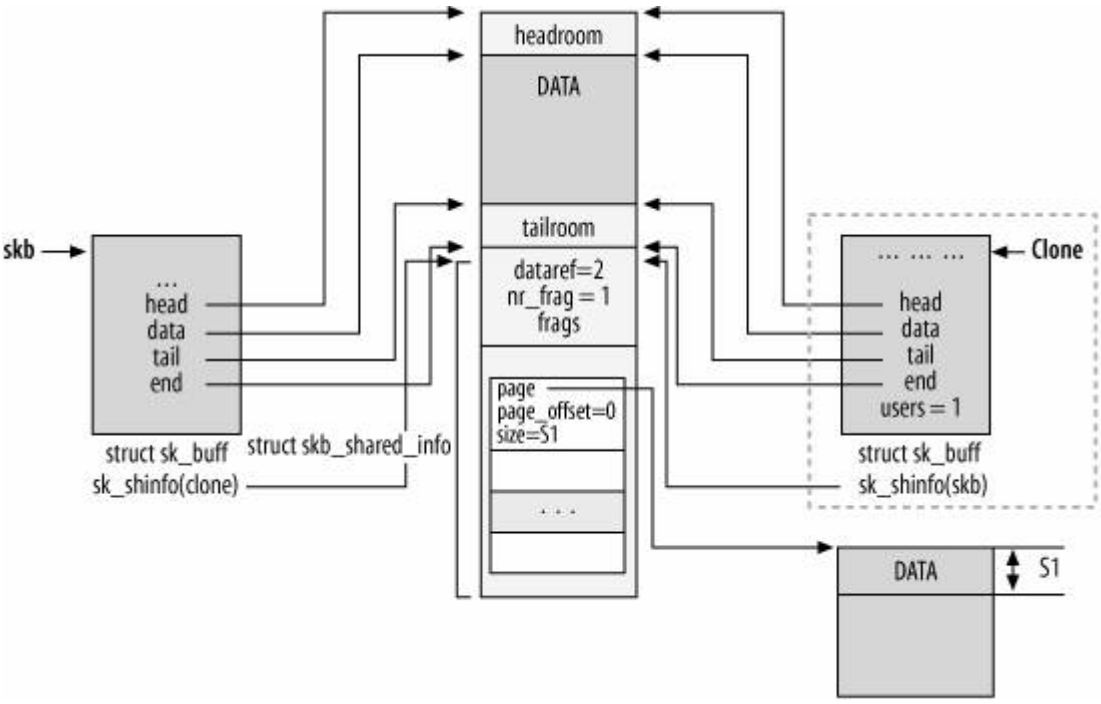
这几个变量都用于跟踪与 packet 相关的 device。由于 packet 在接收的过程中，可能会经过多个 virtual driver 处理，因此需要几个变量。

接收数据包的时候， dev 和 iif 都指向最初的 interface，此后，如果需要被 virtual driver 处理，那么 dev 会发生变化，而 iif 始终不变。

len，data_len, mac_len, truesize

```
unsigned int    len,
                data_len,
                mac_len;
unsigned int    truesize;
```

‘len’ 表示此 SKB 管理的 Data Buffer 中数据的总长度；



如上图，一个 skb 结构描述的内存区域包含几个部分：

1）sk_buff 结构体本身；

2）线性 buffer 区域，由 skb->head,data,tail,end 等几个指针来描述，并且包含 skb_shared_info 结构；

3）“paged data”：通过 skb_shared_info 结构管理的一组保存在 page 中的数据；

4）skb_shared_info(skb)->frag_list 队列包含分片的 skb 队列，队列成员是 skb 链表指针。

skb 有效数据区的长度使用 skb->len 记录，一共包括三个部分：

skb->len = skb_headlen()（?== linear_buffer_len） + skb->data_len

1. 从 skb->data 开始到 skb->tail 结束的线性 buffer 区，这一部分的长度可以使用 skb_headlen()获得；

线性区有效数据长度：

linear_buffer_len = skb->tail - skb->data;(alloc_skb 时为 0) ?== skb_headlen()

线性区总长度：**skb_data_len = skb->end - skb->head**；（不包括 skb_shared_info）

2. skb_shared_info(skb)->frags[]数组里包含的“paged data”，长度为所有 frags->size 的和；这一部分加上 1 中的数据长度可以使用 skb_pagelen()获得；“paged data”占用长度：

page_data_len = sum(skb_shared_info->frags[i]->size)；

skb_pagelen() = skb_headlen() + page_data_len

3. skb_shared_info(skb)->frag_list 链表里所有 skb 的数据区长度（len），这一部分的长度加上 2 中的长度记录在 skb->data_len 里。

skb->data_len = page_data_len + sum(skb_shared_info->frag_list->len)

truesize = sizeof(struct sk_buff) + SKB_DATA_ALIGN (skb_data_len) + skb->data_len (alloc_skb 时为 sizeof(struct sk_buff) + SKB_DATA_ALIGN(size))

len, data_len, truesize 是否包含 frag_list 里的长度？？ 是！

skb->truesize 这个成员变量衡量的是整个 skb 结构所占的内存大小（为啥不包含 struct sk_shared_info？？），在“ip 分片处理”和“socket 的读写缓存分配”中使用，前者将在以后的 ip 层处理相关文档中详细说明，后者挑选几个典型应用如下：

- skb_set_owner_w，skb_set_owner_r：datagram 的 skb 和 socket 的写，读缓存联系起来：

```
sock_hold(sk);
skb->sk = sk;
skb->destructor = sock_wfree/sock_rfree;
atomic_add(skb->truesize, &sk->sk_wmem_alloc/ sk_rmem_alloc);
```

- sock_wfree 和 sock_rfree：在 kfree_skb 时或 skb_orphan 时调用 skb 的 destructor 函数：

```
struct sock *sk = skb->sk;
atomic_sub(skb->truesize, &sk-> sk_wmem_alloc/sk_rmem_alloc);
```

- sock_queue_rcv_skb，sock_queue_err_skb 在把收到的或者错误的 skb 放入队列中时会对 socket 接收缓存，skb->truesize 及当前已经占用的读内存进行判断：

```
if (atomic_read(&sk->sk_rmem_alloc) + skb->truesize >=
        (unsigned)sk->sk_rcvbuf)
          return -ENOMEM;
```

而发送时的判断是调用比较复杂：sk_stream_alloc_pskb→ sk_stream_wmem_schedule

```
static inline struct sk_buff *sk_stream_alloc_pskb(struct sock *sk,
                                   int size, int mem,
                                   gfp_t gfp)
{
……
     skb = alloc_skb_fclone(size + hdr_len, gfp);
     if (skb) {
          skb->truesize += mem;
          if (sk_stream_wmem_schedule(sk, skb->truesize)) {
               skb_reserve(skb, hdr_len);
               return skb;
          }
          __kfree_skb(skb);
     }
……
}
static inline int sk_stream_wmem_schedule(struct sock *sk, int size)
{
     return size <= sk->sk_forward_alloc ||
```

```
        sk_stream_mem_schedule(sk, size, 0);
}
```

跟这几个长度有关的几个 inline 函数如下：

- skb_is_nonlinear()：return skb->data_len; 即是否包含“paged data”或 frag_list
- skb_headlen()：return skb->len - skb->data_len; 即只包含线性区域实际使用长度，或“有效长度”；（？？与 skb->tail - skb->data 的差别？？）
- skb_pagelen()：所有“paged data”长度之和，再加上线性区域实际使用长度；

```
static inline int skb_pagelen(const struct sk_buff *skb)
{
    int i, len = 0;
    for (i = (int)skb_shinfo(skb)->nr_frags - 1; i >= 0; i--)
        len += skb_shinfo(skb)->frags[i].size;
    return len + skb_headlen(skb);
}
```

- skb_push, skb_pull, skb_reserve, skb_put 这几个函数移动 data 或 tail 指针，都会更改 skb->len
- skb_trim：修改 skb->len 的值，并且 skb->tail = skb->data+len
- skb_headroom：skb->data - skb->head；
- skb_tailroom：如果没有 paged data 或 fragment list，为 skb->end - skb->tail，否则为 0

‘mac_len’ 指 MAC 头的长度。目前，它只在 IPSec 解封装的时候被使用。将来可能从 SKB 结构中去掉。

cloned, ip_summed, nohdr, users, dataref 等

```
__u8                    local_df:1,
                        cloned:1,
                        ip_summed:2,
                        nohdr:1,
                        nfctinfo:3;
atomic_t                users;
struct skb_shared_info {
    atomic_t    dataref;
    ……
}
```

“local_df”在 IPv4 中使用，设为 1 后代表允许对已经分片的数据包进行再次分片，在 IPSec 等情况下使用；

“ip_summed”代表网卡是否支持计算接收包 checksum，在老版本 kernel 里可取值如下：

- CHECKSUM_NONE：代表网卡不算 checksum；

- CHECKSUM_HW：代表网卡支持硬件计算 checksum（对 L4 head + payload 的校验），并且已经将计算结果复制给 skb->csum，软件需要计算“伪头(pseudo header)”的校验和，与 skb->csum 相加得到 L4 的结果；
- CHECKSUM_UNNECESSARY：网卡已经计算并校验过整个包的校验，包括伪头，软件无需再次计算，一般用于 loopback device。

本版本的内核里是如下定义，并有详细的注释：

```
#define CHECKSUM_NONE 0
#define CHECKSUM_PARTIAL 1
#define CHECKSUM_UNNECESSARY 2
#define CHECKSUM_COMPLETE 3
/* A. Checksumming of received packets by device.
 *
 *   NONE: device failed to checksum this packet.
 *         skb->csum is undefined.
 *
 *   UNNECESSARY: device parsed packet and wouldbe verified checksum.
 *         skb->csum is undefined.
 *         It is bad option, but, unfortunately, many of vendors do this.
 *         Apparently with secret goal to sell you new device, when you
 *         will add new protocol to your host. F.e. IPv6. 8)
 *
 *   COMPLETE: the most generic way. Device supplied checksum of _all_
 *       the packet as seen by netif_rx in skb->csum.
 *       NOTE: Even if device supports only some protocols, but
 *       is able to produce some skb->csum, it MUST use COMPLETE,
 *       not UNNECESSARY.
 *
 * B. Checksumming on output.
 *
 *   NONE: skb is checksummed by protocol or csum is not required.
 *
 *   PARTIAL: device is required to csum packet as seen by hard_start_xmit
 *   from skb->h.raw to the end and to record the checksum
 *   at skb->h.raw+skb->csum.
 *
 *   Device must show its capabilities in dev->features, set
 *   at device setup time.
 *   NETIF_F_HW_CSUM   - it is clever device, it is able to checksum
 *                       everything.
 *   NETIF_F_NO_CSUM - loopback or reliable single hop media.
 *   NETIF_F_IP_CSUM - device is dumb. It is able to csum only
 *                       TCP/UDP over IPv4. Sigh. Vendors like this
 *                       way by an unknown reason. Though, see comment above
 *                       about CHECKSUM_UNNECESSARY. 8)
```

```
*
*    Any questions? No questions, good.        --ANK
*/
```

“nohdr”：The 'nohdr' field is used in the support of TCP Segmentation Offload ('TSO' for short). Most devices supporting this feature need to make some minor modifications to the TCP and IP headers of an outgoing packet to get it in the right form for the hardware to process. We do not want these modifications to be seen by packet sniffers and the like. So we use this 'nohdr' field and a special bit in the data area reference count to keep track of whether the device needs to replace the data area before making the packet header modifications.

简单来说，nohdr 代表了 skb_shared_info 里的 dataref 有没有被分成两部分：

- nohdr = 0：dataref 代表整个 skb 数据区的引用计数；
- nohdr = 1：dataref 的高 16bits 代表 skb 数据区“payload 部分”的引用计数，低 16bits 代表整个 skb 数据区的引用计数。

与之相关的函数有：skb_header_cloned(), skb_header_release()等。

“cloned”代表 skb 是否被 clone，为了能迅速的引用一个 SKB 的数据，当 clone 一个已存在的 SKB 时，会产生一个新的 SKB，但是这个 SKB 会共享已有 SKB 的数据区。当一个 SKB 被 clone 后，原来的 SKB 和新的 SKB 结构中，‘cloned’都要被设置为 1。

“users”：是 sk_buff 结构本身的引用计数，在 kfree_skb 时会先先使用原子操作对 users 计数-1，如果=0，才调用__kfree_skb 真正去释放 skb。

- skb_shared()检查 skb->users 是否为 1；
- skb_shared_check()：如果 skb->users > 1，则 clone 一个新的。

有关 **dataref，clone，users** 这些引用计数的区别和应用场景请见后续的**“skb** 共享复制相关操作**”**一节。

“nfctinfo”：用于 netfilter 子系统中的 conntrack 模块记录连接状态，可取值为 enum 变量：

```
enum ip_conntrack_info
{
/* Part of an established connection (either direction). */
    IP_CT_ESTABLISHED,
/* Like NEW, but related to an existing connection, or ICMP error (in either
direction). */
    IP_CT_RELATED,
/* Started a new connection to track (only IP_CT_DIR_ORIGINAL); may be a
retransmission. */
    IP_CT_NEW,
/* >= this indicates reply direction */
    IP_CT_IS_REPLY,
/* Number of distinct IP_CT types (no NEW in reply dirn). */
    IP_CT_NUMBER = IP_CT_IS_REPLY * 2 - 1
};
```

pkt_type, fclone, ipvs_property

```
__u8                    pkt_type:3,
                    fclone:2,
                    ipvs_property:1;
```

“pkt_type”表示根据 L2 的目的地址得到包类型，可能取值在 include/linux/if_packet.h，以太网设备中使用 eth_type_trans 函数来初始化这个值：

- PACKET_HOST：
- PACKET_MULTICAST：
- PACKET_BROADCAST：
- PACKET_OTHERHOST：
- PACKET_OUTGOING：
- PACKET_LOOPBACK：
- PACKET_FASTROUTE：

“fclone”：是较新版 kernel 中添加的一个特性，以前版本的 kernel 中 skb 在分配的时候都是从后备高速缓存(lookaside cache)skbuff_head_cache 中获取 sk_buff 的结构；而现在可以在调用 alloc_skb()，通过 fclone 参数选择从 skbuff_head_cache 或者 skbuff_fclone_cache 中分配。两者的区别在于 skbuff_head_cache 在创建时指定的单位内存区域的大小是 sizeof(structsk_buff)，可以容纳任意数目的 struct sk_buff，而 skbuff_fclone_cache 在创建时指定的单位内存区域大小是 2*sizeof(struct sk_buff) +sizeof(atomic_t)，它的最小区域单位是一对 strcut sk_buff 和一个引用计数，这一对 sk_buff 是克隆的，即它们指向同一个数据缓冲区，引用计数值是 0（SKB_FCLONE_UNAVAILABLE）,1（SKB_FCLONE_ORIG）或 2（SKB_FCLONE_CLONE），表示这一对中有几个 sk_buff 已被使用：

- 分配 skb 时，skb->fclone = SKB_FCLONE_ORIG; atomic_set(fclone_ref, 1);
  child->fclone = SKB_FCLONE_UNAVAILABLE;
- skb_clone 时，如下处理

```
struct sk_buff *n;
n = skb + 1;
if (skb->fclone == SKB_FCLONE_ORIG &&
    n->fclone == SKB_FCLONE_UNAVAILABLE) {
      atomic_t *fclone_ref = (atomic_t *) (n + 1);
      n->fclone = SKB_FCLONE_CLONE;
      atomic_inc(fclone_ref);
} else {
      n = kmem_cache_alloc(skbuff_head_cache, gfp_mask);
      if (!n)
            return NULL;
      n->fclone = SKB_FCLONE_UNAVAILABLE;
}
```

即如果设置了 fclone，第一次 clone 采用快速 clone，直接使用 child_skb，并且增加 fclone_ref，child_skb->fclone = SKB_FCLONE_CLONE;其他情况都从 skbuff_head_cache 里分配新的 skb；

- 在释放 skb 结构时如下处理

```
switch (skb->fclone) {
case SKB_FCLONE_UNAVAILABLE:
    kmem_cache_free(skbuff_head_cache, skb);
    break;

case SKB_FCLONE_ORIG:
    fclone_ref = (atomic_t *) (skb + 2);
    if (atomic_dec_and_test(fclone_ref))
        kmem_cache_free(skbuff_fclone_cache, skb);
    break;

case SKB_FCLONE_CLONE:
    fclone_ref = (atomic_t *) (skb + 1);
    other = skb - 1;

    /* The clone portion is available for
     * fast-cloning again.
     */
    skb->fclone = SKB_FCLONE_UNAVAILABLE;

    if (atomic_dec_and_test(fclone_ref))
        kmem_cache_free(skbuff_fclone_cache, other);
    break;
};
```

根据 skb->fclone 标志：

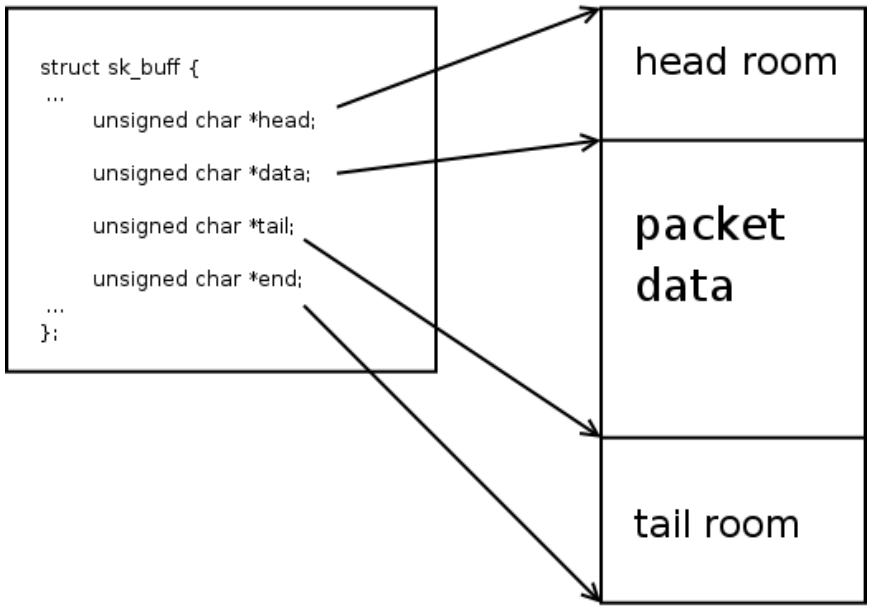
- SKB_FCLONE_UNAVAILABLE：直接归还给 skbuff_head_cache；（虽然没有经过 clone 的 child_skb->fclone 也是这个标志，但没有被 clone 也不会被释放）
- SKB_FCLONE_ORIG：代表释放的是从 skbuff_fclone_cache 里分配的 skb，如果 fclone_ref = 1（未被 clone 过）则归还 skb 给 skbuff_fclone_cache；否则（已经被 clone 过）只是将 fclone_ref--；
- SKB_FCLONE_CLONE：代表释放的代表释放的是从 skbuff_fclone_cache 里分配的 skb 经过 clone 操作后得到的新 skb，则把 fclone 标志置为 SKB_FCLONE_UNAVAILABLE，留作下次快速 clone 使用。如果 fclone_ref--=0，代表这个 skb 的兄弟 skb 已经被释放了，则归还 skb 给 skbuff_fclone_cache。

“ipvs_property”是为 ip_vs 模块添加的变量，置为 1 代表已经被 ip_vs 某个部分处理过，后续不需要再处理。可以参考 ip_vs_out()函数。

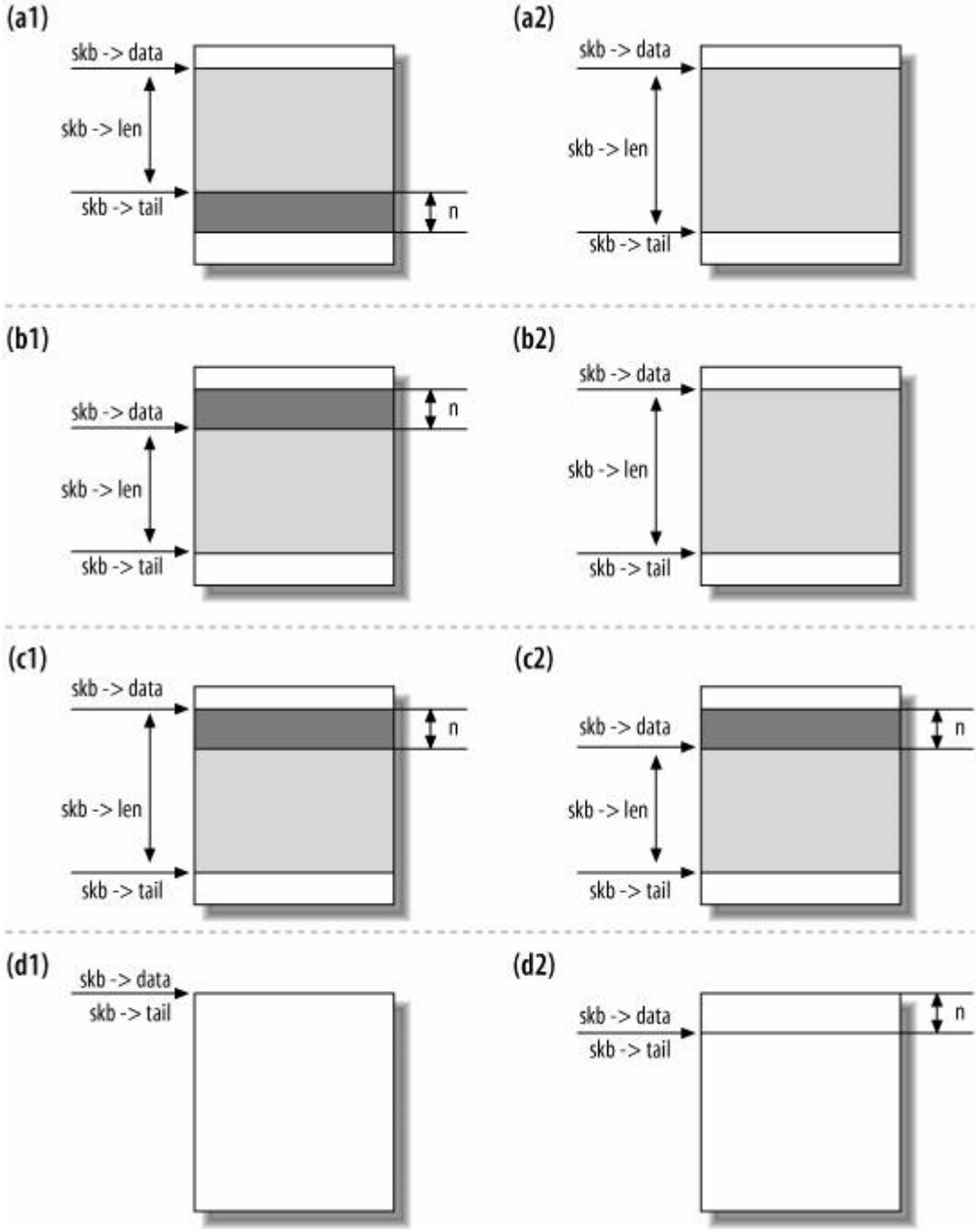
head/data/tail/end 指针， h/nh/mac 指针

```
unsigned char    *head,
                 *data,
                 *tail,
                 *end;
union {
        struct tcphdr     *th;
        struct udphdr     *uh;
        struct icmphdr    *icmph;
        struct igmphdr    *igmph;
        struct iphdr *ipiph;
        struct ipv6hdr    *ipv6h;
        unsigned char     *raw;
} h;
union {
        struct iphdr *iph;
        struct ipv6hdr    *ipv6h;
        struct arphdr     *arph;
        unsigned char     *raw;
} nh;
union {
        unsigned char     *raw;
} mac;
```

head, data, tail, end 四个指针分别指向 data buffer 的不同位置，如下图：



这四个指针的移动，最常用的四个函数为：(a)skb_put, (b)skb_push, (c)skb_pull, (d)skb_reserve ，如下图所示。



h，nh，mac 分别指向各层网络协议的头部，下面讨论各个指针在接收时都在哪些地方被初始化：

- mac 为 L2 的头部指针，在 eth_type_trans 里初始化

```
__be16 eth_type_trans(struct sk_buff *skb, struct net_device *dev)
{
……
      skb->mac.raw = skb->data;
      skb_pull(skb, ETH_HLEN);
      eth = eth_hdr(skb);
```

```
……
}
static inline struct ethhdr *eth_hdr(const struct sk_buff *skb)
{
      return (struct ethhdr *)skb->mac.raw;
}
```

- 通过了 eth_type_trans 后，skb->data 指针指向网络层的头部，在 netif_receive_skb 里初始化 nh.raw 和 h.raw：

```
      skb->h.raw = skb->nh.raw = skb->data;
      skb->mac_len = skb->nh.raw - skb->mac.raw;
```

netif_receive_skb → packet_type->func() → ip_rcv()

- 如果只是转发，不用处理到 L4，只有发给本机的包才需要，在进入 L4 处理之前，ip_local_deliver_finish 会把 h.raw 初始化为 L4 的头部：

```
      int ihl = skb->nh.iph->ihl*4;
      __skb_pull(skb, ihl);
      /* Point into the IP datagram, just past the header. */
      skb->h.raw = skb->data;
```

其他相关结构

skb_shared_info

当调用 alloc_skb() 构造 SKB 和 data buffer 时，需要的 buffer 大小是这样计算的：
data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
除了指定的 size 以外，还包括一个 struct skb_shared_info 结构的空间大小。也就是说，当调用 alloc_skb(size) 要求分配 size 大小的 buffer 的时候，同时还创建了一个 skb_shared_info 。这个结构定义如下：

```
struct skb_shared_info {
      atomic_t      dataref;
      unsigned short   nr_frags;
      unsigned short   gso_size;
      /* Warning: this field is not always filled in (UFO)! */
      unsigned short   gso_segs;
      unsigned short  gso_type;
      __be32          ip6_frag_id;
      struct sk_buff     *frag_list;
      skb_frag_t    frags[MAX_SKB_FRAGS];
};
```

“dataref”：skb 的 data 区域（线性 buffer）引用计数，即有多少个 skb 结构指向这块 data 区域（skb_clone 时++），被分为两部分：高 **16bits** 用于描述 **skb->data** 中 **payload**

部分的引用计数（**skb->nohdr = 1** 时才有效），低 **16bits** 用来描述整个 **skb->data** 的引用计数，如下：

```
/* We divide dataref into two halves.  The higher 16 bits hold references
 * to the payload part of skb->data.  The lower 16 bits hold references to
 * the entire skb->data.  It is up to the users of the skb to agree on
 * where the payload starts.
 *
 * All users must obey the rule that the skb->data reference count must be
 * greater than or equal to the payload reference count.
 *
 * Holding a reference to the payload part means that the user does not
 * care about modifications to the header part of skb->data.
 */
#define SKB_DATAREF_SHIFT 16
#define SKB_DATAREF_MASK ((1 << SKB_DATAREF_SHIFT) - 1)
```

“nr_frags”：“paged_data”计数，最多为 MAX_SKB_FRAGS = 65536/PAGE_SIZE + 2（4k 的 PAGE_SIZE，为 18）

“skb_frag_t frags[MAX_SKB_FRAGS]”：用来记录 paged_data 实际位置，包含 page 指针，在 page 中的 offset，以及占用的 size 。（可以有不同的 skb 指向同一个 page 中的不同 offset 和 size，也可以指向相同的，由 page 里头的_count 来代表引用计数）

```
struct skb_frag_struct {
	struct page *page;
	__u16 page_offset;
	__u16 size;
};
```

“gso_size”，“gso_segs”，“gso_type”由“GSO”（Generic Segmentation Offload，或叫“TSO”，即 TCP Segmentation Offload）功能使用

```
enum {
	SKB_GSO_TCPV4 = 1 << 0,
	SKB_GSO_UDP = 1 << 1,
	/* This indicates the skb is from an untrusted source. */
	SKB_GSO_DODGY = 1 << 2,
	/* This indicates the tcp segment has CWR set. */
	SKB_GSO_TCP_ECN = 1 << 3,
	SKB_GSO_TCPV6 = 1 << 4,
};
```

“frag_list”：分片处理中，记录分片使用。

我们只要把 end 从 char* 转换成 skb_shared_info* ，就能访问到这个结构

Linux 提供一个宏来做这种转换：

```
#define skb_shinfo(SKB)		((struct skb_shared_info *)((SKB)->end))
```

那么，这个隐藏的结构用意何在？它至少有两个目的：

1、 用于管理 paged data

2、 用于管理分片

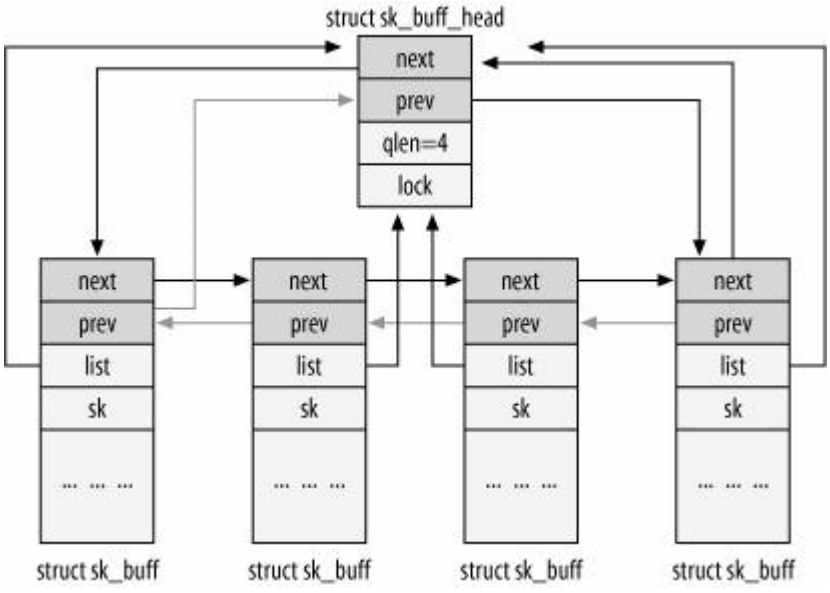
sk_buff_head 和 skb 队列

协议栈处理中经常用到 sk_buff 的队列，队列头使用 struct sk_buff_head 来描述，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;
};
```

其中包含一个指示队列长度的 qlen，和一个自旋锁 lock 用于不同的进程同时操作队列时的保护。

一个 sk_buff 队列如下图：



sk_buff 操作

下面对 net/core/skbuff.c 里调用 EXPORT_SYMBOL 导出的函数进行较为详细的分析，但仅限于每个函数本身完成的功能，具体调用的位置还得参考协议栈处理其他部分的代码。

```
EXPORT_SYMBOL(___pskb_trim);
EXPORT_SYMBOL(__kfree_skb);
EXPORT_SYMBOL(kfree_skb);
EXPORT_SYMBOL(__pskb_pull_tail);
```

```
EXPORT_SYMBOL(__alloc_skb);
EXPORT_SYMBOL(__netdev_alloc_skb);
EXPORT_SYMBOL(pskb_copy);
EXPORT_SYMBOL(pskb_expand_head);
EXPORT_SYMBOL(skb_checksum);
EXPORT_SYMBOL(skb_clone);
EXPORT_SYMBOL(skb_clone_fraglist);
EXPORT_SYMBOL(skb_copy);
EXPORT_SYMBOL(skb_copy_and_csum_bits);
EXPORT_SYMBOL(skb_copy_and_csum_dev);
EXPORT_SYMBOL(skb_copy_bits);
EXPORT_SYMBOL(skb_copy_expand);
EXPORT_SYMBOL(skb_over_panic);
EXPORT_SYMBOL(skb_pad);
EXPORT_SYMBOL(skb_realloc_headroom);
EXPORT_SYMBOL(skb_under_panic);
EXPORT_SYMBOL(skb_dequeue);
EXPORT_SYMBOL(skb_dequeue_tail);
EXPORT_SYMBOL(skb_insert);
EXPORT_SYMBOL(skb_queue_purge);
EXPORT_SYMBOL(skb_queue_head);
EXPORT_SYMBOL(skb_queue_tail);
EXPORT_SYMBOL(skb_unlink);
EXPORT_SYMBOL(skb_append);
EXPORT_SYMBOL(skb_split);
EXPORT_SYMBOL(skb_prepare_seq_read);
EXPORT_SYMBOL(skb_seq_read);
EXPORT_SYMBOL(skb_abort_seq_read);
EXPORT_SYMBOL(skb_find_text);
EXPORT_SYMBOL(skb_append_datato_frags);
```

skb 分配相关操作

__alloc_skb, alloc_skb，alloc_skb_fclone，__dev_alloc_skb, __netdev_alloc_skb,sock_alloc_send_pskb, sock_alloc_send_skb, sk_stream_alloc_skb, sk_stream_alloc_pskb, sock_wmalloc 等

基本上内核里所有的 skb 分配都是通过直接调用__alloc_skb 或相应的包装函数来完成，下面先对__alloc_skb 进行分析，然后再稍微讨论几个包装函数。

1. 输入参数：跟内存分配/相关的暂时不讨论；
   a) size：指的是线性 buffer 的长度，即 skb->end - skb->head；
   b) gfp_mask：allocation_mask，跟内存分配的优先级等相关；
   c) fclone：从 skbuff_fclone_cache 还是 skbuff_head_cache 分配 skb 结构，

用于快速 clone；

d) node：用于分配内存的 numa node

2. 具体操作：
   a) 分配 skb 结构本身：根据 fclone 选择 cache，然后把 gfp_mask&~__GFP_DMA，得到 skb，分配失败则退出；
   b) 分配 skb->data 线性 buffer 区：
      i. size = SKB_DATA_ALIGN(size)；根据 L1_CACHE_BYTES（X86 上为 128 bytes）调整大小，补齐为 128 的整数倍；
      ii. kmalloc 的内存大小为 size + sizeof(struct skb_shared_info)
      iii. 初始化 skb 成员变量
         1. truesize 之前的都置为 0；
         2. truesize = SKB_DATA_ALIGN(size) + sizeof(struct sk_buff)，并不包含 **skb_shared_info**；
         3. users 置为 1，head/data/tail 指向线性 buffer 的开始；end 指向结束；
         4. 根据 fclone 设置 skb->fclone，以及对应的兄弟 skb 及 fclone_ref；
      iv. skb_shared_info 初始化
         1. dataref 置为 1；
         2. 其他置为 0 或 NULL；
3. 返回值：skb 或 data 内存分配失败，返回 NULL；否则返回创建的 sk_buff 指针。

常见的__alloc_skb 包装函数包括：

- alloc_skb 和 alloc_fclone_skb：原型如下，一个 fclone，另一个不理；numa node 都是-1；

```
static inline struct sk_buff *alloc_skb(unsigned int size,gfp_t priority)
{
    return __alloc_skb(size, priority, 0, -1);
}
static inline struct sk_buff *alloc_skb_fclone(unsigned int size,gfp_t priority)
{
    return __alloc_skb(size, priority, 1, -1);
}
```

- dev_alloc_skb 和 __dev_alloc_skb：一般是网卡驱动分配用于接收缓存的 skb，length 会加上 NET_SKB_PAD，调用完 alloc_skb 后，会通过 skb_reserve 在线性 buffer 的头部保留一块长度为 NET_SKB_PAD 的区域，有关 NET_SKB_PAD 和 NET_IP_ALIGN 的说明，可以参见内核代码中的注释；

```
/*
 * CPUs often take a performance hit when accessing unaligned memory
 * locations. The actual performance hit varies, it can be small if the
 * hardware handles it or large if we have to take an exception and fix it
 * in software.
 *
 * Since an ethernet header is 14 bytes network drivers often end up with
```

```
 * the IP header at an unaligned offset. The IP header can be aligned by
 * shifting the start of the packet by 2 bytes. Drivers should do this
 * with:
 *
 * skb_reserve(NET_IP_ALIGN);
 *
 * The downside to this alignment of the IP header is that the DMA is now
 * unaligned. On some architectures the cost of an unaligned DMA is high
 * and this cost outweighs the gains made by aligning the IP header.
 *
  * Since this trade off varies between architectures, we allow NET_IP_ALIGN
 * to be overridden.
 */
#ifndef NET_IP_ALIGN
#define NET_IP_ALIGN  2
#endif

/*
 * The networking layer reserves some headroom in skb data (via
 * dev_alloc_skb). This is used to avoid having to reallocate skb data when
 * the header has to grow. In the default case, if the header has to grow
 * 16 bytes or less we avoid the reallocation.
 *
  * Unfortunately this headroom changes the DMA alignment of the resulting
 * network packet. As for NET_IP_ALIGN, this unaligned DMA is expensive
 * on some architectures. An architecture can override this value,
 * perhaps setting it to a cacheline in size (since that will maintain
 * cacheline alignment of the DMA). It must be a power of 2.
 *
 * Various parts of the networking layer expect at least 16 bytes of
 * headroom, you should not reduce this.
 */
#ifndef NET_SKB_PAD
#define NET_SKB_PAD  16
#endif
```

- netdev_alloc_skb 和__netdev_alloc_skb：也是用于网卡驱动分配 skb，与前面提到的 dev_alloc_skb 的区别在于：调用 alloc_skb 时多了一个 numa node 的判断，如下：

```
static inline struct sk_buff *netdev_alloc_skb(struct net_device *dev,
           unsigned int length)
{
```

```
    return __netdev_alloc_skb(dev, length, GFP_ATOMIC);
}

struct sk_buff *__netdev_alloc_skb(struct net_device *dev,
        unsigned int length, gfp_t gfp_mask)
{
    int node = dev->dev.parent ? dev_to_node(dev->dev.parent) : -1;
    struct sk_buff *skb;

    skb = __alloc_skb(length + NET_SKB_PAD, gfp_mask, 0, node);
    if (likely(skb)) {
        skb_reserve(skb, NET_SKB_PAD);
        skb->dev = dev;
    }
    return skb;
}
```

- sock_alloc_send_skb 和 sock_alloc_send_pskb：sock_alloc_send_skb 是 sock_alloc_send_pskb 的包装，一般在 ip_output，af_unix, af_packet, raw.c，udp 等处调用，我们主要关心 skb 的分配和初始化情况，socket 相关的代码不作讨论：
  1. 输入参数：
     a) size：线性 buffer 的长度；（header_len）
     b) data_len：paged data 的长度；kernel 里目前都是直接使用 sock_alloc_send_skb 来调用 sock_alloc_send_pskb，所以 data_len = 0
  2. skb 分配过程：
     a) 先调用 alloc_skb，传入的 size 参数为 header_len；
     b) 如果没有 data_len，直接返回；否则
        i. 根据 data_len，获得 page 数目，赋给 skb_shinfo(skb)->nr_frags；
        ii. skb->truesize += data_len；
        iii. 通过调用 alloc_page 分配 paged data，除了最后一个 frags 之外，其他的 frags->offset = 0; frags->size = PAGE_SIZE；最后一个的 frags->size = data_len % PAGE_SIZE。

- sk_stream_alloc_skb 是 sk_stream_alloc_pskb 的包装，从名字上看，就是用于 stream 发送的时候分配 skb，一般是在 tcp 层调用，下面三处调用 sk_stream_alloc_pskb
  - tso_fragment
  - tcp_sendmsg
  - do_tcp_sendpages

  下面两个调用 sk_stream_alloc_skb

- tcp_fragment
- tcp_mtu_probe

下面分析代码：

1. 输入参数：
   a) size：待发送的字节流长度；
   b) mem：不知道干啥用的，看起来只是修改了 skb->truesize，所有调用的地方传递的值也都是 0；
2. skb 分配过程：
   a) size + 根据 sk->sk_proto 算出来的最大头部长度（对于 TCP，#define MAX_TCP_HEADER (128 + MAX_HEADER)）作为分配的 skb 里线性 Buffer 的长度；
   b) 调用 alloc_skb_fclone 分配 skb（因为很快就会被 clone？）
   c) 调用 skb_reserve，把头部空出来以备后用。

- sock_wmalloc：如果在发送一个很长的数据包，L4 在处理的时候会为 IP 层的分片提前做一些工作，比如将数据放置在一系列的 skb 中，一般调用 sock_alloc_send_skb 来创建这一系列 skb 中的第一个（可以参考 ip_append_data），调用 sock_wmalloc 来分配剩下的分片 skb。函数体本身很简单，如下：

```
struct sk_buff *sock_wmalloc(struct sock *sk, unsigned long size, int force,
                             gfp_t priority)
{
    if (force || atomic_read(&sk->sk_wmem_alloc) < sk->sk_sndbuf) {
        struct sk_buff * skb = alloc_skb(size, priority);
        if (skb) {
            skb_set_owner_w(skb, sk);
            return skb;
        }
    }
    return NULL;
}
```

skb 释放

kfree_skb 和 *__kfree_skb*

kfree_skb()是__kfree_skb()的一个包装，首先检查 skb 的引用计数 skb->users，为 1 才继续调用__kfree_skb()，否则只是 skb->users - 1（原子操作）。

在 kernel 的协议栈代码中，很多地方都是先显式的调用 atomic_inc(&skb->users);然后处理一堆内容后，调用 kfree_skb，而不是调用 atomic_dec(&skb->users)。这样做的好处是，可以在合适的地方尽快释放 skb，并且可以让处理 **skb** 结构的代码逻辑更独立。（调用处理 skb 的函数之前可能增加过 skb->users，也可能没有）

__kfree_skb()是真正释放 skb 的函数：先将 skb 结构里指向其他网络子系统的指针 release 或 put，并且调用析构函数 skb->destructor()，最后通过 kfree_skbmem()来释放内存：

- 首先调用 skb_release_data 来处理线性 buffer 区域和 paged_data（包括 frag_list）；判断 skb 的 Data Buffer 区域可以被释放的条件是以下二者之一：
  - !skb->cloned：skb 没有被 clone（注意被 clone 的和 clone 生成的 skb->cloned 都被置 1，两个 skb 里的 skb_shinfo(skb)->dataref 都 atomic_inc，见 skb_clone 的讨论），**skb** 没被 **clone**，说明数据区肯定只有唯一一个 **skb** 指向；
  - !atomic_sub_return(skb->nohdr ? (1 << SKB_DATAREF_SHIFT) + 1 : 1,
    &skb_shinfo(skb)->dataref)
    对 skb_shinfo(skb)->dataref 进行判断：
    - 如果 nohdr 被设置，代表 dataref 被分成两部分，前面 SKB_DATAREF_SHIFT（16）bit 代表 skb->data 里的 payload 部分的引用计数；后面 16bits 代表整个 skb->data 的引用计数。所以需要把 dataref-(1 << SKB_DATAREF_SHIFT) + 1），来判断是否可以释放 skb->data；
    - nohdr 没被设置，不需要把 skb->data 的 payload 单独考虑引用计数，只需要 dataref-1 来决定是否释放 skb->data
  - 释放 Data Buffer 分为三个部分：
    - put_page 释放“paged data”；
    - skb_drop_fraglist，进而调用 kfree_skb，来释放 frag_list 里的所有 skb；
    - kfree(skb->head)，释放“线性 Buffer”。
- 然后根据 skb->fclone 标志将 skb 结构本身归还 slab cache。见 fclone 讨论部分

skb 共享复制相关操作

skb_clone, skb_cloned, skb_header_cloned, skb_shared, skb_share_check, skb_unshare, skb_clone_fraglist, skb_copy, pskb_copy, skb_header_release

内核的网络子系统核心数据结构是 sk_buff，同一个 skb 可能会被很多个子系统共享，同时使用或修改，出于性能的考虑，内核中提供了一系列对 skb 的共享和拷贝函数，网络子系统根据各自的需求来调用：

skb_share: 只读的共享需求

- 需要共享的地方调用：直接调用 atomic_inc(skb->users)，或者 skb_get()；
- 判断是否被共享：skb_shared()

```
static inline int skb_shared(const struct sk_buff *skb)
```

```
{
    return atomic_read(&skb->users) != 1;
}
```

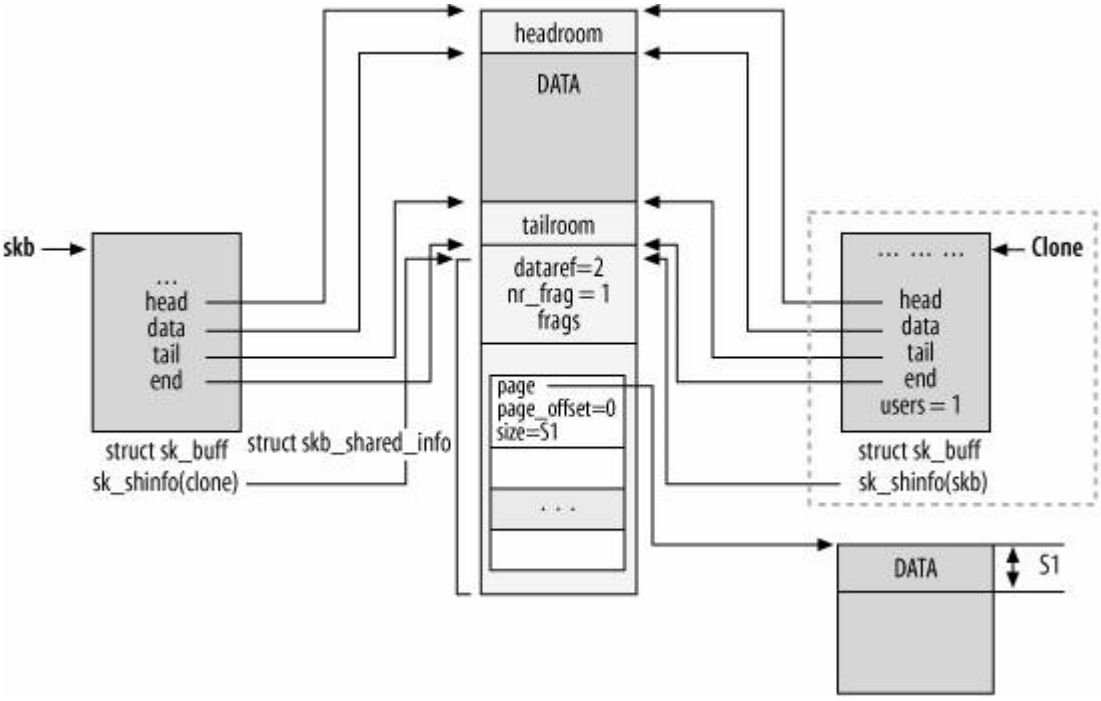
- 取消共享的地方调用：kfree_skb()（见 kfree_skb 一节的讨论）
- 对分片队列的共享：skb_clone_fraglist()，（pskb_expand_head()里用了），其实就是把 skb 队列里每一个 skb 的引用计数+1；然后在取消共享时，对队列里每一个 skb 调用 kfree_skb()见减小引用计数。

```
static inline void skb_drop_fraglist(struct sk_buff *skb)
{
    skb_drop_list(&skb_shinfo(skb)->frag_list);
}

static void skb_drop_list(struct sk_buff **listp)
{
    struct sk_buff *list = *listp;
    *listp = NULL;
    do {
        struct sk_buff *this = list;
        list = list->next;
        kfree_skb(this);
    } while (list);
}

static void skb_clone_fraglist(struct sk_buff *skb)
{
    struct sk_buff *list;
    for (list = skb_shinfo(skb)->frag_list; list; list = list->next)
        skb_get(list);
}
```

skb_clone: 只修改 sk_buff 结构本身

- 需要共享的地方调用：skb_clone()
  i. 首先根据 skb->fclone 决定采用快速 clone 机制还是普通机制；
  ii. 新分配的 skb 暂时不用和 skb 链表，socket 关联，都设为 NULL；
  iii. 其他成员的值都 copy，需要增加引用计数的也都使用 XXX_get()处理；
  iv. 新 skb 的 users 为 1（与 alloc_skb()一样），n->nohdr = 0, destructor = NULL；
  v. 新老 skb 的 skb->cloned = 1；
  vi. 新老 skb 指向的数据区是一样的（包括线性 buffer 和 paged data，分片

等），则需要把 dataref+1：atomic_inc(&(skb_shinfo(skb)->dataref))（整个数据区的引用计数）



- 判断是否被共享：
  - skb_cloned()：返回 1 的条件为 skb->cloned = 1 并且 dataref 的低 16bit 部分（整个数据区的引用计数）不为 1；

```
static inline int skb_cloned(const struct sk_buff *skb)
{
    return skb->cloned &&
           (atomic_read(&skb_shinfo(skb)->dataref) & SKB_DATAREF_MASK) !=
1;
}
```

  - skb_header_cloned()：判断数据区里的 header 部分是否被 clone，返回 1 的条件为 skb->cloned = 1 并且 dataref 的低 16bits（整个数据区的引用计数）- 高 16bits（**payload** 部分引用计数）的差值不为 1；
- 取消共享：
  - 被 clone 过的 skb，以及 clone 生成的 skb 里 skb->cloned = 1，直到被释放时都不会被清除。只是在 __kfree_skb()→kfree_skbmem() → skb_release_data() 里减小 dataref（如果 nohdr = 1，同时将高 16bits[payload 部分]和低 16bits[整个数据区部分]减 1，否则只减低 16bits[整个数据区部分]）

```
static void skb_release_data(struct sk_buff *skb)
{
    if (!skb->cloned ||
       !atomic_sub_return(skb->nohdr ? (1 << SKB_DATAREF_SHIFT) + 1 : 1,
                          &skb_shinfo(skb)->dataref)) { // 如果 skb 被 clone 了，就减一下
引用计数，减完结果为 0 了就可以释放数据区内存了
……
```

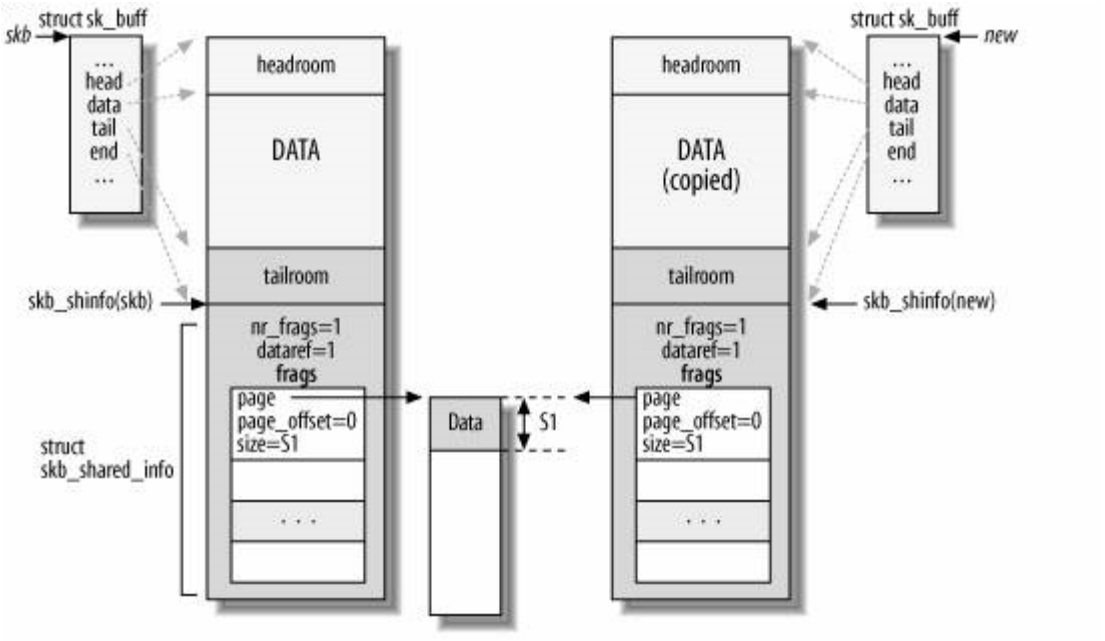
```
}
```

- skb_header_release()：把 nohdr = 1，并且把 dataref 高 16bits（payload 部分引用计数）+1（**header** 部分引用计数 = 整个数据区引用计数 **- payload** 部分引用计数），调用了此函数之后不准再读取数据区的 **header** 部分，相当于放弃对此 skb 数据区中 header 部分的引用。

```
static inline void skb_header_release(struct sk_buff *skb)
{
    BUG_ON(skb->nohdr);
    skb->nohdr = 1;
    atomic_add(1 << SKB_DATAREF_SHIFT, &skb_shinfo(skb)->dataref);
}
```

pskb_copy: 同时需要修改 sk_buff 和线性 Buffer 部分的数据

- 需要共享的地方，调用 pskb_copy()，如下图



1）旧 skb 基本没有任何变化，包括各种 len，users 和 dataref 之类的引用计数；只是各种成员变量的引用计数需要增加，包括 skb_shared_info 里的 frags 对应的 page 被 get_page()一次，frag_list 里对应的 skb->users 被+1，skb->dst 被 dst_clone()等；

2）新 skb 里的长度信息（len, data_len, truesize 等）与旧 skb 是一样的，引用计数与 alloc_skb 得到的 skb 是一致的：skb->users = 1,

skb_shared_info->dataref = 1, skb->cloned = 0。

- 不用判断是否被共享；
- 取消共享：skb_release_data()：put_page()，并 skb_drop_fraglist()

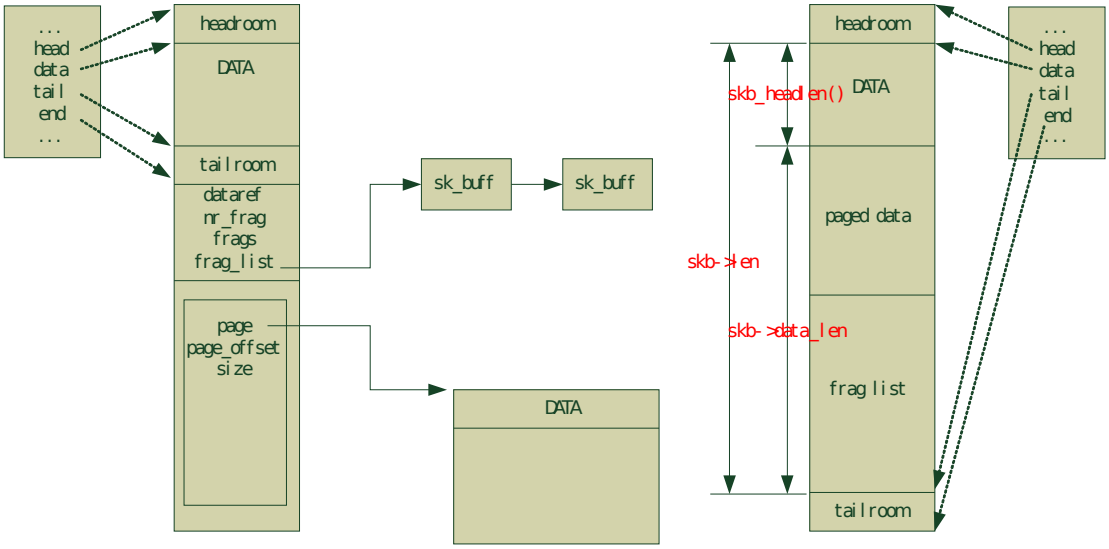
skb_copy：skb 结构，线性 buffer 区和 paged data 都需要修改

- 需要共享的地方调用 skb_copy

  1）旧的 skb 没有变化，也只是成员变量的引用计数需要增加，与 pskb_copy() 不同的是 page 和 frag_list 的引用计数不用改变；

  2）会把 paged data 和 frag_list 里的数据都给 copy 到线性 Buffer 区域，新 skb（new）的 nr_frags = 0, frag_list = NULL，长度信息变化如下：

  i. data_len = 0;
  ii. new->len = skb->len;
  iii. new->truesize = sizeof(struct sk_buff) + **SKB_DATA_ALIGN(skb->end - skb->head + skb->data_len)**；对比旧的：skb-> truesize = sizeof(struct sk_buff) + **SKB_DATA_ALIGN**（**skb->end - skb->head**）+ **skb->data_len**）
  iv. skb_headlen(new) = skb->len, skb_pagelen(new) = skb->len



3）新 skb 的引用计数与 alloc_skb 得到的一致。

- 新的 skb 和旧的 skb 除了数据部分是 copy 的，没有任何关系，所以不用判断共享状态，也不用取消共享。

skb 与 kernel buffer 拷贝数据

skb_copy_bits,skb_store_bits, skb_copy_and_csum_bits, skb_checksum, skb_copy_and_csum_dev

- ✓ skb_copy_bits()，从 skb 的数据区指定偏移（int offset）处，copy 一定长度（int len）的数据到特定指针（void *to）指向的内存空间，成功返回 0，否则返回-EFAULT。skb 数据区的长度使用 skb->len 记录，一共包括三个部分：
  1. 从 skb->data 开始到 skb->tail 结束的线性 buffer 区，这一部分的长度可以使用 skb_headlen()获得；
  2. skb_shared_info(skb)->frags[]队列里包含的“paged data”，长度为所有 frags->size 的和；这一部分加上 1 的数据长度可以使用 skb_pagelen()获得；
  3. skb_shared_info(skb)->frag_list 队列里所有 skb 的数据区长度（len），这一部分的长度加上 2 中的长度记录在 skb->data_len 里。
  skb_copy_bits()的 copy 顺序就是以 skb->data - offset 为起点，按照上述 1~3，如果凑不够需要的 len，就返回错误；一旦凑够立刻返回 0。需要注意的是 offset 可能为负数，即从 skb->data 往前，最多到 skb->head。skb_copy()里就曾这么使用。
- ✓ skb_store_bits()，skb_copy_bits 的逆操作，从指针（void *from）指向的内存空间，copy 一定长度（int len）到 skb 的数据区指定偏移（int offset）处，成功返回 0，否则返回-EFAULT，copy 顺序也如上所述。
- ✓ skb_copy_and_csum_bits()，与 skb_copy_bits()基本一致，只是需要把 copy 出来的数据按照 32bit 对齐计算 checksum 并返回（跟 TCP，IP 头部计算校验和一样的算法）；还有一个类似的函数是 skb_checksum()，只是把对应 offset 位置，长度为 len 的数据计算 checksum，并不 copy；
- ✓ skb_copy_and_csum_dev()，与 skb_copy_and_csum_bits()一样的功能，不同的是利用硬件计算的部分 checksum 结果（skb->ip_summed == CHECKSUM_PARTIAL）
- ✓ 功能类似的函数还有 skb_copy_and_csum_datagram(),skb_copy_datagram_iovec()以及 skb_copy_datagram_iovec()，详细功能分析暂缺。

skb 数据区相关操作

__pskb_trim, pskb_expand_head, __pskb_pull_tail, __pskb_pull, pskb_pull, pskb_may_pull, __skb_linearize, skb_pad, skb_padto, skb_realloc_headroom, skb_append_datato_frags, skb_copy_expand, skb_split，skb_split_inside_header，skb_split_no_header

___pskb_trim，pskb_trim，__pskb_trim，skb_trim，__skb_trim，pskb_trim_unique，pskb_trim_rcsum，__pskb_trim_head

trim 的目的是将 skb 指向的 buffer 区域（包括线性 buffer 和 paged data 一起）的长度缩小为 len，从 skb->data 开始计算（其实就是 skb->len 标识的区域大小），注意___pskb_trim 有三个下划线……跟它相关的函数还包括了：

1. pskb_trim()：检查 len 的合法性，只有比 skb->len 小，才能调用__pskb_trim；否则什么都不做。
2. __pskb_trim()：两个下划线，如果 skb->data 区域包含“paged data”，调用___pskb_trim()（三个下划线），否则调用__skb_trim()。
3. skb_trim()和__skb_trim()：前者包含 len 的合法性检查，后者就是将 skb->len 改为 len，并且移动 skb->tail 到 skb->data+len。
4. 最后主角___pskb_trim()出场，分以下几种情况来处理：
   - skb_headlen >= len：即线性 buffer 的有效长度已经大于 len，这样只用保留线性 buffer 从 skb->data 开始的长度为 len 的区域，分别调用 put_page 和 skb_frag_list 释放 paged_data 和 frag_list； 然后将 skb->len = len；skb->tail = skb->data + len; skb->data_len = 0
   - skb_headlen < len，但是加上 skb_shared_info->frags 可以凑够 len，则将当前的 **frags.size = len - offset**；其他多余的 paged_data 和 frag_list 都释放；然后 skb->data_len = skb->len - len; skb->len = len；
   - skb_headlen + skb_shared_info->frags 的长度都凑不够 len，只能从 frag_list 下手了。遍历 frag_list 链表，如果被 share，则 clone 一个新的，然后计算总的 offset，如果已经凑够了 len，则调用 pskb_trim()保留当前的 **frag_skb** 里的 **buffer** 长度为 **len - offset**；将多余的 frag_skb 全部释放。
   - 如果 skb->data+paged_data+frag_list 一起都凑不够 len，什么也不做。（这种情况在 pskb_trim 里已经进行过合法性检查，一般不会进来）
5. pskb_trim_unique()：就是 pskb_trim，由调用者保证 skb 没被 clone，则不会出现需要重新分配数据区导致 out-of-memory 而造成 pskb_trim 返回错误；
6. pskb_trim_rcsum()：与 pskb_trim 一样，但是保证 pskb_trim 之后接收包的 checksum 依然有效：

```
if (skb->ip_summed == CHECKSUM_COMPLETE) //接收包的 checksum 已经由网卡计算并通过
        skb->ip_summed = CHECKSUM_NONE;  //pskb_trim 会更改数据区内容，所以网卡校验过的 checksum 无效
```

7. __pskb_trim_head()：与本节讨论的 trim 操作没什么太大关系，从实现上看其完成的功能是：

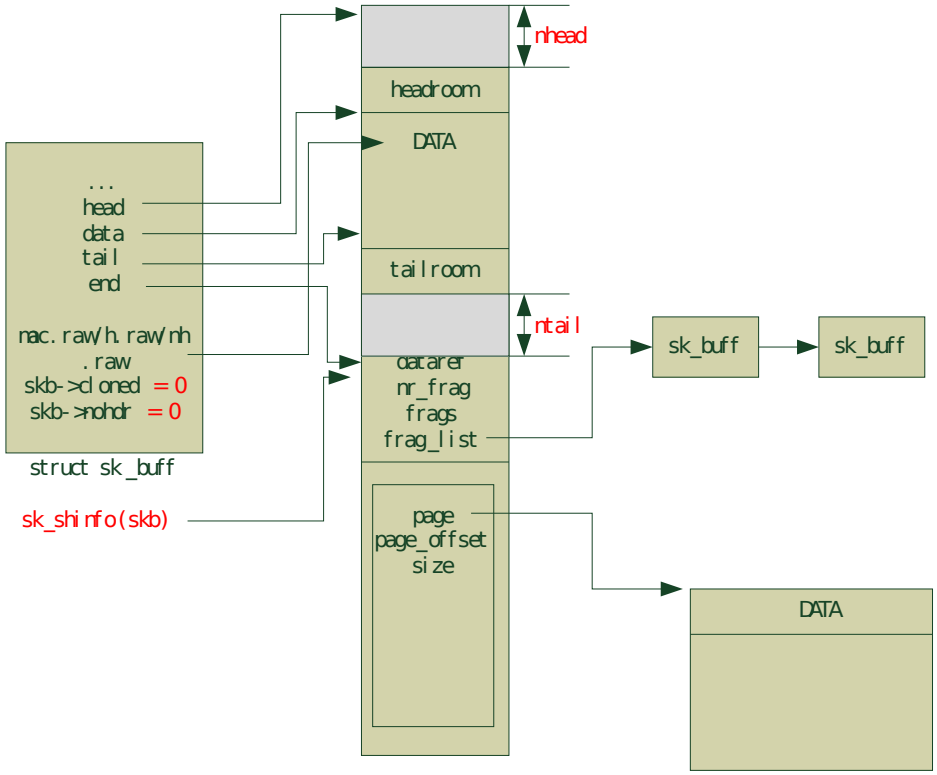
- 按照顺序从 paged data 里扣除长度为 len 的数据，删除的数据使用 put_page 释放引用计数；修改 skb->data_len -= len。 需要注意的是，如果出现 paged data 长度加起来都不足 len 的情况，则把所有的 page 都给 put 了，此时 data_len 有可能变为负数。看起来是由调用者（tcp_trim_head() 和 tcp_mtu_probe()）保证这种异常不会出现；
- 把线性 buffer 区长度变为 0，skb->tail = skb->data；修改 skb->len = skb->data_len。

pskb_expand_head

这里的 head 就是指 skb 的“线性 buffer”区域，此函数的功能就是将 skb->data 指向的区域扩大，head 前加 nhead，end 后加 ntail，并且调用 skb_release_data()将原有的内存区域释放。如下图，对 skb 的影响包括：

- skb 结构本身：head/data/tail/end，以及 mac/h/nh.raw 等指针的指向被修改；cloned 和 nohdr 被置 0（因为新的 data 区域只有本 skb 指向）；
- 线性区域：新线性区域里的内容与旧的一样，包括 skb_shared_info 结构里的值。旧的 skb->head 如果有其他 skb 指向，就什么都不干；否则就 put_page()，drop_frag_list，再 kfree(skb->head)；
- skb_shared_info：
  - paged_data：get_page()，新的 skb_shared_info->frags 指向；
  - frag_list：skb_clone_fraglist，新的 skb_shared_info->frag_list 指向；
  - data_ref：被置为 1；

有些代码里调用此函数，把 nhead 和 ntail 都设为 0，目的只是为了在 skb 已经 clone 的情况下，复制一份 skb->data 用于修改，例如上面的___pskb_trim()函数。



__pskb_pull_tail, __pskb_pull, pskb_pull, pskb_may_pull, __skb_linearize，skb_linearize，skb_linearize_cow

这几个都是__pskb_pull 相关，注意与 skb_pull 或__skb_pull 区别，后者一般只在线性 buffer 区移动相关指针，而本节涉及到的函数都“可能”重新分配 skb 的所有数据区，包括线性 buffer 区，paged data 以及 frag list 三种，并移动所有相关的指针。

- __pskb_pull_tail：把 skb 的 tail 指针后移 delta，并从 paged data 和 frag list 里 copy 出长度为 delta 的数据，放在扩大的区域；如果成功返回新的 skb->tail，否则返回 NULL；
  1. 首先检查 skb->tail 到 skb->end 之间是否有足够的空间；
  2. 如果空间不足，或者 skb 被 clone 了，需要调用 pskb_expand_head 重新扩大

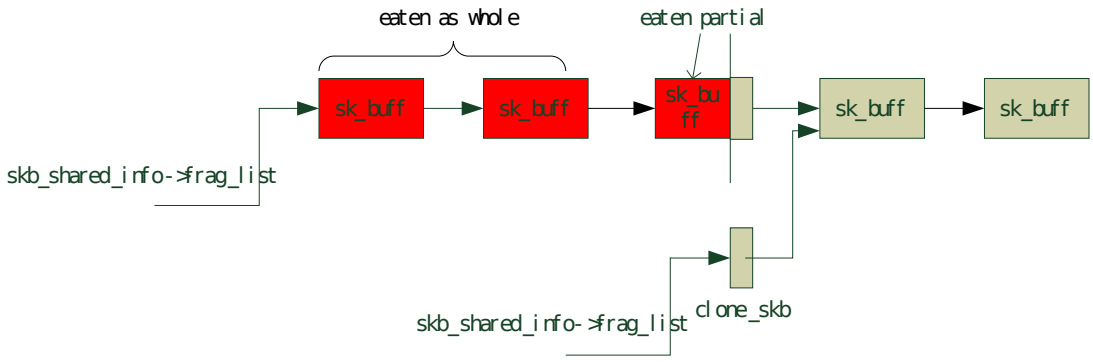
（扩大 delta+128 bytes，以备后用）或复制一份数据区；扩大不了就返回 NULL；

3. 接着使用 skb_copy_bits 从原始 skb 有效数据区里的 skb->data + skb_headlen()开始，copy 出长度为 delta 的数据到 skb->tail 指向的内存，即从 paged data 和 frag list 里复制数据到线性 buffer 区。如果 copy 出错，报 bug；
4. 往下需要修改 skb 数据区的一些指针，以及 skb_shared_info 里的 frags 和 frag_list：
   a) 如果没有 frag_list，或者是 paged data 的长度已经足够 delta 了，则只用修改 frags 和 tail 指针：使用 put_page 删掉需要取出的 paged data，然后把剩余的往前移，如下：

```
eat = delta;
k = 0;
for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
    if (skb_shinfo(skb)->frags[i].size <= eat) {
        put_page(skb_shinfo(skb)->frags[i].page);
        eat -= skb_shinfo(skb)->frags[i].size;
    } else {
        skb_shinfo(skb)->frags[k] = skb_shinfo(skb)->frags[i];
        if (eat) {
            skb_shinfo(skb)->frags[k].page_offset += eat;
            skb_shinfo(skb)->frags[k].size -= eat;
            eat = 0;
        }
        k++;
    }
}
skb_shinfo(skb)->nr_frags = k;

skb->tail     += delta;
skb->data_len -= delta;
```

   b) 如果 paged data 加起来还不够 delta，需要修改 frag_list，比较复杂：
      i. 先“吃掉”所有的 paged data，还需要吃掉的长度记录到 eat 里；
      ii. 遍历 frag_list 链表：如果 list->len <= eat，整个吃掉，直到吃到某个 frag_skb 部分吃掉就能满足。
      iii. 先处理部分吃掉的 frag_skb，如果被 share 了，还需要先 clone。然后调用 pskb_pull（这个函数也可能调用__pskb_pull_tail，递归的含义）来实现部分吃掉；
      iv. 最后处理 frag_list 链表，把完全吃掉的 frag_skb 从链表中删除，并且调用 kfree_skb 删除，这样之后 frag_list 的指针会指向部分吃掉的 frag_skb，如果之前 clone 过，就使用 clone 生成的 skb 取代当前部分吃掉的 frag_skb 放入链表。
      v. 最后是删掉所有的 paged data，并修改 skb->tail 指针和 skb-

>data_len



- pskb_pull 和__pskb_pull：skb_pull（__skb_pull）和 pskb_pull（__pskb_pull）完成的功能类似，区别在于前者认为线性 buffer 区足够长（skb_headlen() >= len），只用移动 skb->data 指针，并把 skb->len 减小；而后者只要求（skb->len >= len），如果线性 buffer 区不够长，会调用__pskb_pull_tail 来从 paged data 和 frag list 里往线性 buffer 里补充。
- pskb_may_pull：用于判断 skb 的线性 buffer 区可否 pull：
  1. 如果 len <= skb_headlen()，return 1；
  2. 如果 len > skb->len，return 0；
  3. 如果 len 介于二者之间，就调用_pskb_pull_tail 从 paged data 和 frag list 里往线性 buffer 里补充，执行成功返回 1，否则返回 0。
- __skb_linearize，skb_linearize，skb_linearize_cow：skb 不是线性的意思是 skb->data_len > 0，即 skb 的数据区除了线性区外还包含 paged data 或 frag list。则 skb 线性化的意思就是把 paged data 和 frag list 里的数据补充到线性 buffer 里，利用__pskb_pull_tail(skb, skb->data_len)就可完成，成功返回 0，否则返回-ENOMEM。skb_linearize_cow 除了把 skb 线性化，而且保证 skb 的数据区可写（被 clone 了就调用 pskb_expand_head()复制一份数据区）

skb_pad 和 skb_padto

这两个函数是用于往 skb 的 tailroom 里补 0，网卡驱动在 DMA 或把数据在内存和 wire 之间搬运时使用：

- skb_pad：在 skb->tail 之后补充长度为 pad 的 0。如果成功返回 0，否则会释放 skb 并返回错误类型；
  1. 如果 skb 没被 clone，并且是线性的，而且 skb_tailroom 的长度还大于 pad，直接 memset(skb->data+skb->len, 0, pad);
  2. 如果 skb 被 clone 了，会复制一份数据区；
  3. 如果 tailroom 的长度不足以放下 skb->data_len + pad，则 pskb_expand_head 扩大一下线性数据区；
  4. 把 skb 线性化，然后 memset(skb->data+skb->len, 0, pad)；如果出错 kfree_skb 并返回错误类型。
- skb_padto：在 skb->tail 后补充 0，直到 skb->len + pad_length 满足一定需求

（= len）：

1. 如果 skb->len >= len，返回 0；
2. 否则调用 skb_pad，补充长度为 len - skb->len 的 0。

skb_realloc_headroom

输入参数@skb，@headroom；返回一个可写的（private）skb（包括线性数据区），其中 headroom 的长度要大于等于@headroom，成功则返回新的 skb 指针，否则返回 NULL。

skb_copy_expand

注释非常清楚，函数代码也很简单，直接 copy 过来：

```
/**
 *	skb_copy_expand	-	copy and expand sk_buff
 *	@skb: buffer to copy
 *	@newheadroom: new free bytes at head
 *	@newtailroom: new free bytes at tail
 *	@gfp_mask: allocation priority
 *
 *	Make a copy of both an &sk_buff and its data and while doing so
 *	allocate additional space.
 *
 *	This is used when the caller wishes to modify the data and needs a
 *	private copy of the data to alter as well as more space for new fields.
 *	Returns %NULL on failure or the pointer to the buffer
 *	on success. The returned buffer has a reference count of 1.
 *
 *	You must pass %GFP_ATOMIC as the allocation priority if this function
 *	is called from an interrupt.
 *
 *	BUG ALERT: ip_summed is not copied. Why does this work? Is it used
 *	only by netfilter in the cases when checksum is recalculated? --ANK
 */
```

skb_append_datato_frags

```
/**
 * skb_append_datato_frags: - append the user data to a skb
```

```
 * @sk: sock  structure
 * @skb: skb structure to be appened with user data.
 * @getfrag: call back function to be used for getting the user data
 * @from: pointer to user message iov
 * @length: length of the iov message
 *
 * Description: This procedure append the user data in the fragment part
 * of the skb if any page alloc fails user this procedure returns  -ENOMEM
 */
```

把用户态数据 copy 到 skb 的 paged data 部分，需要 alloc page

skb_split，skb_split_inside_header，skb_split_no_header

把 skb 分成两个，skb 和 skb1；skb 的长度为 len，skb1 为

- skb_split：如果 skb_headlen(skb) > len，调用 skb_split_inside_header，即在线性区里就可以完成分割；否则新的 skb 将保留全部线性区，会在非线性区完成分割，调用 skb_split_no_header；
- skb_split_inside_header：
  1. 首先把 skb_put(skb1, skb_headlen() - len)，在 skb1 的 tailroom 里预留出 skb 线性区里超过 len 部分数据的长度；然后从 skb->data + len 开始把这部分数据 copy 到 skb1->tail 指示的位置；
  2. 然后把 skb 的 paged data 都 copy 给 skb1，直接移交，不用调用 get_page 增加 page 的引用计数；并且认为 skb1 以前都没有 paged data。
  3. 修改 skb，skb1 里的 len，data_len 和 tail 指针。
- skb_split_no_header：从 skb 的 paged data 开始，计算出足够 len 长度的 frags，保留给 skb，然后把剩余的移交给 skb1，并且也认为 skb1 以前都没有 paged data。

skb 队列相关操作

skb_dequeue, skb_dequeue_tail, skb_insert, skb_queue_purge, skb_queue_head, skb_queue_tail, skb_unlink, skb_append

暂时不太关注。

参考文献

- 《Linux 网络子系统：sk_buffer 详细分析》 http://blog.csdn.net/rstevens/archive/2007/04/10/1559447.aspx
- 《How SKBs work》：http://vger.kernel.org/~davem/skb.html, skb_data.html
- 《Understanding Linux Network Internals》