Linux 網路子系統

sk_buffer 詳細分析

作者： 小馬哥 rstevens (rstevens2008@hotmail.com)

1. 定義

Packet： 通過網卡收發的報文，包括鏈路層、網路層、傳輸層的協定頭和攜帶的資料

Data Buffer：用於存儲 packet 的記憶體空間

SKB： struct sk_buffer 的簡寫

2. 概述

Struct sk_buffer 是 linux TCP/IP stack 中，用於管理 Data Buffer 的結構。Sk_buffer 在資料包的發送和接收中起著重要的作用。

爲了提高網路處理的性能，應儘量避免資料包的拷貝。Linux 內核開發者們在設計 sk_buffer 結構的時候，充分考慮到這一點。目前 Linux 協定棧在接收資料的時候，需要拷貝兩次：資料包進入網卡驅動後拷貝一次，從內核空間遞交給用戶空間的應用時再拷貝一次。

Sk_buffer 結構隨著內核版本的升級，也一直在改進。

學習和理解 sk_buffer 結構，不僅有助於更好的理解內核代碼，而且也可以從中學到一些設計技巧。

3. Sk_buffer 定義

```
struct sk_buff {
 struct sk_buff  *next;
 struct sk_buff  *prev;
 struct sock  *sk;
 struct skb_timeval  tstamp;
 struct net_device  *dev;
 struct net_device  *input_dev;

 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;

 struct  dst_entry    *dst;
 struct  sec_path  *sp;
 char  cb[40];

 unsigned int  len,
 data_len,
 mac_len,
 csum;
 __u32  priority;
```

```
    __u8  local_df:1,
    cloned:1,
    ip_summed:2,
    nohdr:1,
    nfctinfo:3;
    __u8  pkt_type:3,
    fclone:2;
    __be16  protocol;
    void  (*destructor)(struct sk_buff *skb);

    /* These elements must be at the end, see alloc_skb() for details.  */
    unsigned int  truesize;
    atomic_t  users;
    unsigned char  *head,
     *data,
     *tail,
     *end;
};
```

4. 成員變數

struct skb_timeval tstamp;

此變數用於記錄 packet 的到達時間或發送時間。由於計算時間有一定開銷，因此只在必要時才使用此變數。需要記錄時間時，調用 net_enable_timestamp()，不需要時，調用 net_disable_timestamp() 。

tstamp 主要用於包過濾，也用於實現一些特定的 socket 選項，一些 netfilter 的模組也要用到這個域。

- **struct net_device *dev;**
- **struct net_device *input_dev;**

這幾個變數都用於跟蹤與 packet 相關的 device。由於 packet 在接收的過程中，可能會經過多個 virtual driver 處理，因此需要幾個變數。
接收資料包的時候， dev 和 input_dev 都指向最初的 interface，此後，如果需要被 virtual driver 處理，那麼 dev 會發生變化，而 input_dev 始終不變。

(These three members help keep track of the devices assosciated with a packet. The reason we have three different device pointers is that the main **'skb->dev'** member can change as we encapsulate and decapsulate via a **virtual device**.
So if we are receiving a packet from a device which is part of a bonding device instance, initially 'skb->dev' will be set to point the real underlying bonding slave. When the packet enters the networking (via 'netif_receive_skb()') we save 'skb->dev' away in 'skb->real_dev' and update 'skb->dev' to point to the bonding device.
Likewise, the **physical device** receiving a packet always records itself in **'skb->input_dev'**. In this way, no matter how many layers of virtual devices end up being decapsulated, 'skb->input_dev' can always be used to find the top-level device that actually received this packet from the network.)

char cb[40];

此陣列作爲 SKB 的控制塊，具體的協議可用它來做一些私有用途，例如 TCP 用這個控制塊保存序列號和重傳狀態。

unsigned int len, data_len, mac_len, csum;

‘len’ 表示此 SKB 管理的 Data Buffer 中資料的總長度；

通常，Data Buffer 只是一個簡單的線性 buffer，這時候 len 就是線性 buffer 中的資料長度；但在有 ‘paged data’ 情況下， Data Buffer 不僅包括第一個線性 buffer ，還包括多個 page buffer；這種情況下， ‘data_len’ 指的是 page buffer 中資料的長度，’ len’ 指的是線性 buffer 加上 page buffer 的長度；len － data_len 就是線性 buffer 的長度。

‘mac_len’ 指 MAC 頭的長度。目前，它只在 IPSec 解封裝的時候被使用。將來可能從 SKB 結構中去掉。

‘csum’ 保存 packet 的校驗和。

(Finally, 'csum' holds the checksum of the packet. When building send packets, we copy the data in from userspace and calculate the 16-bit two's complement sum in parallel for performance. This sum is accumulated in 'skb->csum'. This helps us compute the final checksum stored in the protocol packet header checksum field. This field can end up being ignored if, for example, the device will checksum the packet for us.

On input, the 'csum' field can be used to store a checksum calculated by the device. If the device indicates 'CHECKSUM_HW' in the SKB 'ip_summed' field, this means that 'csum' is the two's complement checksum of the entire packet data area starting at 'skb->data'. This is generic enough such that both IPV4 and IPV6 checksum offloading can be supported.)

```
__u32                   priority;
```

“priority” 用於實現 QoS，它的值可能取之於 IPv4 頭中的 TOS 域。Traffic Control 模組需要根據這個域來對 packet 進行分類，以決定調度策略。

```
__u8                    local_df:1,
                        cloned:1,
                        ip_summed:2,
                        nohdr:1,
                        nfctinfo:3;
```

爲了能迅速的引用一個 SKB 的資料，
當 clone 一個已存在的 SKB 時，會產生一個新的 SKB，但是這個 SKB 會共用已有 SKB 的資料區。
當一個 SKB 被 clone 後，原來的 SKB 和新的 SKB 結構中，”cloned” 都要被設置爲 1。

(The 'local_df' field is used by the IPV4 protocol, and when set allows us to locally fragment frames which have already been fragmented. This situation can arise, for example, with IPSEC.
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.
The type of the packet (basically, who is it for), is stored in the 'pkt_type' field. It takes on one of the 'PACKET_*' values defined in the 'linux/if_packet.h' header file. For example, when an incoming ethernet frame is to a destination MAC address matching the MAC address of the ethernet device it arrived on, this field will be set to 'PACKET_HOST'. When a broadcast frame is received, it will be set to 'PACKET_BROADCAST'. And likewise when a multicast packet is received it will be set to 'PACKET_MULTICAST'.
The 'ip_summed' field describes what kind of checksumming assistence the card has provided for a receive packet. It takes on one of three values: 'CHECKSUM_NONE' if the card provided no checksum assistence, 'CHECKSUM_HW' if the two's complement checksum over the entire packet has been provides in 'skb->csum', and 'CHECKSUM_UNNECESSARY' if it is not necessary to verify the checksum of this packet. The latter usually occurs when the packet is received over the loopback device. 'CHECKSUM_UNNECESSARY' can also be used when the device only provides a 'checksum OK' indication for receive packet checksum offload.)

```
void                    (*destructor)(struct sk_buff *skb);
unsigned int            truesize;
```

一個 SKB 所消耗的記憶體包括 SKB 本身和 data buffer。
truesize 就是 data buffer 的空間加上 SKB 的大小。
struct sock 結構中，有兩個域，用於統計用於發送的記憶體空間和用於接收的記憶體空間，它們是：

rmem_alloc
wmem_alloc

另外兩個域則統計接收到的資料包的總大小和發送的資料包的總大小。

rcvbuf

sndbuf

rmem_alloc 和 rcvbuf，wmem_alloc 和 sndbuf 用於不同的目的。

當我們收到一個資料包後，需要統計這個 socket 總共消耗的記憶體，這是通過 skb_set_owner_r() 來做的。

```
static inline void skb_set_owner_r(struct sk_buff *skb, struct sock *sk)
{
        skb->sk = sk;
        skb->destructor = sock_rfree;
        atomic_add(skb->truesize, &sk->sk_rmem_alloc);
}
```

最後，當釋放一個 SKB 後，需要調用 skb->destruction() 來減少 rmem_alloc 的值。
同樣，在發送一個 SKB 的時候，需要調用 skb_set_owner_w() ，

```
static inline void skb_set_owner_w(struct sk_buff *skb, struct sock *sk)
{
        sock_hold(sk);
        skb->sk = sk;
        skb->destructor = sock_wfree;
        atomic_add(skb->truesize, &sk->sk_wmem_alloc);
}
```

在釋放這樣的一個 SKB 的時候，需要 調用 sock_free()

```
void sock_wfree(struct sk_buff *skb)
{
        struct sock *sk = skb->sk;

        /* In case it might be waiting for more memory. */
        atomic_sub(skb->truesize, &sk->sk_wmem_alloc);
        if (!sock_flag(sk, SOCK_USE_WRITE_QUEUE))
                sk->sk_write_space(sk);
        sock_put(sk);
}
```

(Another subtle issue is worth pointing out here. For receive buffer accounting, we do not grab a reference to the socket (via 'sock_hold()'), because the socket handling code will always make sure to free up any packets in it's receive queue before allowing the socket to be destroyed. Whereas for send packets, we have to do proper accounting with 'sock_hold()' and 'sock_put()'. Send packets can be freed asynchronously at any point in time. For example, a packet could sit in a devices transmit queue for a long time under certain conditions. If, meanwhile, the socket is closed, we have to keep the socket reference around until SKBs referencing that socket are liberated.)

```
unsigned char                    *head,
                                 *data,
                                 *tail,
                                 *end;
```

SKB 對 Data Buffer 的巧妙管理，就是靠這四個指標實現的。

下圖展示了這四個指標是如何管理資料 buffer 的：
Head 指向 buffer 的開始，end 指向 buffer 結束。 Data 指向實際資料的開始，tail 指向實際資料的結束。這四個指針將整個 buffer 分成三個區：

Packet data:這個空間保存的是真正的資料
Head room：處於 packet data 之上的空間，是一個空閒區域
Tail room：處於 packet data 之下的空間，也是空閒區域。

由於 TCP/IP 協定族是一種分層的協定，傳輸層、網路層、鏈路層，都有自己的協定頭，因此 TCP/IP 協定棧對於資料包的處理是比較複雜的。爲了提高處理效率，避免資料移動、拷貝，sk_buffer 在對資料 buffer 管理的時候，在 packet data 之上和之下，都預留了空間。如果需要增加協定頭，只需要從 head room 中拿出一塊空間即可，而如果需要增加資料，則可以從 tail room 中獲得空間。這樣，整個記憶體只分配一次空間，此後 協議的處理，只需要挪動指針。

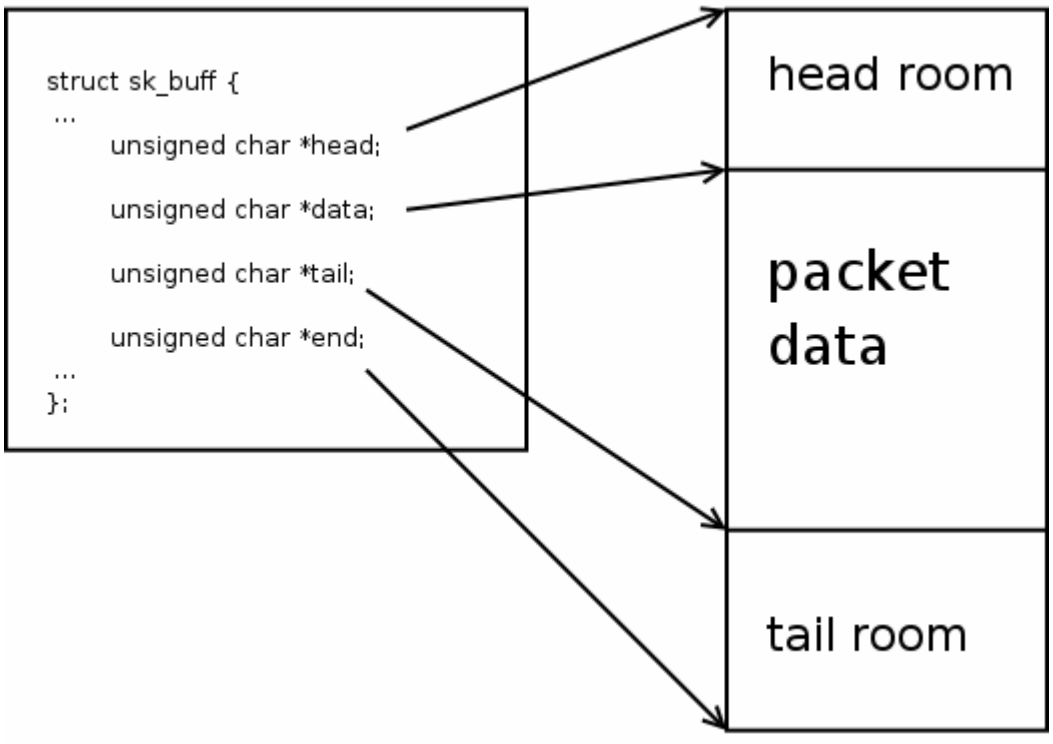
5. Sk_buffer 對記憶體的管理

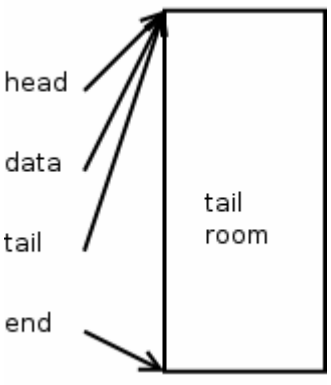
我們以構造一個用於發送的資料包的過程，來理解 sk_buffer 是如何管理記憶體的。

5.1. 構造 Skb_buffer

alloc_skb() 用於構造 skb_buffer，它需要一個參數，指定了存放 packet 的空間的大小。
構造時，不僅需要創建 skb_buffer 結構本身，還需要分配空間用於保存 packet。

skb = alloc_skb(len, GFP_KERNEL);





上圖是在調用完 alloc_skb() 後的情況：

head, data, tail 指向 buffer 開始，end 指向 buffer 結束，整個 buffer 都被當作 tail room。Sk_buffer 當前的資料長度是 0。

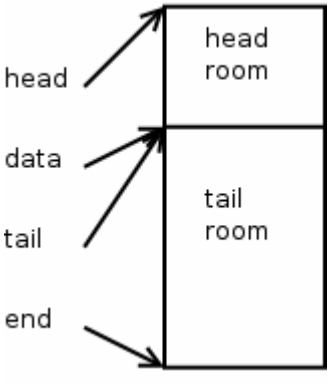
5.2. 爲 protocol header 留出空間

通常，當構造一個用於發送的資料包時，需要留出足夠的空間給協議頭，包括 TCP/UDP header, IP header 和鏈路層頭。

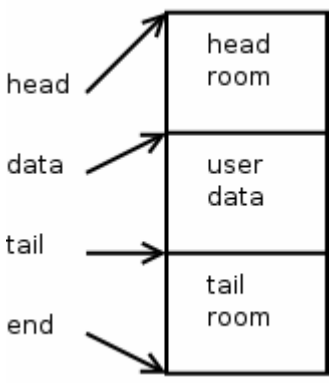
對 IPv4 資料包，可以從 sk->sk_prot->max_header 知道協議頭的最大長度。

skb_reserve(skb, header_len);

上圖是調用 skb_reserver() 後的情況



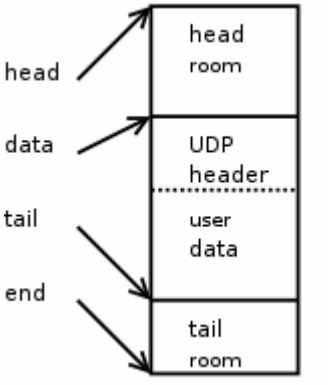
5.3. 將用戶空間資料拷貝到 buffer 中



首先通過 skb_put(skb, user_data_len) ，從 tail room 中留出用於保存資料的空間
然後通過 csum_and_copy_from_user() 將資料從用戶空間拷貝到這個空間中。

5.4. 構造 UDP 協議頭

通過 skb_push() ，向 head room 中要一塊空間
然後在此空間中構造 UDP 頭。



5.5. 構造 IP 頭

通過 skb_push() ，向 head room 中要一塊空間
然後在此空間中構造 IP 頭。

6. Sk_buffer 的秘密

當調用 alloc_skb() 構造 SKB 和 data buffer 時，需要的 buffer 大小是這樣計算的：

data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);

除了指定的 size 以外，還包括一個 struct skb_shared_info 結構的空間大小。也就是說，當調用 alloc_skb(size) 要求分配 size 大小的 buffer 的時候，同時還創建了一個 skb_shared_info 。

這個結構定義如下：

```
struct skb_shared_info {
  atomic_t  dataref;
  unsigned int  nr_frags;
  unsigned short  tso_size;
  unsigned short  tso_segs;
  struct sk_buff  *frag_list;
  skb_frag_t  frags[MAX_SKB_FRAGS];
};
```

我們只要把 end 從 char* 轉換成 skb_shared_info* ，就能訪問到這個結構
Linux 提供一個宏來做這種轉換：

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

那麼，這個隱藏的結構用意何在？
它至少有兩個目的：

1、 用於管理 paged data
2、 用於管理分片

接下來分別研究 sk_buffer 對 paged data 和分片的處理。

7. 對 paged data 的處理

某些情況下，希望能將保存在檔中的資料，通過 socket 直接發送出去，這樣，避免了把資料先從檔拷貝到緩衝區，從而提高了效率。
Linux 採用一種 “paged data” 的技術，來提供這種支援。這種技術將檔中的資料直接被映射爲多個 page。

Linux 用 struct skb_frag_strut 來管理這種 page：

```
typedef struct skb_frag_struct skb_frag_t;

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

並在 shared info 中，用陣列 frags[] 來管理這些結構。

如此一來，sk_buffer 就不僅管理著一個 buffer 空間的資料了，它還可能通過 share info 結構管理一組保存在 page 中的資料。

在採用 “paged data” 時，data_len 成員派上了用場，它表示有多少資料在 page 中。因此，如果 data_len 非 0，這個 sk_buffer 管理的資料就是“非線性”的。
Skb->len – skb->data_len 就是非 paged 數據的長度。

在有 “paged data” 情況下， skb_put()就無法使用了，必須使用 pskb_put() 。。。

8. 對分片的處理

9. SKB 的管理函數

9.1. Data Buffer 的基本管理函數

unsigned char *skb_put(struct sk_buff *skb, unsigned int len)

“推”入數據

在 buffer 的結束位置，增加資料，len 是要增加的長度。

這個函數有兩個限制，需要調用者自己注意，否則後果由調用者負責

1）、不能用於 “paged data” 的情況

這要求調用者自己判斷是否爲 “paged data” 情況

2）、增加新資料後，長度不能超過 buffer 的實際大小。

這要求調用者自己計算能增加的資料大小

unsigned char *skb_push(struct sk_buff *skb, unsigned int len)

“壓”入數據

從 buffer 起始位置，增加資料，len 是要增加的長度。

實際就是將新的資料“壓”入到 head room 中

unsigned char *skb_pull(struct sk_buff *skb, unsigned int len)

“拉”走數據

從 buffer 起始位置，去除數據， len 是要去除的長度。

如果 len 大於 skb->len，那麼，什麼也不做。

在處理接收到的 packet 過程中，通常要通過 skb_pull() 將最外層的協議頭去掉；例如當網路層處理完畢後，就需要將網路層的 header 去掉，進一步交給傳輸層處理。

void skb_trim(struct sk_buff *skb, unsigned int len)

調整 buffer 的大小，len 是調整後的大小。

如果 len 比 buffer 小，則不做調整。

因此，實際是將 buffer 底部的資料去掉。

對於沒有 paged data 的情況，很好處理；

但是有 paged data 情況下，則需要調用 __pskb_trim() 來進行處理。

9.2. “Paged data”和 分片的管理函數

char *pskb_pull(struct sk_buff *skb, unsigned int len)

“拉“走數據

如果 len 大於線性 buffer 中的資料長度，則調用__pskb_pull_tail() 進行處理。

（Q：最後， return skb->data += len; 是否會導致 skb->data 超出了鏈頭範圍？）

int pskb_may_pull(struct sk_buff *skb, unsigned int len)

在調用 skb_pull() 去掉外層協議頭之前，通常先調用此函數判斷一下是否有足夠的資料用於“pull”。

如果線性 buffer 足夠 pull，則返回 1；

如果需要 pull 的資料超過 skb->len，則返回 0；

最後，調用__pskb_pull_tail() 來檢查 page buffer 有沒有足夠的資料用於 pull。

int pskb_trim(struct sk_buff *skb, unsigned int len)

將 Data Buffer 的資料長度調整爲 len

在沒有 page buffer 情況下，等同於 skb_trim()；

在有 page buffer 情況下，需要調用___pskb_trim() 進一步處理。

int skb_linearize(struct sk_buff *skb, gfp_t gfp)

struct sk_buff *skb_clone(struct sk_buff *skb, gfp_t gfp_mask)

‘clone’ 一個新的 SKB。新的 SKB 和原有的 SKB 結構基本一樣，區別在於：

1）、它們共用同一個 Data Buffer

2）、它們的 cloned 標誌都設爲 1

3）、新的 SKB 的 sk 設置爲空

（Q：在什麼情況下用到克隆技術？）

struct sk_buff *skb_copy(const struct sk_buff *skb, gfp_t gfp_mask)

struct sk_buff *pskb_copy(struct sk_buff *skb, gfp_t gfp_mask)

struct sk_buff *skb_pad(struct sk_buff *skb, int pad)

void skb_clone_fraglist(struct sk_buff *skb)

void skb_drop_fraglist(struct sk_buff *skb)

void copy_skb_header(struct sk_buff *new, const struct sk_buff *old)

pskb_expand_head(struct sk_buff *skb, int nhead, int ntail, gfp_t gfp_mask)

int skb_copy_bits(const struct sk_buff *skb, int offset, void *to, int len)

int skb_store_bits(const struct sk_buff *skb, int offset, void *from, int len)

struct sk_buff *skb_dequeue(struct sk_buff_head *list)

struct sk_buff *skb_dequeue(struct sk_buff_head *list)
void skb_queue_purge(struct sk_buff_head *list)
void skb_queue_purge(struct sk_buff_head *list)
void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk)
void skb_unlink(struct sk_buff *skb, struct sk_buff_head *list)
void skb_append(struct sk_buff *old, struct sk_buff *newsk, struct sk_buff_head *list)
void skb_insert(struct sk_buff *old, struct sk_buff *newsk, struct sk_buff_head *list)
int skb_add_data(struct sk_buff *skb, char __user *from, int copy)
struct sk_buff *skb_padto(struct sk_buff *skb, unsigned int len)
int skb_cow(struct sk_buff *skb, unsigned int headroom)

這個函數要對 SKB 的 header room 調整，調整後的 header room 大小是 headroom.
如果 headroom 長度超過當前 header room 的大小，或者 SKB 被 clone 過，那麼需要調整，方法是：
分配一塊新的 data buffer 空間，SKB 使用新的 data buffer 空間，而原有空間的引用計數減 1。在沒有其他使用者的情況下，原有空間被釋放。

struct sk_buff *dev_alloc_skb(unsigned int length)
void skb_orphan(struct sk_buff *skb)
void skb_reserve(struct sk_buff *skb, unsigned int len)
int skb_tailroom(const struct sk_buff *skb)
int skb_headroom(const struct sk_buff *skb)
int skb_pagelen(const struct sk_buff *skb)
int skb_headlen(const struct sk_buff *skb)
int skb_is_nonlinear(const struct sk_buff *skb)
struct sk_buff *skb_share_check(struct sk_buff *skb, gfp_t pri)

如果 skb 只有一個引用者，直接返回 skb
否則 clone 一個 SKB，將原來的 skb->users 減 1，返回新的 SKB

需要特別留意 pskb_pull() 和 pskb_may_pull() 是如何被使用的：

1）、在接收資料的時候，大量使用 pskb_may_pull()，其主要目的是判斷 SKB 中有沒有足夠的資料，例如在 ip_rcv() 中：

```
if (!pskb_may_pull(skb, sizeof(struct iphdr)))
  goto inhdr_error;

iph = skb->nh.iph;
```

它的目的是拿到 IP header，但取之前，先通過 pskb_may_pull() 判斷一下有沒有足夠一個 IP header 的資料。

2）、當我們構造 IP 分組的時候，對於資料部分，通過 put 向下擴展空間（如果一個 sk_buffer 不夠用怎麼分片？）；對於 傳輸層、網路層、鏈路層的頭，通過 push 向上擴展空間；

3）、當我們解析 IP 分組的時候，通過 pull()，從頭開始，向下壓縮空間。

因此，put 和 push 主要用在發送資料包的時候；而 pull 主要用在接收資料包的時候。

10. 各種 header

```
        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;
```

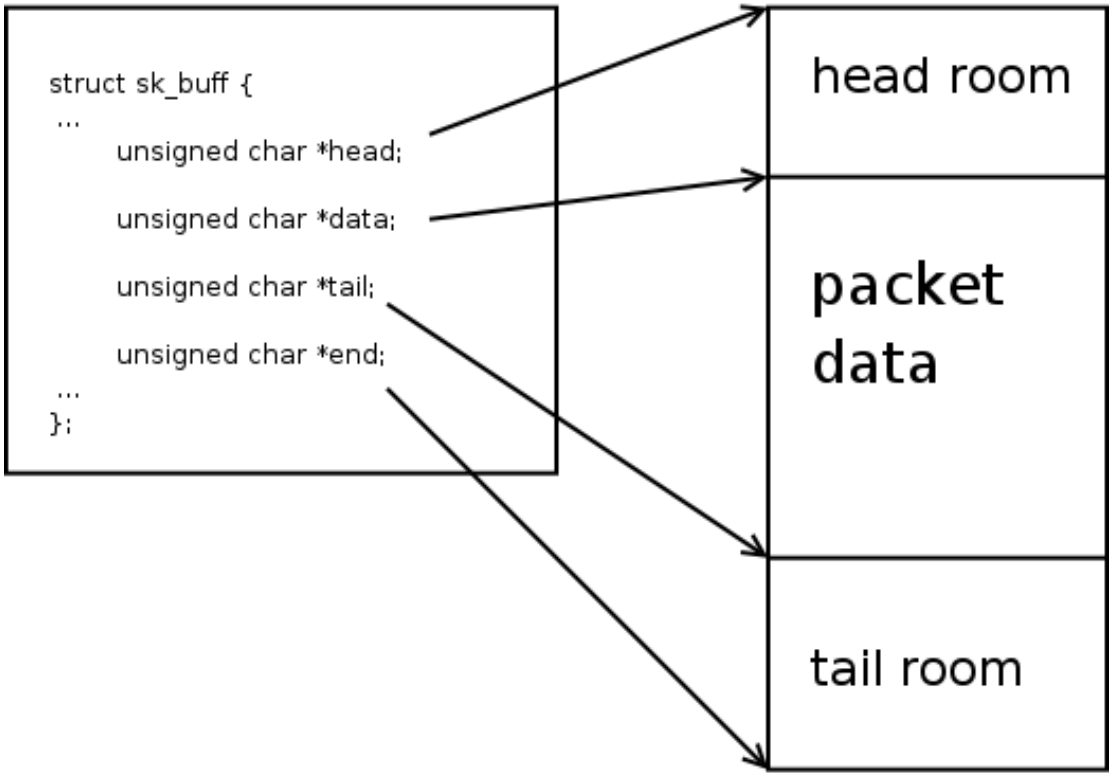
11. 參考資料

1． http://vger.kernel.org/~davem/skb.html
2． Linux 2.4 內核源碼
3． Linux 2.6 內核源碼
4． <<The.Linux.Networking.Architecture_Design.and.Implementation.of.Network.Protocols.in.the.Linux.Kernel >>
5． <<Understanding Linux Network Internals>>
6． << The Linux TCPIP Stack- Networking for Embedded Systems>>

Trackback: http://tb.blog.csdn.net/TrackBack.aspx?PostId=1559447

---

http://vger.kernel.org/~davem/skb_data.html

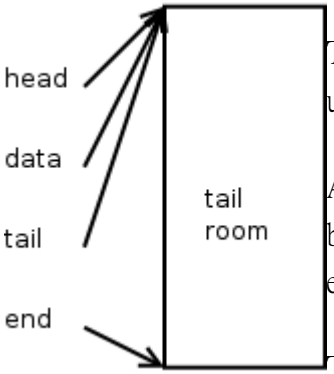


This first diagram illustrates the layout of the SKB data area and where in that area the various pointers in 'struct sk_buff' point.

The rest of this page will walk through what the SKB data area looks like in a newly allocated SKB. How to modify those pointers to add headers, add user data, and pop headers.

Also, we will discuss how page non-linear data areas are implemented. We will also discuss how to work with them.

```
skb = alloc_skb(len, GFP_KERNEL);
```



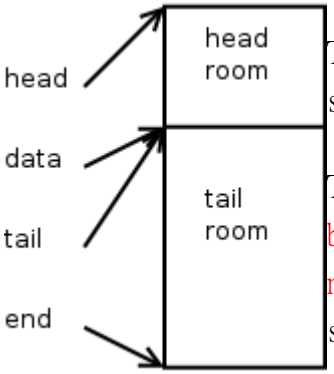
This is what a new SKB looks like right after you allocate it using `alloc_skb()`

As you can see, the head, data, and tail pointers all point to the beginning of the data buffer. And the end pointer points to the end of it. Note that all of the data area is considered tail room.

The length of this SKB is zero, it isn't very interesting since it

doesn't contain any packet data at all. Let's reserve some space for protocol headers using `skb_reserve()`

```
skb_reserve(skb, header_len);
```



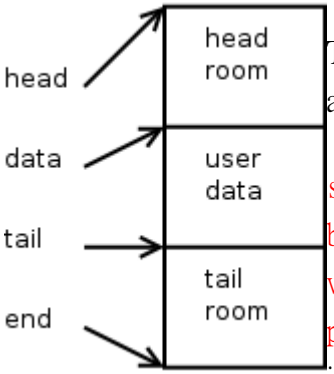
This is what a new SKB looks like right after the `skb_reserve()` call.

Typically, when building output packets, we reserve enough bytes for the maximum amount of header space we think we'll need. Most IPV4 protocols can do this by using the socket value `sk->sk_prot->max_header`.

When setting up receive packets that an ethernet device will DMA into, we typically call `skb_reserve(skb, NET_IP_ALIGN)`. By default NET_IP_ALIGN is defined to '2'. This makes it so that, after the ethernet header, the protocol header will be aligned on at least a 4-byte boundary. Nearly all of the IPV4 and IPV6 protocol processing assumes that the headers are properly aligned.

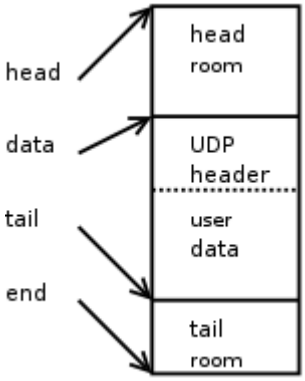
Let's now add some user data to the packet.

```
unsigned char *data = skb_put(skb, user_data_len);
int err = 0;
skb->csum = csum_and_copy_from_user(user_pointer, data,
                                    user_data_len, 0, &err);
if (err)
        goto user_fault;
```



This is what a new SKB looks like right after the user data is added.

`skb_put()` advances 'skb->tail' by the specified number of bytes, it also increments 'skb->len' by that number of bytes as well. This routine must not be called on a SKB that has any paged data. You must also be sure that there is enough tail room in the SKB for the amount of bytes you are trying to put. Both



of these conditions are checked for by skb_put() and an assertion failure will trigger if either rule is violated.

The computed checksum is remembered in 'skb->csum'. Now, it's time to build the protocol headers. We'll build a UDP header, then one for IPV4.

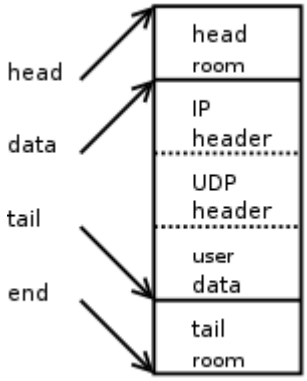
```
        struct inet_sock *inet = inet_sk(sk);
        struct flowi *fl = &inet->cork.fl;
        struct udphdr *uh;

        skb->h.raw = skb_push(skb, sizeof(struct udphdr));
        uh = skb->h.uh
        uh->source = fl->fl_ip_sport;
        uh->dest = fl->fl_ip_dport;
        uh->len = htons(user_data_len);
        uh->check = 0;
        skb->csum = csum_partial((char *)uh,
                                 sizeof(struct udphdr), skb->csum);
        uh->check = csum_tcpudp_magic(fl->fl4_src, fl->fl4_dst,
                                      user_data_len, IPPROTO_UDP,
skb->csum);
        if (uh->check == 0)
                uh->check = -1;
```

This is what a new SKB looks like after we push the UDP header to the front of the SKB.

skb_push() will decrement the 'skb->data' pointer by the specified number of bytes. It will also increment 'skb->len' by that number of bytes as well. The caller must make sure there is enough head room for the push being performed. This condition is checked for by skb_push() and an assertion failure will trigger if this rule is violated.

Now, it's time to tack on an IPV4 header.

```
        struct rtable *rt = inet->cork.rt;
        struct iphdr *iph;
```




```
        skb->nh.raw = skb_push(skb, sizeof(struct iphdr));
        iph = skb->nh.iph;
        iph->version = 4;
        iph->ihl = 5;
        iph->tos = inet->tos;
iph->tot_len = htons(skb->len);
iph->frag_off = 0;
iph->id = htons(inet->id++);
iph->ttl = ip_select_ttl(inet, &rt->u.dst);
iph->protocol = sk->sk_protocol; /* IPPROTO_UDP in this case */
iph->saddr = rt->rt_src;
iph->daddr = rt->rt_dst;
ip_send_check(iph);

skb->priority = sk->sk_priority;
skb->dst = dst_clone(&rt->u.dst);
```

This is what a new SKB looks like after we push the IPv4 header to the front of the SKB.

Just as above for UDP, skb_push() decrements 'skb->data' and increments 'skb->len'. We update the 'skb->nh.raw' pointer to the beginning of the new space, and build the IPv4 header.

This packet is basically ready to be pushed out to the device once we have the necessary information to build the ethernet header (from the generic neighbour layer and ARP).

Things start to get a little bit more complicated once paged data begins to be used. For the most part the ability to use [page, offset, len] tuples for SKB data came about so that file system file contents could be directly sent over a socket. But, as it turns out, it is sometimes beneficial to use this for nomal buffering of process sendmsg() data.

It must be understood that once paged data starts to be used on an SKB, this puts a specific restriction on all future SKB data area operations. In particular, it is no longer possible to do skb_put() operations.

We will now mention that there are actually two length variables assosciated with an SKB, len and data_len. The latter only comes into play when there is paged data in the SKB. skb->data_len tells how many bytes of paged data there are in the SKB. From this we can derive a few more things:

- The existence of paged data in an SKB is indicated by skb->data_len being non-zero. This is codified in the helper routine skb_is_nonlinear() so that it the function you should use to test this.
- The amount of non-paged data at skb->data can be calculated as skb->len - skb->data_len. Again, there is a helper routine already defined for this called skb_headlen() so please use that.

The main abstraction is that, when there is paged data, the packet begins at skb->data for skb_headlen(skb) bytes, then continues on into the paged data area for skb->data_len bytes. That is why it is illogical to try and do an skb_put(skb) when there is paged data. You have to add data onto the end of the paged data area instead.

Each chunk of paged data in an SKB is described by the following structure:

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

There is a pointer to the page (which you must hold a proper reference to), the offset within the page where this chunk of paged data starts, and how many bytes are there.

The paged frags are organized into an array in the shared SKB area, defined by this structure:

```
#define MAX_SKB_FRAGS (65536/PAGE_SIZE + 2)

struct skb_shared_info {
        atomic_t dataref;
        unsigned int    nr_frags;
        unsigned short  tso_size;
        unsigned short  tso_segs;
        struct sk_buff *frag_list;
        skb_frag_t      frags[MAX_SKB_FRAGS];
};
```

The nr_frags member states how many frags there are active in the frags[] array. The tso_size and tso_segs is used to convey information to the device driver for TCP segmentation offload. The frag_list is used to maintain a chain of SKBs organized for fragmentation purposes, it is _not_ used for maintaining paged data. And finally the frags[] holds the frag descriptors themselves.

A helper routine is available to help you fill in page descriptors.

```
void skb_fill_page_desc(struct sk_buff *skb, int i,
                        struct page *page,
                        int off, int size)
```

This fills the i'th page vector to point to page at offset off of size size. It also updates the nr_frags member to be one past i.

If you wish to simply extend an existing frag entry by some number of bytes, increment the size member by that amount.

With all of the complications imposed by non-linear SKBs, it may seem difficult to inspect areas of a packet in a straightforward way, or to copy data out from a packet into another buffer. This is not the case. There are two helper routines available which make this pretty easy.

First, we have:

---

void *skb_header_pointer(const struct sk_buff *skb, int offset, int len, void *buffer)

You give it the SKB, the offset (in bytes) to the piece of data you are interested in, the number of bytes you want, and a local buffer which is to be used _only_ if the data you are interested in resides in the non-linear data area.

You are returned a pointer to the data item, or NULL if you asked for an invalid offset and len parameter. This pointer could be one of two things. First, if what you asked for is directly in the skb->data linear data area, you are given a direct pointer into there. Else, you are given the buffer pointer you passed in.

Code inspecting packet headers on the output path, especially, should use this routine to read and interpret protocol headers. The netfilter layer uses this function heavily.

For larger pieces of data other than protocol headers, it may be more appropriate to use the following helper routine instead.

```
int skb_copy_bits(const struct sk_buff *skb, int offset,
                  void *to, int len);
```

This will copy the specified number of bytes, and the specified offset, of the given SKB into the 'to' buffer. This is used for copies of SKB data into kernel buffers, and therefore it is not to be used for copying SKB data into userspace. There is another helper routine for that:

```
int skb_copy_datagram_iovec(const struct sk_buff *from,
                            int offset, struct iovec *to,
                            int size);
```

Here, the user's data area is described by the given IOVEC. The other parameters are nearly identical to those passed in to `skb_copy_bits()` above.