LINUX 网络协议栈实现分析（一）
SKBUFF 的实现

本文是我尝试分析 LINUX 网络协议栈实现的一系列文章中的第一篇，主要分析 LINUX 网络协议栈中 SKBUFF 的实现。分析以 LINUX2.2.x 为基础，同时也包括了相同的描述对象在 LINUX2.4.x 中的新变化。本文引用的代码的版本分别是：LINUX2.2.25，LINUX2.4.20。

1 简介

了解网络协议栈的人都知道，网络协议栈是一个有层次的软件结构，层与层之间通过预定的接口传递网络报文。网络报文中包含了在协议栈各层使用到的各种信息。网络报文的长度是不固定的，因此采用什么样的数据结构来存储这些网络报文就显得非常重要。在 BSD 的实现中，采用的数据结构是 mbuf，它所能存储的数据的长度是固定的，如果一个网络报文需要多个 mbuf，这些 mbuf 链接成一个链表。所以同一个网络报文里的数据在内存中的存储可能是不连续的。在 LINUX 的实现中，同一个网络报文的数据在内存中是连续存放的，每个网络报文都有一个控制结构，叫做 sk_buff。当然，这只是在 LINUX2.2.x 里面的情况，sk_buff 在 LINUX2.4.x 有一点变化，将会在下面讲到。

2 LINUX2.2.x 中的 SKBUFF

2.1 sk_buff 的定义

前面提到，sk_buff 是一个控制结构，通过它，才可以访问网络报文里的各种数据。所以在分配网络报文存储空间时，同时也分配它的控制结构 sk_buff。在这个控制结构里，有指向网络报文的指针，也有描述网络报文的变量。下面是 sk_buff 的定义，依次注释如下：

```
struct sk_buff {
        struct sk_buff  * next;
        struct sk_buff  * prev;
        struct sk_buff_head * list;
```

以上三个变量将 sk_buff 链接到一个双向循环链表中，链表的结构会在后面讲到。

```
        struct sock     *sk;
```

此报文所属的 sock 结构，此值在本机发出的报文中有效，从网络设备收到的报文此值为空。

```
        struct timeval  stamp;          //此报文收到时的时间
        struct device   *dev;           //收到此报文的网络设备

        union
        {
                struct tcphdr   *th;
                struct udphdr   *uh;
                struct icmphdr *icmph;
                struct igmphdr *igmph;
                struct iphdr    *ipiph;
                struct spxhdr   *spxh;
                unsigned char  *raw;
        } h;
        union
        {
```

```
        struct iphdr     *iph;
        struct ipv6hdr  *ipv6h;
        struct arphdr    *arph;
        struct ipxhdr    *ipxh;
        unsigned char  *raw;
    } nh;
    union
    {
        struct ethhdr    *ethernet;
        unsigned char  *raw;
    } mac;
```

以上三个 union 结构依次是传输层，网络层，链路层的头部结构指针。这些指针在网络报文进入这一层时被赋值，其中 raw 是一个无结构的字符指针，用于扩展的协议。

```
    struct  dst_entry *dst; //此报文的路由，路由确定后赋此值
    char            cb[48]; //用于在协议栈之间传递参数，参数内容的涵义由
```

使用它的函数确定。

```
    unsigned int    len;
```

此报文的长度，这是指网络报文在不同协议层中的长度，包括头部和数据。在协议栈的不同层，这个长度是不同的。

```
    unsigned char  is_clone,
                   cloned,
```

以上两个变量描述此控制结构是否是 clone 的控制结构。一个网络报文可以对应多个控制结构，其中只有一个是原始的结构，其他的都是 clone 出来的。由于可能存在多个控制结构，所以在释放网络报文时要确定它所有的控制结构都已被释放。

```
                   pkt_type,
```

网络报文的类型，常见的有 PACKET_HOST，代表发给本机的报文；还有 PACKET_OUTGOING，代表本机发出的报文。

```
    unsigned short protocol;        //链路层协议
    unsigned int    truesize;       //此报文存储区的长度，这个长度是 16 字节
```

对齐的，一般要比报文的长度大。

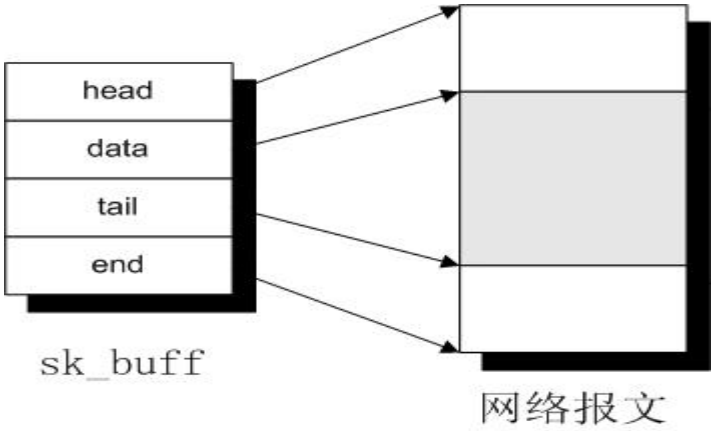
```
    unsigned char  *head;
    unsigned char  *data;
    unsigned char  *tail;
    unsigned char  *end;
```

以上四个变量指向此报文存储区，具体的涵义后面会解释。

```
    __u32           fwmark;         //防火墙在报文中做的标记
};
```

网络报文的存储空间是在网络设备收到网络报文或者应用程序发送数据时分配的，分配的空间以 16 字节对齐。分配成功之后，将网络报文填充到这个存储空间中去。填充时先在存储空间的头部预留了一定数量的空隙，然后将网络报文放到剩余的空间中去。但是网络报文不一定填满整个存储空间，有可能在存储空间的后部还有一定数量的空隙，所以 sk_buff 里面的 head 指针指向存储空间的起始地址，end 指针指向存储空间的结束地址，data 指针指向网络报文的起始地址，tail 指针指向网络报文的结束地址。网络报文在存储空间里的存放的顺序依次是：链路层的头部，网络层的头部，传输层的头部，传输层的数据。在协

议栈的不同层，sk_buff 的指针 data 指向这一层的网络报文的头部。同时，在 sk_buff 里，也有相关的数据结构来表示不同层头部信息。sk_buff 和网络报文之间的关系如图所示：



[图 2.1 sk_buff 与网络报文之间的关系]

（注：控制结构 sk_buff 和网络报文的存储空间是从两个不同的缓存中分配的，所以它们在内存中不是连续存放的。在参考资料里也有一个关于 sk_buff 和网络报文之间的关系的一个图，但是不要误解它们在内存中是连续存放的）

2.2 与 sk_buff 相关的函数

与 sk_buff 相关的函数涉及到网络报文存储结构和控制结构的分配、复制、释放，以及控制结构里的各指针的操作，还有各种标志的检查。重要的函数说明如下：

struct sk_buff *alloc_skb(unsigned int size,int gfp_mask)

分配大小为 size 的存储空间存放网络报文，同时分配它的控制结构。size 的值是 16 字节对齐的，gfp_mask 是内存分配的优先级。常见的内存分配优先级有 GFP_ATOMIC，代表分配过程不能被中断，一般用于中断上下文中分配内存；GFP_KERNEL，代表分配过程可以被中断，相应的分配请求被放到等待队列中。分配成功之后，因为还没有存放具体的网络报文，所以 sk_buff 的 data，tail 指针都指向存储空间的起始地址，len 的大小为 0，而且 is_clone 和 cloned 两个标记的值都是 0。

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

从控制结构 skb 中 clone 出一个新的控制结构，它们都指向同一个网络报文。clone 成功之后，将新的控制结构和原来的控制结构的 is_clone，cloned 两个标记都置位。同时还增加网络报文的引用计数（这个引用计数存放在存储空间的结束地址的内存中，由函数 atomic_t *skb_datarefp(struct sk_buff *skb)访问，引用计数记录了这个存储空间有多少个控制结构）。由于存在多个控制结构指向同一个存储空间的情况，所以在修改存储空间里面的内容时，先要确定这个存储空间的引用计数为 1，或者用下面的拷贝函数复制一个新的存储空间，然后才可以修改它里面的内容。

struct sk_buff *skb_copy(struct sk_buff *skb, int gfp_mask)

复制控制结构 skb 和它所指的存储空间的内容。复制成功之后，新的控制结构和存储空间与原来的控制结构和存储空间相对独立。所以新的控制结构里的 is_clone，cloned 两个标记都是 0，而且新的存储空间的引用计数是 1。

void kfree_skb(struct sk_buff *skb)

释放控制结构 skb 和它所指的存储空间。由于一个存储空间可以有多个控制结构，所以只有在存储空间的引用计数为 1 的情况下才释放存储空间，一般情况下，只释放控制结构 skb。

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

将 tail 指针下移，并增加 skb 的 len 值。data 和 tail 之间的空间就是可以存放网络报文的空间。这个操作增加了可以存储网络报文的空间，但是增加不能使 tail 的值大于 end 的值，skb 的 len 值大于 truesize 的值。

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

将 data 指针上移，并增加 skb 的 len 值。这个操作在存储空间的头部增加了一段可以存储网络报文的空间，上一个操作在存储空间的尾部增加了一段可以存储网络报文的空间。但是增加不能使 data 的值小于 head 的值，skb 的 len 值大于 truesize 的值。

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

将 data 指针下移，并减小 skb 的 len 值。这个操作使 data 指针指向下一层网络报文的头部。

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

将 data 指针和 tail 指针同时下移。这个操作在存储空间的头部预留 len 长度的空隙。

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

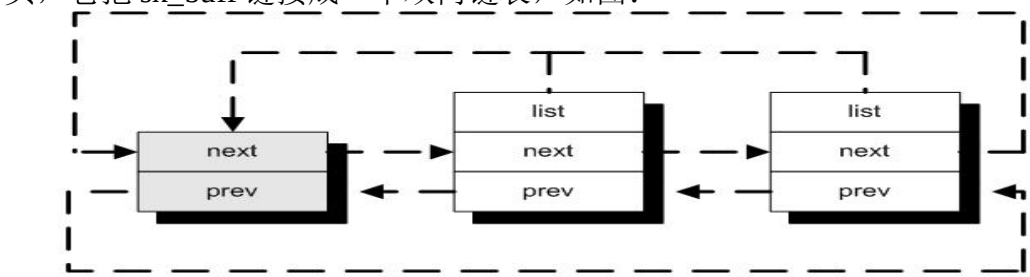
将网络报文的长度缩减到 len。这个操作丢弃了网络报文尾部的填充值。

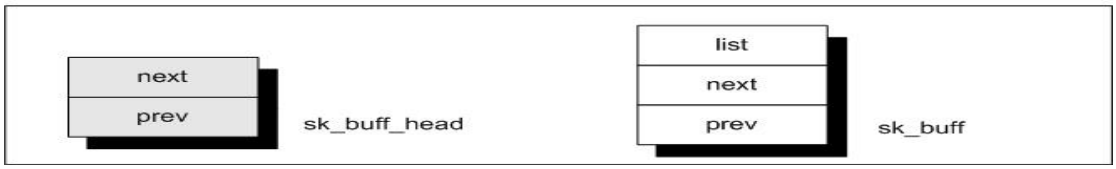
int skb_cloned(struct sk_buff *skb)

判断 skb 是否是一个 clone 的控制结构。如果是 clone 的，它的 cloned 标记是 1，而且它指向的存储空间的引用计数大于 1。

2.3 sk_buff_head 的定义

在网络协议栈的实现中，有时需要把许多网络报文放到一个队列中做异步处理。LINUX 为此定义了相关的数据结构 sk_buff_head。这是一个双向链表的头，它把 sk_buff 链接成一个双向链表，如图：





[图 2.2 sk_buff_head 与 sk_buff 的关系]

2.4 与 sk_buff_head 相关的函数

与链表相关的函数，其功能无非是添加，删除链表上的节点，重要的函数说明如下：

void skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk)

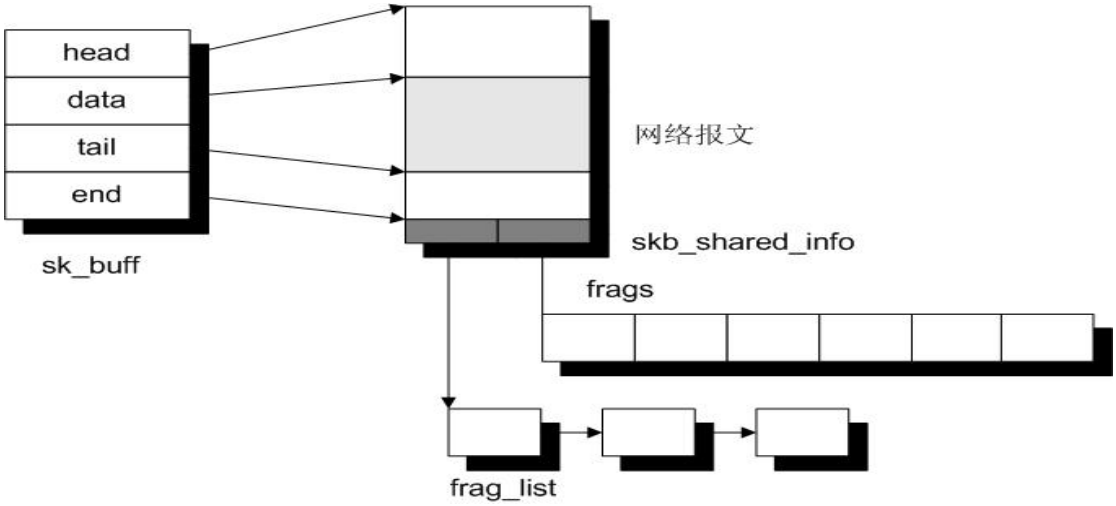
将 newsk 加到链表 list 的头部。

void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk)

将 newsk 加到链表 list 的尾部。
struct sk_buff *skb_dequeue(struct sk_buff_head *list)
从链表 list 的头部取下一个 sk_buff。
struct sk_buff *skb_dequeue_tail(struct sk_buff_head *list)
从链表 list 的尾部取下一个 sk_buff。
skb_insert(struct sk_buff *old, struct sk_buff *newsk)
将 newsk 加到 old 所在的链表上，并且 newsk 在 old 的前面。
void skb_append(struct sk_buff *old, struct sk_buff *newsk)
将 newsk 加到 old 所在的链表上，并且 newsk 在 old 的后面。
void skb_unlink(struct sk_buff *skb)
将 skb 从它所在的链表上取下。
以上的链表操作都是先关中断的。这在中断上下文中是不需要的，所以另外有一套与上面函数同名但是有前缀“__”的函数供运行在中断上下文中的函数调用。

3 LINUX2.4.x 中的 SKBUFF

LINUX2.4.x 中的网络报文在内存中不一定是连续存储的，同一个网络报文有可能被分成几片存放在内存的不同位置，这一点与 LINUX2.2.x 不同（注意不要和 IP 的分片混淆，IP 分片是将一个网络报文分成多个网络报文，这里是将一个网络报文分成几片存放在不同的内存空间中）。一个大概的示意图如下：



[3.1 LINUX2.4.x 的 sk_buff 与网络报文之间的关系]

图中的 frags 是一个数组，frag_list 是一个单向链表。它们所指向的存储空间是一个页的大小（即 4k）。这些额外的存储空间并不是一开始就使用的，只有在 data 所指的存储空间不够用的情况下才使用这些存储空间。以页为单位划分的存储空间有利于和用户空间的程序共享这一块内存的数据。
为了记录网络报文的长度，在 sk_buff 里增加了一个变量 data_len。这个变量记录的是在 frags 和 frag_list 里面存储的报文的长度。原有的变量 len 记录网络报文的总长度。truesize 是 head 所指的存储区的大小。
LINUX2.2.x 里分配，复制，释放 sk_buff 以及存储区的函数在 LINUX2.4.x 中的涵义没有变化，只是在操作时增加了对 frags 和 frag_list 的分配，复制和释放，并且在需要的时候将分散存储的网络报文整合成一个连续存储的网络报文。具体的函数可以参考源代码。

LINUX2.4.x 中对 sk_buff_head 的操作与 LINUX2.2.x 基本相同，只是多加了一个 spinlock 使队列可以在 SMP 的机器上更好地共享。具体地例子可以参考源代码，在此不做赘述。

4 小结

网络报文的存储结构是实现网络协议栈的基础。网络报文在协议栈各层之间传递，因此，如何快速地定位本层关心的数据，并尽量避免在处理时复制网络报文成为提高协议栈性能的关键。本文分析了 LINUX2.2.x 和 LINUX2.4.x 中网络报文的存储结构，以及对存储结构的操作。可以看到，在 LINUX 的协议栈实现中，一般情况下只分配一个网络报文的存储空间，只要不修改网络报文的内容，不同层或不同的处理函数都是通过控制结构 sk_buff 来共享这个网络报文的。只有在需要修改此报文的情况下，才复制一份。这样即节约的存储空间也方便了数据的定位，使得 LINUX 的网络协议栈的性能在应用中表现良好。

5 参考资料

1：《 TCP/IP 详解 卷 2：实现》， Gray R. Wright，W.Richard Stevens，机械工业出版社

2：**Kernel Korner: Network Buffers and Memory Management**，**www.linuxjournal.com**

3：**Linux IP Networking by Glenn Herrin**

4：**Building Into The Linux Network Layer**，**phrack55**