网络代码分析第二部分

——网络子系统在 **IP 层**的收发过程剖析

R.wen (rwen2012@126.com)

一、IP 层数据包处理全景

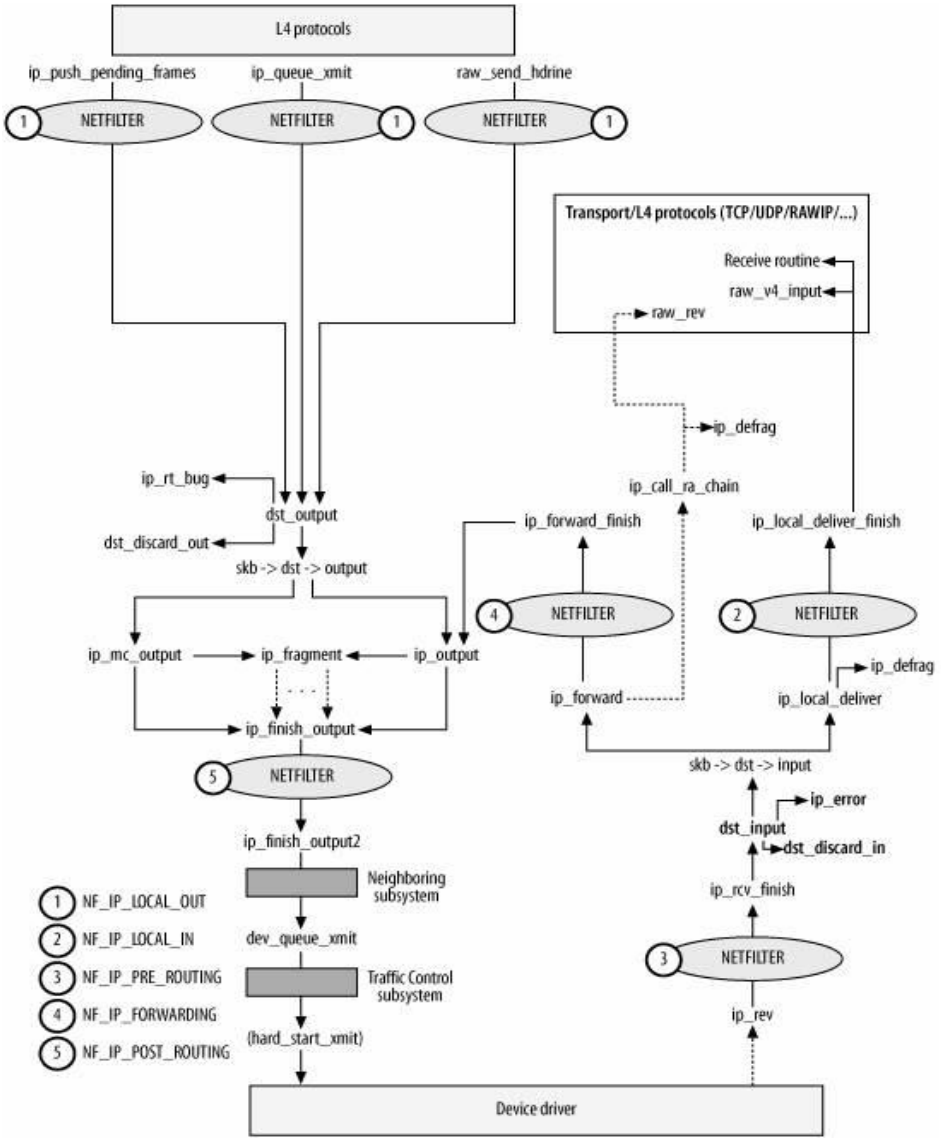

图 1.1

1、接收的全过程

在上一节可以看到，链路层将数据包上传到 IP 层时，由 IP 层相关协议的处理例程处理。对于 IP 协议，这个注册的处理例程是 ip_rcv()，它处理完成后交给 NETFILTE（PRE-ROUTING）R 过滤，再上递给 ip_rcv_finish(), 这个函数根据 skb 包中的路由信息，决定这个数据包是转发还是上交给本机，由此产生两条路径，一为 ip_local_deliver()，它首先检查这个包是否是一个

分片包，如果是，它要调动 ip_defrag()将分片重装，然后再次将包将给 NETFILTER（LOCAL_IN）过滤后，再由 ip_local_deliver_finish()将数据上传到 L4 层，这样就完成了 IP 层的处理；它负责将数据上传，另一路径为 ip_forward()，它负责将数据转发，经由 NETFILTER（FORWARD）过滤后将给 ip_forward_finish()，然后调用 dst_output()将数据包发送出去。

2、发送全过程

由上图可以看到，当 L4 层有数据包待发送时，它将调用 ip_append_data/ip_push_pending_frams(udp,icmp, Raw IP), 或 ip_append_page(UDP), ip_queue_xmit(TCP,SCTP), 或者 raw_send_hdrinc(Raw IP, IGMP)， 它们将这些包交由 NETFILTER（LOLACL_OUT）处理后，然后交给 dst_output，这会根据是多播或单播选择合适的发送函数。如果是单播，它会调用 ip_output()，然后是 ip_finish_output()，这个函数主要是检查待发送的数据包大小是否超过 MTU，如果是，则要首先调用 ip_fragment()将其分片，然后再传给 ip_finish_output2(),由它交给链路层处理了。

二、接收的详细过程

1、我们已经知道，链路层首先将数据包上传给 IP 层的 ip_rcv()函数，这个函数主要做一些检查工作：

首先，这个函数不会接收不是发给这个主机的数据包，如果主机是工作在混杂模式，这个数据包已经由 netif_receive_skb()去完成处理了。注意，这里所说的“不属于”这个主机，是指在这个包目标主机的 MAC 地址不是本机，而不是 L3 层的 ip 地址。所以，它不包括路由的包。

```
if (skb->pkt_type == PACKET_OTHERHOST)
    goto drop;
```

接下来是一个共享的检查，如果是共享的数据包，因为它可能需要修改 skb 中的信息，所以要先复制一个副本，再作进一步的处理。

```
if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL) {
    IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
    goto out;
}
```

再下来就是检查首部的长度是否够长，校检和等等：

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

去掉 padded 部分的长度：

```
len = ntohs(iph->tot_len);
/* Our transport medium may have padded the buffer out. Now we know it
  * is IP we can trim to the true length of the frame.
  * Note this now means skb->len holds ntohs(iph->tot_len).
  */
if (pskb_trim_rcsum(skb, len)) {
```

```
            IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
            goto drop;
        }

return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
                        ip_rcv_finish);
```

2、ip_rcv_finish()

ip_rcv 将数据传给 ip_rcv_finish()继续处理，这个函数工作也比较简单：

它首先查找路由信息，在这里先忽略这部分：

```
        if (skb->dst == NULL) {
            int err = ip_route_input(skb, iph->daddr, iph->saddr, iph->tos,
                                skb->dev);
            …
        }
```

然后就是对 IP 头选项部分的处理了：

```
        if (iph->ihl > 5 && ip_rcv_options(skb))
            goto drop;
```

最后就是 dst_input 了：

```
        return dst_input(skb);
```

dst_input 的工作更为简单，它只是根据 skb 的路由信息调用相应的 input 函数了：

```
        skb->dst->input(skb);
```

由全景图可以看到，这个 input 有可能是 ip_local_deliver()或 ip_forward()。

3、ip_forward()

检查 skb 是否共享，或是否头部预留的空间是否足够存放 L2 的头部，因为在转发这个数据包的时候要将 L2 的头部拷贝进去。

```
        /* We are about to mangle packet. Copy it! */
        if (skb_cow(skb, LL_RESERVED_SPACE(rt->u.dst.dev)+rt->u.dst.header_len))
            goto drop;
```

接着就是 ip_forward_finish()了。

```
        return NF_HOOK(PF_INET, NF_IP_FORWARD, skb, skb->dev, rt->u.dst.dev,
                        ip_forward_finish);
```

ip_forward_finish：

首先处理未处理的头部

```
        if (unlikely(opt->optlen))
            ip_forward_options(skb);
```

然后就是 dst_output：

```
        return dst_output(skb);
```

这个已经是发送部分所做的工作了，我们将在下一部分讨论它。

4、ip_local_deliver()

首先，确定接收到的包是不是分片，如果是，则要将分片重装成一个完整的 IP 包再上传给 L4 层。

```
if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET))
    skb = ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER);
```

然后就是 ip_local_deliver_finish：

```
return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,
            ip_local_deliver_finish);
```

5、ip_local_deliver_finish

至此，已经确定将这个数据包传送给 L4 层，而 L3 层的头已没有作用，所以，先去掉 L 3 的头部：

```
int ihl = skb->nh.iph->ihl*4;
__skb_pull(skb, ihl);   //将 skb->data 指向 L4 的头部
/* Point into the IP datagram, just past the header. */
skb->h.raw = skb->data;   //重新设备 skb->h.raw 值，
                          //它在 receive_skb()中被置为 L3 层头部的位置。
```

接下来就要处理与 L 4 层相关的协议了，首先处理的是 Raw IP，它先查看 raw_v4_htable 有没有注册到这个 L 4 协议的 Raw IP：

```
hash = protocol & (MAX_INET_PROTOS - 1);
raw_sk = sk_head(&raw_v4_htable[hash]);
```

如果有，则要执行 raw_v4_input()，为其提交一份副本：

```
if (raw_sk && !raw_v4_input(skb, skb->nh.iph, hash))
    raw_sk = NULL;
```

// 如何处理接收所有协议的 Raw IP?

当 socket(AF_INET, SOCK_RAW, IPPROTO_RAW)时，它会接收所有协议的数据包，并且 IP_HDRINCL 是默认打开的，即是说应用层要提供 L3 和 L4 层的头。再如，如果是 IPPROTO_TCP 时，它只接收到 TCP 包。而 IP_HDRINCL 是默认不打开的，即系统会处理 L3 的头部。

接着就是对特定 L4 协议的处理：

```
if ((ipprot = rcu_dereference(inet_protos[hash])) != NULL) {
    int ret;
    ret = ipprot->handler(skb);
}
```

它首先查找 inet_protos 数组，看有没有相关的注册的协议，如果有，则执行它的处理例程。

6 、协议的注册

在 inet_init()的时候，系统会注册几个常用的 L4 层协议：

```
        if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
                printk(KERN_CRIT "inet_init: Cannot add ICMP protocol\n");
        if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
                printk(KERN_CRIT "inet_init: Cannot add UDP protocol\n");
        if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0);
                printk(KERN_CRIT "inet_init: Cannot add TCP protocol\n");
```

其中，协议的结构如下：

```
/* This is used to register protocols. */
struct net_protocol {
        int                     (*handler)(struct sk_buff *skb);
        void                    (*err_handler)(struct sk_buff *skb, u32 info);
        int                     (*gso_send_check)(struct sk_buff *skb);
        struct sk_buff          *(*gso_segment)(struct sk_buff *skb, int features);
        int                             no_policy;
};
```

关键之处在于 handler 域，它用于将数据包上传给 L4 层处理，如 UDP 协议初始化如下：

```
static struct net_protocol udp_protocol = {
                                        .handler =   udp_rcv,
                                        .err_handler =    udp_err,
                                        .no_policy =      1,
};
```

再来看看 inet_add_protocol 函数：

```
int inet_add_protocol(struct net_protocol *prot, unsigned char protocol)
{
    int hash, ret;
    hash = protocol & (MAX_INET_PROTOS - 1);

    spin_lock_bh(&inet_proto_lock);
    if (inet_protos[hash]) {
            ret = -1;
    } else {
        inet_protos[hash] = prot;
        ret = 0;
  }
  spin_unlock_bh(&inet_proto_lock);
  return ret;
}
```

可以看到，这个函数只是将待注册的协议填入全局数组，要注意的是，每个协议只能注册一个相关的实例。但是在用户态是，一个协议可以注册多个实例。

二、发送详细过程

由全景图可以看到，L4 层在发送数据时会根据协议的不同调用上面提到的几个辅助函数之一，它们的主要作用是将数据分成合适大小的块然后再传递给 L3 层，理论上 L4 层可以不做这些工作，因为分片对 L4 层来说是透明的，这样做的目的是为了实现快速分片。

1.1、 ip_queue_xmit()

这个函数是由 TCP 协议使用的，由于 L4 层处理了部分分片工作，这个函数的工作主要有以下几个：

a. 查找路由信息：

```
rt = (struct rtable *) skb->dst;
if (rt != NULL)
    goto packet_routed;
… …
```

如果还没有，它则要在路由表中查找相关的 rt，但我们这里不关心路由部分。

b. 填充 IP 头信息，这里用 skb_push 为 IP 头留出空间

```
/* OK, we know where to send it, allocate and build IP header. */
iph = (struct iphdr *) skb_push(skb, sizeof(struct iphdr) + (opt ? opt->optlen : 0));
*((__u16 *)iph)      = htons((4 << 12) | (5 << 8) | (inet->tos & 0xff));
iph->tot_len = htons(skb->len);
if (ip_dont_fragment(sk, &rt->u.dst) && !ipfragok)
        iph->frag_off = htons(IP_DF);
else
        iph->frag_off = 0;
iph->ttl            = ip_select_ttl(inet, &rt->u.dst);
iph->protocol = sk->sk_protocol;
iph->saddr        = rt->rt_src;
iph->daddr        = rt->rt_dst;
skb->nh.iph      = iph;
/* Transport layer set skb->h.foo itself. */
```

b. 创建选项部分和计算校检：

```
if (opt && opt->optlen) {
        iph->ihl += opt->optlen >> 2;
        ip_options_build(skb, opt, inet->daddr, rt, 0);
}
//设置包头的 ID 域
ip_select_ident_more(iph, &rt->u.dst, sk,
                     (skb_shinfo(skb)->gso_segs ?: 1) - 1);

/* Add an IP checksum. */
ip_send_check(iph);
```

最后就是调用 dst_output 了。

1.2、 ip_append_data(ip_append_page)/ ip_push_pending_frames

这是一对由 UDP 等协议使用的一对函数，ip_append_data 是一个比较复杂的函数，它主要将收到的大数据包分成多个小于 MTU（1500）的 skb，为 L3 层要实现的 IP 分片做准备。例如如果待发送的数据包大小为 4000 字节，假设先前 sock 中的队列又非空（sk->sk_write_queue!=NULL），（因为 ip_append_data 可以被 L4 层多次调用，用于添加数据。）并且之前一个 skb 还没填满，剩余大小为 500 字节。这时，当 L4 层调用 ip_append_data 时，它首先将这个剩余的 skb 填满，这里还有一个问题就是关于 scatter/gather IO 的，当 NIC 不支持时，它会直接将数据写到这个 skb->tail 处，但是，如果 NIC 支持这种 IO，它便会将数据写到 frags 所指向的指针中，如果相关的 page 已经填满，它会再分配一个新的 page 用于这个 skb。这一步完成之后，ip_append_data 再次进入下三次循环，每次循环都分配一个 skb，并将数据通过 getfrag 从 L4 层复制下来。在循环结束之前，它通过__skb_queue_tail(&sk->sk_write_queue, skb)，将这个 skb 链入这个 sock 的 sk_write_queue 队列中去。待到这个循环结束时，所有的数据都从 L4 复制到各个 skb，并链入了它的 sk_write_queue 队列。换言之，待发送数据已经在 sk_write_queue 队列中了。

```
static inline void __skb_queue_tail(struct sk_buff_head *list,
                                    struct sk_buff *newsk)
{
        struct sk_buff *prev, *next;

        list->qlen++;
        next = (struct sk_buff *)list;
        prev = next->prev;
        newsk->next = next;
        newsk->prev = prev;
        next->prev    = prev->next = newsk;
}
```

它的循环过程如下，由于代码较长，这里就不列出来了。

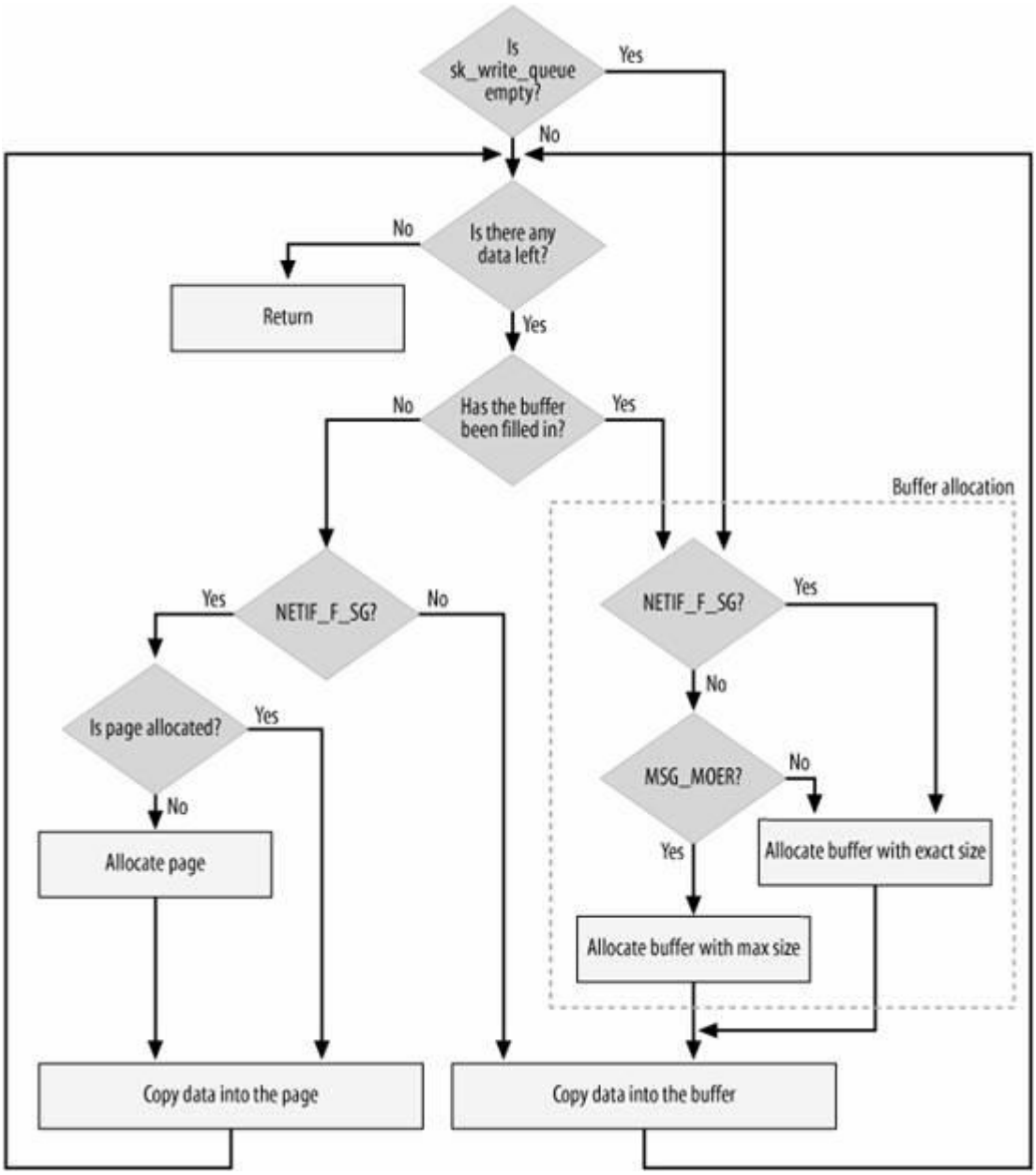

图 2.1

ip_append_page()跟 ip_append_data()是实现同样功能的函数，它们的主要不同在于，ip_append_data()需要将用户空间的数据复制到内核空间，而 ip_append_page()则不需要这个复制，它直接使用用户提供的数据，从而实现了数据的“零拷贝”。

现在来看一下它们之间的差别：

a. 分配 skb 空间：

ip_append_data()：

```
alloclen = datalen + fragheaderlen;
if (atomic_read(&sk->sk_wmem_alloc) <=    2 * sk->sk_sndbuf)
                skb = sock_wmalloc(sk, alloclen + hh_len + 15, 1,
                                   sk->sk_allocation);
```

ip_append_page()：

```
alloclen = fragheaderlen + hh_len + fraggap + 15;
skb = sock_wmalloc(sk, alloclen, 1, sk->sk_allocation);
```

我们可以看到，这两个函数分配的空间大小是不一样的，对于 ip_append_page()，它不为新的数据分配空间，只是分配一些头部所需要的空间，和一个 fraggap，它个值大小为 1~7 个字节，是由于对齐问题从上一个 skb 移动过来的。

b. ip_append_page()只能用于支持 S/G IO 的 NIC，它只是将数据连接到 frags 数组中，而没有 ip_append_data()的复制数据。

```
i = skb_shinfo(skb)->nr_frags;
        if (len > size)
                len = size;
        if (skb_can_coalesce(skb, i, page, offset)) {
                skb_shinfo(skb)->frags[i-1].size += len;
        } else if (i < MAX_SKB_FRAGS) {
                get_page(page);
                skb_fill_page_desc(skb, i, page, offset, len);
        } else {
                err = -EMSGSIZE;
                goto error;
        }
```

当 ip_append_data/ip_append_page 完成后，sock 的 sk_write_queue 队列如下图(a)如示，而当 ip_push_pending_frames()完成后，它如下图(b)所示，即 ip_push_pending_frames()完成了将所有的数据分片组装成一个大的 IP 数据报。它不再是链入 skb->netx，而是 skb_shinfo(skb)->frag_list 域了。

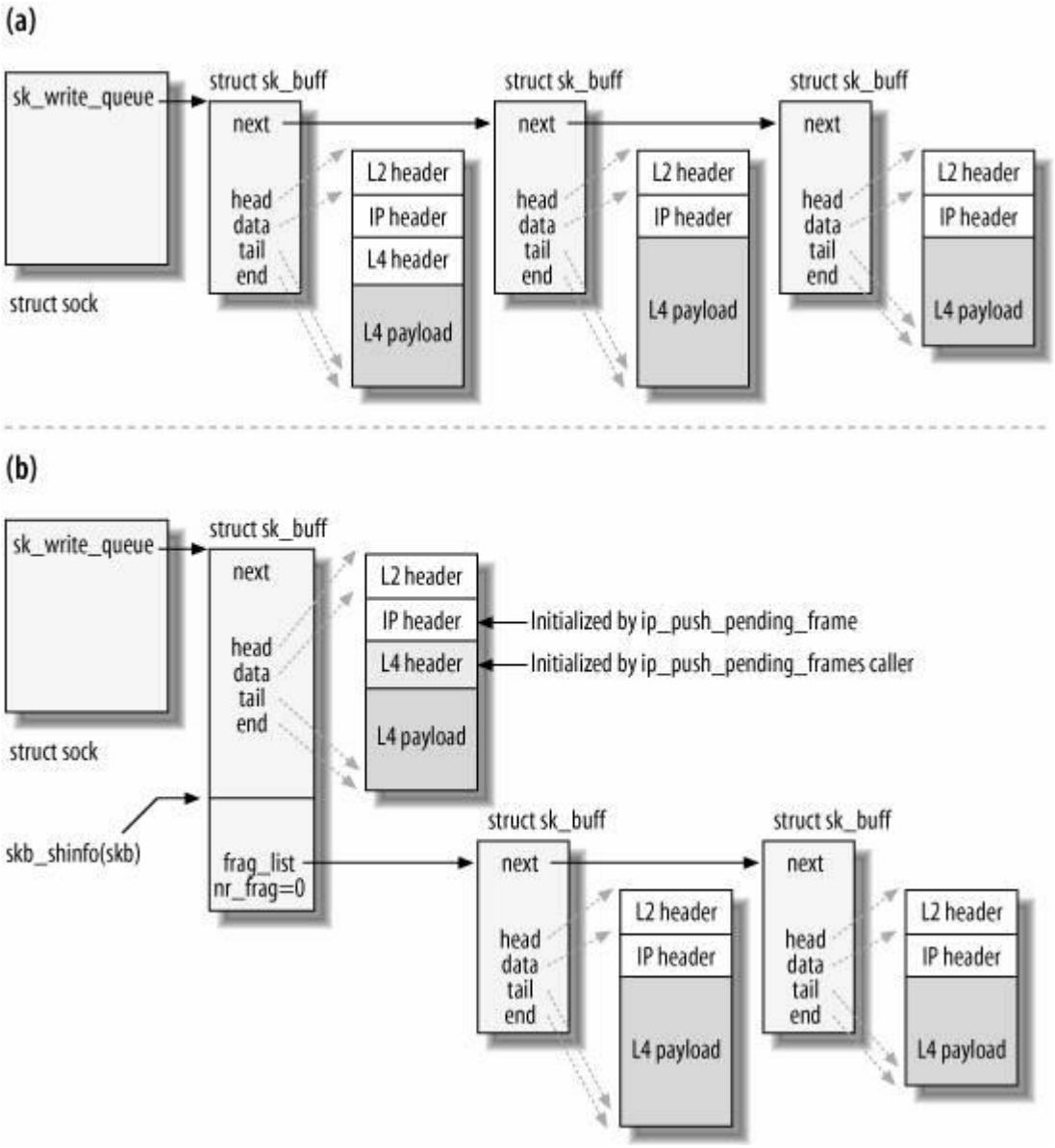

图 2.2

我们看看 ip_push_pending_frames()的实现要点：

它首先将队列 sk_write_queue 里的 skb 数据逐个出队，然后再将它们链入 frag_list 中，由此得到了上图中由(a)到(b)变化的情景。

```
        tail_skb = &(skb_shinfo(skb)->frag_list);
        /* move skb->data to ip header from ext header */
        if (skb->data < skb->nh.raw)
                __skb_pull(skb, skb->nh.raw - skb->data);
        while ((tmp_skb = __skb_dequeue(&sk->sk_write_queue)) != NULL) {
                __skb_pull(tmp_skb, skb->h.raw - skb->nh.raw);
                *tail_skb = tmp_skb;
                tail_skb = &(tmp_skb->next);
                skb->len += tmp_skb->len;
                skb->data_len += tmp_skb->len;
                skb->truesize += tmp_skb->truesize;
                __sock_put(tmp_skb->sk);
                tmp_skb->destructor = NULL;
                tmp_skb->sk = NULL;
        }
```

接下来就是一些值的设置了，如建立 IP 头，IP 选项，计算校检值等，这些跟 ip_queue_xmit() 所做的工作差不多。最后跟 ip_queue_xmit()一样，也是调用 dst_output()完成发送工作。

我们看到，对于 UDP，它需要调用 ip_append_data/ip_append_page 处理很多辅助分片的工作，而对于 TCP，ip_queue_xmit()没有做这份工作，它是由 L4 层去完成的，相对于 ip_append_data/ip_append_page，它有 tcp_sendmsg/tcp_sendpage。

1.3、 dst_output()

由开始的全景图可以看到，对于单播 IP 来说，它执行的是 ip_output()：

```
int ip_output(struct sk_buff *skb)
{
        struct net_device *dev = skb->dst->dev;

        IP_INC_STATS(IPSTATS_MIB_OUTREQUESTS);

        skb->dev = dev;
        skb->protocol = htons(ETH_P_IP);

        return NF_HOOK_COND(PF_INET, NF_IP_POST_ROUTING, skb, NULL, dev,
                            ip_finish_output,
                        !(IPCB(skb)->flags & IPSKB_REROUTED));
}
```

由上，它再执行 ip_finish_output：

```
static inline int ip_finish_output(struct sk_buff *skb)
{
        if (skb->len > dst_mtu(skb->dst) && !skb_is_gso(skb))
                return ip_fragment(skb, ip_finish_output2);
        else
                return ip_finish_output2(skb);
}
```

由 ip_finish_output 可以看到，如果 IP 数据包的长度大于 MTU，它便会先将 IP 分成合适大小的分片，最后调用 ip_finish_output2()，将这个包路由出去。

三、IP 的分片与重装(ip_fragment/ip_defrag)

先来看看产生 IP 分片与重装的地方，这里只考虑主机的情况，而不关心路由。

由上可以看到，当一个 IP 包在 ip_finish_output()中被检查到它的长度大于 MTU，则它要调用 ip_fragment 将这个 IP 包进行分片。而当 IP 分片由 ip_local_deliver()上传给 L4 层时，它要将这些分片暂时保存起来，一旦所有的分片都到达时，它便将一个完整的 IP 包上传给 L4 层。

1、 ip_fragment

当一个包由于太大而需要通过分片传递时，它便过调用这个函数将这个大的数据包分成多个小的 IP 分片。由图 2.2 可以看到，ip_append_data/ip_push_pending_frames 为 IP 分片做了很多辅助工作。当一个 L4 层传递过来的数据包经过 ip_append_data 和 ip_push_pending_frames 处理后，它得到的结果如图 2.2(b)所示，这时，队列中每个 skb 的大小都小于 MTU 减小链路头部的大小（因为发送数据时需要加上链路层头部），因些，ip_fragment 只要将这些 skb 依照第一个 skb 的 IP 头部为每个 skb 加上 IP 头就完成了这个分片工作，这种分片称为“快速分片”：

```
if (skb_shinfo(skb)->frag_list) {
        struct sk_buff *frag;
        int first_len = skb_pagelen(skb); //第一个 skb 长度

        … //省略了一些检查工作

        /* Everything is OK. Generate! */
        //处理第一个 skb
        err = 0;
        offset = 0;
        frag = skb_shinfo(skb)->frag_list;
        skb_shinfo(skb)->frag_list = NULL;
        skb->data_len = first_len - skb_headlen(skb);
        skb->len = first_len;
        iph->tot_len = htons(first_len);
        iph->frag_off = htons(IP_MF);
        ip_send_check(iph);

        for (;;) {
            /* Prepare header of the next frame,
              * before previous one went down. */
            if (frag) {
                frag->ip_summed = CHECKSUM_NONE;
                frag->h.raw = frag->data;
                frag->nh.raw = __skb_push(frag, hlen);
                memcpy(frag->nh.raw, iph, hlen); //为每个分片复制一份 IP 头
```

```
                iph = frag->nh.iph;
                iph->tot_len = htons(frag->len);
                ip_copy_metadata(frag, skb);   //复制其它一些关于 skb 的设置
                if (offset == 0)
                    ip_options_fragment(frag);
                offset += skb->len - hlen;
                iph->frag_off = htons(offset>>3);
                if (frag->next != NULL)
                    iph->frag_off |= htons(IP_MF);
                /* Ready, complete checksum */
                ip_send_check(iph);
            }

            err = output(skb);   //发送函数

            skb = frag;
            frag = skb->next;
            skb->next = NULL;
        }
```

而当 L4 层传递过来的包没有满足快速分片的要求时，它只能使用一般的分片办法了，这个办法原理很简单，假设 L4 传递过来的一个包大小为 4000 字节，而 MTU 大小为 1500，那这个方法就是通过一个循环，每次分配一个 skb，大小分别为 1500，1500，1000（忽略头部与对齐、填充等空间），然后再将原来的 skb 里的 IP 头，部分数据拷贝进新分配的 skb 中去，直至所有的数据都拷贝完成。

```
slow_path:
    left = skb->len - hlen;          /* Space per frame */ //原始长度
    ptr = raw + hlen;           /* Where to start from */
    offset = (ntohs(iph->frag_off) & IP_OFFSET) << 3;
    not_last_frag = iph->frag_off & htons(IP_MF);

    while(left > 0) {
        len = left;
        /* IF: it doesn't fit, use 'mtu' - the data space left */
        if (len > mtu)   //每次拷贝的长度
            len = mtu;
            ……
        if ((skb2 = alloc_skb(len+hlen+ll_rs, GFP_ATOMIC)) == NULL) {
            //出错处理
        }
        //设置新的 skb 头与复制 IP 头
        ip_copy_metadata(skb2, skb);
        skb_reserve(skb2, ll_rs);
```

```
            skb_put(skb2, len + hlen);
            skb2->nh.raw = skb2->data;
            skb2->h.raw = skb2->data + hlen;

            memcpy(skb2->nh.raw, skb->data, hlen);

            //复制数据部分，注意不能简单的 memcpy，因为 skb 中可以存在数据分片
            if (skb_copy_bits(skb, ptr, skb2->h.raw, len))
                BUG();
            left -= len;

            /*
             *    Fill in the new header fields.
             */
            iph = skb2->nh.iph;
            iph->frag_off = htons((offset >> 3));

            if (offset == 0)
                ip_options_fragment(skb);

            if (left > 0 || not_last_frag)
                iph->frag_off |= htons(IP_MF);
            ptr += len;
            offset += len;

            iph->tot_len = htons(len + hlen);
            ip_send_check(iph);
            err = output(skb2);
            ……
}
```

2、 ip_defrag 当 L3 层在将数据包上传给 L4 层的时候，它如果发现这个包只是一个 IP 分片，那么它将调用 ip_defrag 这个函数，将这个分片暂时队列起来，如果所有的分片都到齐，它便将这些分片重装成一个完整的 IP 包再上传给 L4 层。

由下图看到，系统为分片准备了一个 hash 表 ipq_hash，大小为 64，并且每个 ipq 结构代表一个被重装成一个完成 IP 包的所有 IP 分片的队列头，将被重装的 skb 存放在这个 ipq 的 fragments 链表中。注意，同一个 hash 链中的 ipq 结构是相互独立无关的，它们只是 hash 表的冲突链表。如下图所示，ID 为 1234 的 IP 包收到了 2 个分片，它们根据 offset 的值顺次插入到这个 ipq 的 fragments 链表中。

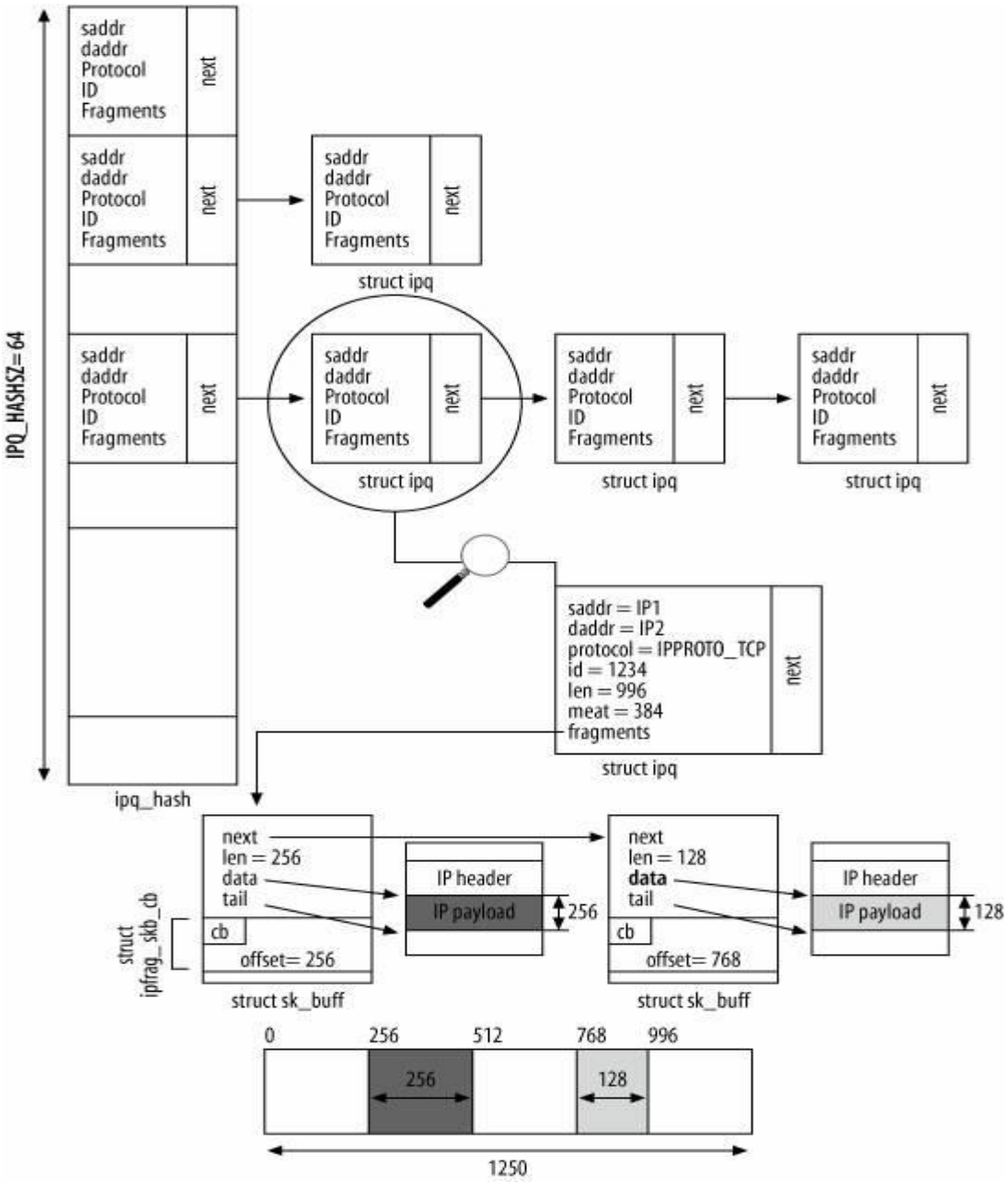

图 3.1

```
/* Process an incoming IP datagram fragment. */
struct sk_buff *ip_defrag(struct sk_buff *skb, u32 user)
{
   struct iphdr *iph = skb->nh.iph;
   struct ipq *qp;
   struct net_device *dev;

   /* Start by cleaning up the memory. */
   if (atomic_read(&ip_frag_mem) > sysctl_ipfrag_high_thresh)
         ip_evictor(); //如果分片的数量超过了 sysctl_ipfrag_high_thresh，
```

```
                                    //则调用这个函数释放掉一些 LRU ipq。

        dev = skb->dev;
        /* Lookup (or create) queue header */
        if ((qp = ip_find(iph, user)) != NULL) {
            struct sk_buff *ret = NULL;
            spin_lock(&qp->lock);
            ip_frag_queue(qp, skb); //将这个 skb 插入到 qp 的 fragment 队列中

            if (qp->last_in == (FIRST_IN|LAST_IN) &&
                    qp->meat == qp->len)      //所有的分片已收到
                    ret = ip_frag_reasm(qp, dev);    //将这些分片重装

            spin_unlock(&qp->lock);
            ipq_put(qp, NULL);
            return ret;
        }
        ……
    }
```

2.1、ip_frag_queue 最主要的工作就是将收到的 skb 插入相应的 ipq，但它比想像中要复杂，因为它要处理很多出错的情况，如收到多个相同的分片，收到重叠的分片等。

先来看看 ipq 结构：

```
    /* Describe an entry in the "incomplete datagrams" queue. */
    struct ipq {
        struct hlist_node list;
        struct list_head lru_list;  /* lru list member                 */
        u32         user;
        __be32          saddr;
        __be32          daddr;
        __be16          id;
        u8          protocol;
        u8          last_in;
    #define COMPLETE        4
    #define FIRST_IN            2
    #define LAST_IN                 1

        struct sk_buff  *fragments;     /* linked list of received fragments*/
        int         len;        /* total length of original datagram*/
        int         meat;
        spinlock_t      lock;
        atomic_t  refcnt;
        struct timer_list timer;    /* when will this queue expire?            */
        struct timeval  stamp;
```

```
    int                     iif;
    unsigned int        rid;
    struct inet_peer *peer;
};
```

这个结构中有两个域 len 和 meat，当有到最后一个分片时，len 值便为原始 IP 包的长度，而不是最后一片时，它的值便是 len= end = offset + skb->len – ihl 与当前 len 值的最大值。而 maet 则为当前 ipq 收到分片的实际长度，所以说，当这个 ipq 已经收到了最后一个包，并且有 meat=len 时，这个包便接收完成了，现在，就可以将它重装成一个完成的 IP 包了。还有它会将每个 skb 中的 IP 头去掉。

2.2、ip_frag_reasm()

这个是将所有 IP 分片重装的函数，重装当然是分片的逆向动作，所以，重装的得到的结果是到 IP 数据包恢复到 ip_fragment 之前的情景。就是图 2.2(b)的情形。而 ip_frag_reasm 得到的包的结构则如图 3.1 中的情景。换言之，ip_frag_reasm()所做的，就是将图 3.1(b)所示的结构转换成图 2.2(b)所示的结构。

```
head = qp->fragments;
//将第一个 skb 的 next 链到 skb_shinfo(head)->frag_list 中
//这就完成了这两个结构的转换
skb_shinfo(head)->frag_list = head->next;
skb_push(head, head->data - head->nh.raw);
atomic_sub(head->truesize, &ip_frag_mem);
//通过循环，将链表中的所有 skb 的长度加到第一个 skb 中，形成一个大的数据包的长度。
for (fp=head->next; fp; fp = fp->next) {
    head->data_len += fp->len;
    head->len += fp->len;
    if (head->ip_summed != fp->ip_summed)
        head->ip_summed = CHECKSUM_NONE;
    else if (head->ip_summed == CHECKSUM_COMPLETE)
        head->csum = csum_add(head->csum, fp->csum);
    head->truesize += fp->truesize;
    atomic_sub(fp->truesize, &ip_frag_mem);
}
……
return head;
```

通过这个转换，得到的 skb 便是一个完整的 IP 数据包，这时，就可以向上层传递了。