PF_RING 源码学习笔记

一、引言

在 PF_RING 相关的文档中，作者宣称极大地提供了包捕获的性能，故利用上班的时间打了点酱油，大致走读了下 PF_RING（v4.1）的源码,根据以往的教训，光走读而不写下点什么，数月过个，忘个精光，这一次写下这个走读笔记，希望效果有所改观。

另外，笔者对kernel理解尚浅，很多地方(比如锁的使用、mmap的实现)都不是很清楚，所以这些地方被一带而过了，等以后有了更深的了解后，再回头补充这个笔记, 对于笔记中的错误和疏漏，也恳请大家指正，邮箱：d00fy@163.com

二、模块初始化

```
static int __init ring_init(void)
{
  int i, rc;

  printk("[PF_RING] Welcome to PF_RING %s ($Revision: %s$)\n"
     "(C) 2004-10 L.Deri <deri@ntop.org>\n",
     RING_VERSION, SVN_REV);
  /* 注册 PF_RING 协议 */
  if((rc = proto_register(&ring_proto, 0)) != 0)
    return(rc);

  /*初始化 4 个双向循环队列*/
  INIT_LIST_HEAD(&ring_table);
  INIT_LIST_HEAD(&ring_cluster_list);
  INIT_LIST_HEAD(&ring_aware_device_list);
  INIT_LIST_HEAD(&ring_dna_devices_list);

  for (i = 0; i < MAX_NUM_DEVICES; i++)
    INIT_LIST_HEAD(&device_ring_list[i]);

  /* 注册 PF_RING socket, 用于用户态和内核态通信，类似于 UNIX 域套接字 */
  sock_register(&ring_family_ops);
  /* 注册设备通知的函数，当网卡 down、up 时，ring_netdev_notifier 将被调用*/
  register_netdevice_notifier(&ring_netdev_notifier);
```

```
  /* Sanity check */
  if(transparent_mode > driver2pf_ring_non_transparent)
    transparent_mode = standard_linux_path;

  printk("[PF_RING] Ring slots      %d\n", num_slots);
  printk("[PF_RING] Slot version    %d\n",RING_FLOWSLOT_VERSION);
  printk("[PF_RING] Capture TX      %s\n",
     enable_tx_capture ? "Yes [RX+TX]" : "No [RX only]");
  printk("[PF_RING] Transparent Mode %d\n",
     transparent_mode);
  printk("[PF_RING] IP Defragment   %s\n",
     enable_ip_defrag ? "Yes" : "No");
  printk("[PF_RING] Initialized correctly\n");

  ring_proc_init();
  register_device_handler();

  pfring_enabled = 1;
  return 0;
}
```

下面分析这 5 双向循环队列的作用：
ring_table: 存放所有的 ring sockets

```
ring_table struct ring_element {
   struct list_head    list;
   struct sock     *sk;
}
```

ring_cluster_list:存放所有的 cluster，每一个 cluster 以 cluster_id 标识，最重要的成员是 sk，包含一组 struct sock *。

```
typedef struct {
struct ring_cluster cluster;
struct list_head list;
} ring_cluster_element;

struct ring_cluster {
u_short           cluster_id; /* 0 = no cluster */
u_short           num_cluster_elements;
enum cluster_type  hashing_mode;
u_short           hashing_id;
struct sock       *sk[CLUSTER_LEN];
};
```

实际上在 PF_RING 中，数据结构 struct ring_opt 和 struct sock 是一一对应的关系，宏

```
#define ring_sk(__sk) ((__sk)->sk_protinfo)
```

sk_protinfo 是一个 void *，很自然地，就是设计用来存储每个协议私有信息的，在 PF_RING 中，指向一个 struct ring_opt。

ring_aware_device_list：存放所有已经注册 PF_RING 的 device,即用户使用接口 pfring_open(char *device，)打开的设备列表。

```
typedef struct {
   struct net_device *dev;
   struct list_head list;
} ring_device_element;
```

ring_dna_devices_list: 存放所有的 dna 设备

```
typedef struct {
  dna_device dev;
  struct list_head list;
} dna_device_list;
```

device_ring_list[MAX_NUM_DEVICES]：对于每一个设备，都维护了在该设备上注册的 ring_opt，因为多个应用程序可能在同一个网卡上捕获报文。

```
typedef struct {
   struct ring_opt *the_ring;
   struct list_head list;
} device_ring_list_element;
```

ring_init() -> sock_register()

```
sock_register(&ring_family_ops);
static struct net_proto_family ring_family_ops = {
  .family = PF_RING,
  .create = ring_create,
  .owner = THIS_MODULE,
};
```

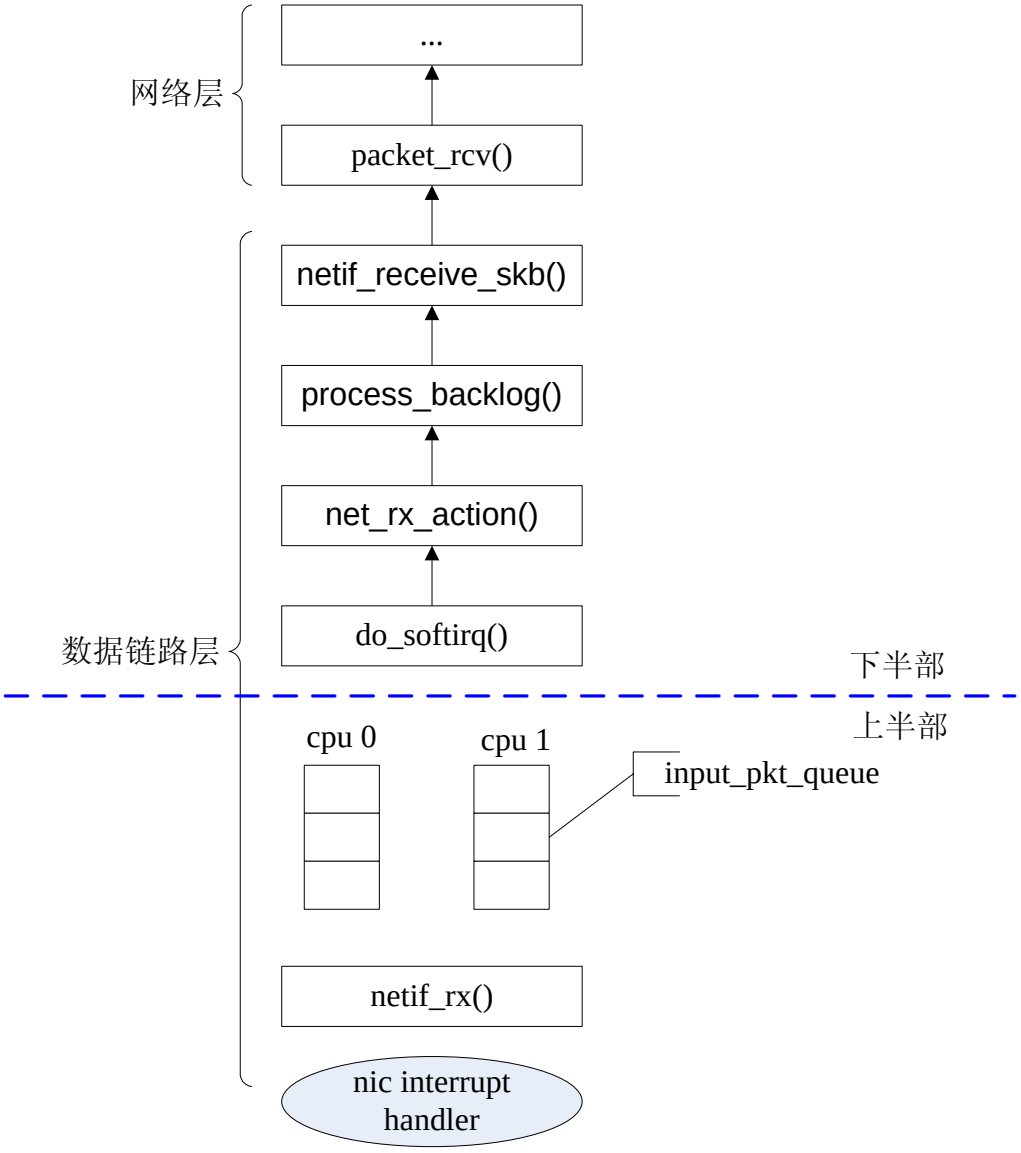
sock_register()注册新的套接字类型 PF_RING

ring_init() ->register_device_handler()

```
void register_device_handler(void) {
  if(transparent_mode != standard_linux_path) return;

  prot_hook.func = packet_rcv;
  prot_hook.type = htons(ETH_P_ALL);
  dev_add_pack(&prot_hook);
}
```

注册一个 hook，当报文从网卡递送到 kernel 时，packet_rcv()被 netif_receive_skb()调用，从中断的角度看，netif_receive_skb()属于下半部，被 do_softirq()调用，调用关系如下：



packet_rcv()调用过程

当应用程序使用 socket(PF_RING，…)创建套接字时，ring_create 被调用(socket()->ring_create()这个过程可参看 net/socket.c)，下面分析 ring_create 过程：

```
static int ring_create(
#if(LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,24))
              struct net *net,#endif
              struct socket *sock, int protocol)
{
  struct sock *sk;
  struct ring_opt *pfr;
  int err;

#if defined(RING_DEBUG)
```

```
  printk("[PF_RING] ring_create()\n");
#endif

  /* Are you root, superuser or so ? */
  if(!capable(CAP_NET_ADMIN))
    return -EPERM;
  /* 只支持 SOCK_RAW 类型，不支持传统的 SOCK_STREAM、SOCK_DGRAM*/
  if(sock->type != SOCK_RAW)
    return -ESOCKTNOSUPPORT;
  /*所有类型的报文 */
  if(protocol != htons(ETH_P_ALL))
    return -EPROTONOSUPPORT;

  err = -ENOMEM;
  /* 分配 struct sock */
  // BD: -- broke this out to keep it more simple and clear as to what the
  // options are.
#if(LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,11))
  sk = sk_alloc(PF_RING, GFP_KERNEL, 1, NULL);
#else
#if(LINUX_VERSION_CODE < KERNEL_VERSION(2,6,24))
    // BD: API changed in 2.6.12, ref:
    // http://svn.clkao.org/svnweb/linux/revision/?rev=28201
    sk = sk_alloc(PF_RING, GFP_ATOMIC, &ring_proto, 1);#else
    sk = sk_alloc(net, PF_INET, GFP_KERNEL, &ring_proto);
#endif
#endif

  if(sk == NULL)
    goto out;
  /*这里才真正将操作函数集（ops）赋上*/
  sock->ops = &ring_ops;
  sock_init_data(sock, sk);
#if(LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,11))
  sk_set_owner(sk, THIS_MODULE);
#endif

  err = -ENOMEM;
  /* 分配 struct ring_opt */
  ring_sk(sk) = ring_sk_datatype(kmalloc(sizeof(*pfr), GFP_KERNEL));

  if(!(pfr = ring_sk(sk))) {
    sk_free(sk);
    goto out;
```

```
  }
  memset(pfr, 0, sizeof(*pfr));
  pfr->ring_active = 0; /* We activate as soon as somebody waits for packets */
  pfr->num_rx_channels = UNKNOWN_NUM_RX_CHANNELS;
  pfr->channel_id = RING_ANY_CHANNEL;
  pfr->bucket_len = DEFAULT_BUCKET_LEN;
  pfr->handle_hash_rule = handle_filtering_hash_bucket;
  init_waitqueue_head(&pfr->ring_slots_waitqueue);
  rwlock_init(&pfr->ring_index_lock);
  rwlock_init(&pfr->ring_rules_lock);
  atomic_set(&pfr->num_ring_users, 0);
  INIT_LIST_HEAD(&pfr->rules);
  sk->sk_family = PF_RING;

  sk->sk_destruct = ring_sock_destruct;
  /* 初始化完成后，将 sk 插入到 ring_table 中 */
  ring_insert(sk);

#if defined(RING_DEBUG)
  printk("[PF_RING] ring_create() - created\n");
#endif

  return(0);
 out:
  return err;
}
```

ring_create()的流程:

1. 分配并初始化 struct sockt 结构
2. 分配并初始化 struct ring_opt 结构
3. 将 struct sock 插入到 ring_table 中

并且，每个 ring 中的环形缓冲区此时并没有被分配，而是需要等到与具体设备 bind 时才分配。

ring_bind()

```
static int ring_bind(struct socket *sock, struct sockaddr *sa, int addr_len)
{
  /* 获取 struct sock* */
  struct sock *sk = sock->sk;
  struct net_device *dev = NULL;

#if defined(RING_DEBUG)
  printk("[PF_RING] ring_bind() called\n");
#endif
```

```
  /*
   *     Check legality
   */
  if(addr_len != sizeof(struct sockaddr))
    return -EINVAL;
  if(sa->sa_family != PF_RING)
    return -EINVAL;
  if(sa->sa_data == NULL)
    return -EINVAL;

  /* Safety check: add trailing zero if missing */
  sa->sa_data[sizeof(sa->sa_data) - 1] = '\0';

#if defined(RING_DEBUG)
  printk("[PF_RING] searching device %s\n", sa->sa_data);
#endif
  /* 通过 device name 获取 struct net_device * */
  if((dev = __dev_get_by_name(
 #if(LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,24))
                  &init_net,
 #endif
                  sa->sa_data)) == NULL) {
 #if defined(RING_DEBUG)
    printk("[PF_RING] search failed\n");
 #endif
    return(-EINVAL);
  } else
    return(packet_ring_bind(sk, dev));
}
```

可见，ring_bind()函数的功能很简单：

1) 检查参数 sa 合法性；
2) 通过 device name(比如 eth0)，查找到对应 struct net_device *；
3) 调用 pakcet_ring_bind()；

这里，简单说明一下 struct sock 和 struct socket 的关系，两者都是描述套接字的结构，且属于一一对应的关系(各自都有成员指向对方),从内容上看，struct sock 面向内核，struct socket 面向进程，具体参考《linux 内核情景分析》第二卷，里面有详细的描述。

packet_ring_bind()

```
static int packet_ring_bind(struct sock *sk, struct net_device *dev)
{
  u_int the_slot_len;
```

```
  u_int32_t tot_mem;
  struct ring_opt *pfr = ring_sk(sk);
  // struct page *page, *page_end;

  if(!dev)
    return(-1);

#if defined(RING_DEBUG)
  printk("[PF_RING] packet_ring_bind(%s) called\n", dev->name);
#endif

  /* ************************************************

   * ************************************
   * *                                  *
   * *       FlowSlotInfo                 *
   * *                                  *
   * ************************************ <-+
   * *       FlowSlot                   *   |
   * ************************************   |
   * *       FlowSlot                   *   |
   * ************************************   +- num_slots
   * *       FlowSlot                   *   |
   * ************************************   |
   * *       FlowSlot                   *   |
   * ************************************ <-+
   *
   * ********************************************** */

  the_slot_len = sizeof(u_char) /* flowSlot.slot_state */
#ifdef RING_MAGIC
    + sizeof(u_char)
#endif
    + sizeof(struct pfring_pkthdr)
    + pfr->bucket_len /* flowSlot.bucket */ ;

  tot_mem = sizeof(FlowSlotInfo) + num_slots * the_slot_len;
  if(tot_mem % PAGE_SIZE)
    tot_mem += PAGE_SIZE - (tot_mem % PAGE_SIZE);

  pfr->ring_memory = rvmalloc(tot_mem);

  if(pfr->ring_memory != NULL) {
    printk("[PF_RING] successfully allocated %lu bytes at 0x%08lx\n",
```

```
      (unsigned long)tot_mem, (unsigned long)pfr->ring_memory);
  } else {
    printk("[PF_RING] ERROR: not enough memory for ring\n");
    return(-1);
  }

  // memset(pfr->ring_memory, 0, tot_mem); // rvmalloc does the memset already

  pfr->slots_info = (FlowSlotInfo *) pfr->ring_memory;
  pfr->ring_slots = (char *)(pfr->ring_memory + sizeof(FlowSlotInfo));

  pfr->slots_info->version = RING_FLOWSLOT_VERSION;
  pfr->slots_info->slot_len = the_slot_len;
  pfr->slots_info->data_len = pfr->bucket_len;
  pfr->slots_info->tot_slots =
    (tot_mem - sizeof(FlowSlotInfo)) / the_slot_len;
  pfr->slots_info->tot_mem = tot_mem;
  pfr->slots_info->sample_rate = 1;

  printk("[PF_RING] allocated %d slots [slot_len=%d][tot_mem=%u]\n",
     pfr->slots_info->tot_slots, pfr->slots_info->slot_len,
     pfr->slots_info->tot_mem);

#ifdef RING_MAGIC
  {
    int i;

    for (i = 0; i < pfr->slots_info->tot_slots; i++) {
      unsigned long idx = i * pfr->slots_info->slot_len;
      FlowSlot *slot = (FlowSlot *) & pfr->ring_slots[idx];
      slot->magic = RING_MAGIC_VALUE;
      slot->slot_state = 0;
    }
  }
#endif

  pfr->sample_rate = 1; /* No sampling */
  pfr->insert_page_id = 1, pfr->insert_slot_id = 0;
  pfr->rules_default_accept_policy = 1, pfr->num_filtering_rules = 0;
  /* 在 proc 目录建立一个 ring 文件, 以 pid+device_name 形式命名*/
  ring_proc_add(ring_sk(sk), dev);

  if(dev->ifindex < MAX_NUM_DEVICES) {
    device_ring_list_element *elem;
```

```
   /* printk("[PF_RING] Adding ring to device index %d\n", dev->ifindex); */

   elem = kmalloc(sizeof(device_ring_list_element), GFP_ATOMIC);
   if(elem != NULL) {
     elem->the_ring = pfr;
     INIT_LIST_HEAD(&elem->list);
     write_lock(&ring_list_lock);
     /* 将该 ring 绑定到该设备上 */
     list_add(&elem->list, &device_ring_list[dev->ifindex]);
     write_unlock(&ring_list_lock);
     /* printk("[PF_RING] Added ring to device index %d\n", dev->ifindex); */
   }
  }

  /*
    IMPORTANT
    Leave this statement here as last one. In fact when
    the ring_netdev != NULL the socket is ready to be used.
  */
  pfr->ring_netdev = dev;

  /*
    As the 'struct net_device' does not contain the number
    of RX queues, we can guess that its number is the same as the number
    of TX queues. After the first packet has been received by the adapter
    the num of RX queues is updated with the real value
   */
#if(LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,31))
  pfr->num_rx_channels = pfr->ring_netdev->real_num_tx_queues;
#else
  pfr->num_rx_channels = 1;
#endif
  return(0);
}
```

在 packet_ring_bind()中，环形缓冲区被分配并初始化，proc 文件被建立，一切就绪，只等标志 ring->ring_active 被激活，这个标识会在 recv()或者 poll()被调用时激活，由于 poll 支持多路 IO，这里我们只分析 ring_poll()，忽略 ring_recvmsg()。

ring_poll()

```
unsigned int ring_poll(struct file *file,
               struct socket *sock, poll_table * wait)
{
  FlowSlot *slot;
```

```
  struct ring_opt *pfr = ring_sk(sock->sk);
  int rc;

  /* printk("[PF_RING] -- poll called\n");  */

  if(pfr->dna_device == NULL) {
    /* PF_RING mode */

#if defined(RING_DEBUG)
    printk("[PF_RING] poll called (non DNA device)\n");
#endif
    /* 激活 ring  */
    pfr->ring_active = 1;
    slot = get_remove_slot(pfr);
    /* 如果当前没有数据可读(slot_state == 0)，将请求插入到等待队列中 */
    if((slot != NULL) && (slot->slot_state == 0)) {
      poll_wait(file, &pfr->ring_slots_waitqueue, wait);
      smp_mb();
    }

#if defined(RING_DEBUG)
    printk("[PF_RING] poll returning %d\n", slot->slot_state);
#endif
    /* 如果 slot 可读，立即返回给应用程序，表示有数据可读，否则返回 0 给 do_poll()，应用程序
会睡眠，直到被中断唤醒或者超时 */
    if((slot != NULL) && (slot->slot_state == 1))
      return(POLLIN | POLLRDNORM);
    else
      return(0);
  } else {
    /* DNA mode 不作分析 */

#if defined(RING_DEBUG)
    printk("[PF_RING] poll called on DNA device [%d]\n",
       *pfr->dna_device->interrupt_received);
#endif

    if(pfr->dna_device->wait_packet_function_ptr == NULL)
      return(0);

    rc = pfr->dna_device->wait_packet_function_ptr(pfr->dna_device->
                           adapter_ptr, 1);
    if(rc == 0) {   /* No packet arrived yet */
      /* poll_wait(file, pfr->dna_device->packet_waitqueue, wait); */
```

```
   } else
     rc = pfr->dna_device->wait_packet_function_ptr(pfr->
                            dna_device->
                            adapter_ptr,
                            0);

   //*pfr->dna_device->interrupt_received = rc;
   if(rc == 0)
     rc = *pfr->dna_device->interrupt_received;

#if defined(RING_DEBUG)
   printk("[PF_RING] poll %s return [%d]\n",
      pfr->ring_netdev->name,
      *pfr->dna_device->interrupt_received);
#endif

   if(rc) {
     return(POLLIN | POLLRDNORM);
   } else {
     return(0);
   }
  }
}
```

ring_poll 的作用就是查询当前是否有 slot 可读，如果有，立即返回，否则，将请求插入到等待队列，如果应用程序调用 poll(fd, flags，timeout)时设置 timeout,在 ring_poll 返回后，应用程序会被睡眠，直到超时或者有包到来。

至此，PF_RING 初始化完成，应用程序也做好了准备,分别执行了 socket()、bind()、poll()，只等待包的到来，下面将分析一个报文从被 PF_RING 接受到存储到环形缓冲区的过程，这个过程即从 packet_rcv()被调用开始。

三、拷贝报文至缓冲区

packet_rcv()

```
static int packet_rcv(struct sk_buff *skb, struct net_device *dev,
            struct packet_type *pt, struct net_device *orig_dev)
{
  int rc;
  /* 过滤环回接口 */
  if(skb->pkt_type != PACKET_LOOPBACK) {
    rc = skb_ring_handler(skb,
            (skb->pkt_type == PACKET_OUTGOING) ? 0 : 1,
```

```
            1, UNKNOWN_RX_CHANNEL, UNKNOWN_NUM_RX_CHANNELS);

  } else
    rc = 0;

  kfree_skb(skb);
  return(rc);
}
```

packet_recv()过滤了环回接口的数据，然后调用了 skb_ring_handler()处理 sk_buff，完成后，对 skb 减少一次引用计数。

packet_rcv() -> skb_ring_handler()

```
static int skb_ring_handler(struct sk_buff *skb,
                u_char recv_packet,
                u_char real_skb /* 1=real skb, 0=faked skb */ ,
                u_int8_t channel_id,
                u_int8_t num_rx_channels)
{
  struct sock *skElement;
  int rc = 0, is_ip_pkt;
  struct list_head *ptr;
  struct pfring_pkthdr hdr;
  int displ;
  struct sk_buff *skk = NULL;
  struct sk_buff *orig_skb = skb;

#ifdef PROFILING
  uint64_t rdt = _rdtsc(), rdt1, rdt2;
#endif

#if defined(RING_DEBUG)
  printk("[PF_RING] --> skb_ring_handler() [channel_id=%d/%d]\n",
     channel_id, num_rx_channels);
#endif
  /* 如果设置了不捕捉 TX 报文，这里会过滤掉从网卡发出的报文 */
  if((!skb)     /* Invalid skb */
     ||((!enable_tx_capture) && (!recv_packet))) {
    /*
      An outgoing packet is about to be sent out
      but we decided not to handle transmitted
      packets.
    */
    return(0);
```

```
  }
#if defined(RING_DEBUG)
  if(1) {
    struct timeval tv;

    skb_get_timestamp(skb, &tv);
    printk
      ("[PF_RING]    skb_ring_handler()
[skb=%p][%u.%u][len=%d][dev=%s][csum=%u]\n",
       skb, (unsigned int)tv.tv_sec, (unsigned int)tv.tv_usec,
       skb->len,
       skb->dev->name == NULL ? "<NULL>" : skb->dev->name,
       skb->csum);
  }
#endif

#if defined (RING_DEBUG)
  /* printk("[PF_RING] channel_id=%d\n", channel_id); */
#endif

#ifdef PROFILING
  rdt1 = _rdtsc();
#endif

  /* 这里的 displ 指链路层报头的长度，由于 packet_rcv 处于网络层，对于接受到得报文，
     skb->data 指向网络层头部,对于发送的报文，skb->data 链路层头部。
  */
  if(recv_packet) {
    /* Hack for identifying a packet received by the e1000 */
    if(real_skb)
      displ = SKB_DISPLACEMENT;
    else
      displ = 0; /* Received by the e1000 wrapper */
  } else
    displ = 0;

  /* 分析 sk_buff，并将信息存入到 pfring_pkthdr 中 */
  is_ip_pkt = parse_pkt(skb, displ, &hdr);

  /* ip defrag 略*/
  /* (de)Fragmentation <fusco@ntop.org> */
  if(enable_ip_defrag
     && real_skb && is_ip_pkt && recv_packet && (ring_table_size > 0)) {
    struct sk_buff *cloned = NULL;
```

```
   struct iphdr *iphdr = NULL;

   skb_reset_network_header(skb);
   skb_reset_transport_header(skb);
   skb_set_network_header(skb, ETH_HLEN - displ);

   iphdr = ip_hdr(skb);

   if(iphdr) {
#if defined (RING_DEBUG)
     printk("[PF_RING] [version=%d] %X -> %X\n",
        iphdr->version, iphdr->saddr, iphdr->daddr);
#endif
     if(iphdr->frag_off & htons(IP_MF | IP_OFFSET)) {
    if((cloned = skb_clone(skb, GFP_ATOMIC)) != NULL) {
#if defined (RING_DEBUG)
      int offset = ntohs(iphdr->frag_off);
      offset &= IP_OFFSET;
      offset <<= 3;

      printk
        ("[PF_RING] There is a fragment to handle [proto=%d][frag_off=%u]"
         "[ip_id=%u][network_header=%d][displ=%d]\n",
         iphdr->protocol, offset,
         ntohs(iphdr->id),
         hdr.parsed_pkt.pkt_detail.offset.
         l3_offset - displ, displ);
#endif
      skk = ring_gather_frags(cloned);

      if(skk != NULL) {
#if defined (RING_DEBUG)
        printk
          ("[PF_RING] IP reasm on new skb [skb_len=%d]"
           "[head_len=%d][nr_frags=%d][frag_list=%p]\n",
           (int)skk->len,
           skb_headlen(skk),
           skb_shinfo(skk)->nr_frags,
           skb_shinfo(skk)->
           frag_list);
#endif
        skb = skk;
        parse_pkt(skb, displ, &hdr);
        hdr.len = hdr.caplen = skb->len + displ;
```

```
     } else {
       //printk("[PF_RING] Fragment queued \n");
       return(0);  /* mask rcvd fragments */
     }
    }
     } else {
#if defined (RING_DEBUG)
    printk("[PF_RING] Do not seems to be a fragmented ip_pkt[iphdr=%p]\n",
          iphdr);
#endif
     }
   }
  }

  /* BD - API changed for time keeping */
#if(LINUX_VERSION_CODE < KERNEL_VERSION(2,6,14))
  if(skb->stamp.tv_sec == 0)
    do_gettimeofday(&skb->stamp);
  hdr.ts.tv_sec = skb->stamp.tv_sec, hdr.ts.tv_usec = skb->stamp.tv_usec;
#elif(LINUX_VERSION_CODE < KERNEL_VERSION(2,6,22))
  if(skb->tstamp.off_sec == 0)
    __net_timestamp(skb);
  hdr.ts.tv_sec = skb->tstamp.off_sec, hdr.ts.tv_usec =
    skb->tstamp.off_usec;
#else /* 2.6.22 and above */
  if(skb->tstamp.tv64 == 0)
    __net_timestamp(skb);
  hdr.ts = ktime_to_timeval(skb->tstamp);
#endif

  hdr.len = hdr.caplen = skb->len + displ;

  /* Avoid the ring to be manipulated while playing with it */
  read_lock_bh(&ring_mgmt_lock);

  /* 遍历所有没有被集群的sock(ring)，如果条件符合则拷贝报文至缓冲区 */
  /* [1] Check unclustered sockets */
  list_for_each(ptr, &ring_table) {
    struct ring_opt *pfr;
    struct ring_element *entry;

    entry = list_entry(ptr, struct ring_element, list);

    skElement = entry->sk;
```

```
   pfr = ring_sk(skElement);

   /* 需要满足的条件:
    * 1. ring 存在且没有被集群
    * 2. ring 的缓冲区存在，且方向符合
    * 3. 报文的 device 或者其 master 符合 ring 设定的 device
    */
   if((pfr != NULL)
    && (pfr->cluster_id == 0 /* No cluster */ )
    && (pfr->ring_slots != NULL)
    && is_valid_skb_direction(pfr->direction, recv_packet)
    && ((pfr->ring_netdev == skb->dev)
        || ((skb->dev->flags & IFF_SLAVE)
        && (pfr->ring_netdev == skb->dev->master)))) {
     /* We've found the ring where the packet can be stored */
     int old_caplen = hdr.caplen;  /* Keep old lenght */
     /* 如果 bucket_len 设定过小，会被截断 */
     hdr.caplen = min(hdr.caplen, pfr->bucket_len);
     /* 将报文插入至该环形缓冲区中 */
     add_skb_to_ring(skb,   pfr,   &hdr,   is_ip_pkt,   displ,   channel_id,
num_rx_channels);
     hdr.caplen = old_caplen;
     rc = 1;    /* Ring found: we've done our job */
   }
  }
  /* 处理被集群的 ring,在 PF_RING 看来，被集群的数个 ring 就是一个 ring,报文只会被递送到
   * 其中的一个 ring 上，至于散列到哪一个 ring，由 hash 算法决定，这里，PF_RING 提供了一种
   * 负载均衡的机制。
   */
  /* [2] Check socket clusters */
  list_for_each(ptr, &ring_cluster_list) {
   ring_cluster_element *cluster_ptr;
   struct ring_opt *pfr;

   cluster_ptr = list_entry(ptr, ring_cluster_element, list);

   if(cluster_ptr->cluster.num_cluster_elements > 0) {
     u_int skb_hash = hash_pkt_cluster(cluster_ptr, &hdr);

     skElement = cluster_ptr->cluster.sk[skb_hash];

     if(skElement != NULL) {
    pfr = ring_sk(skElement);
```

```
    if((pfr != NULL)
       && (pfr->ring_slots != NULL)
       && ((pfr->ring_netdev == skb->dev)
        || ((skb->dev->flags & IFF_SLAVE)
           && (pfr->ring_netdev ==
            skb->dev->master)))
       && is_valid_skb_direction(pfr->direction, recv_packet)
       ) {
      /* We've found the ring where the packet can be stored */
      add_skb_to_ring(skb, pfr, &hdr,
              is_ip_pkt, displ,
              channel_id, num_rx_channels);
      rc = 1;   /* Ring found: we've done our job */
    }
     }
   }
  }

  read_unlock_bh(&ring_mgmt_lock);

#ifdef PROFILING
  rdt1 = _rdtsc() - rdt1;
#endif

#ifdef PROFILING
  rdt2 = _rdtsc();
#endif

  /* Fragment handling */
  if(skk != NULL)
    kfree_skb(skk);

  if(rc == 1) {
    if(transparent_mode != driver2pf_ring_non_transparent) {
      rc = 0;
    } else {
      if(recv_packet && real_skb) {
#if defined(RING_DEBUG)
    printk("[PF_RING] kfree_skb()\n");
#endif

    kfree_skb(orig_skb);
     }
    }
```

```
  }
#ifdef PROFILING
  rdt2 = _rdtsc() - rdt2;
  rdt = _rdtsc() - rdt;

#if defined(RING_DEBUG)
  printk
    ("[PF_RING] # cycles: %d [lock costed %d %d%%][free costed %d %d%%]\n",
     (int)rdt, rdt - rdt1,
     (int)((float)((rdt - rdt1) * 100) / (float)rdt), rdt2,
     (int)((float)(rdt2 * 100) / (float)rdt));
#endif
#endif

  //printk("[PF_RING] Returned %d\n", rc);
  return(rc);       /*  0 = packet not handled */
}
```

首先过滤 TX 方向的报文（如果配置不捕获发送报文的话），并根据方向判断 displ 的取值，并调用 parse_pkt()分析报文，填充 pfring_pkthdr。

packet_rcv() -> skb_ring_handler()->parse_pkt()

```
static int parse_pkt(struct sk_buff *skb,
             u_int16_t skb_displ, struct pfring_pkthdr *hdr)
{
  struct iphdr *ip;
  /* 获取链路层头部 */
  struct ethhdr *eh = (struct ethhdr *)(skb->data - skb_displ);
  u_int16_t displ;

  memset(&hdr->parsed_pkt, 0, sizeof(struct pkt_parsing_info));
  hdr->parsed_header_len = 9;

  /* MAC adderess */
  memcpy(&hdr->parsed_pkt.dmac, eh->h_dest, sizeof(eh->h_dest));
  memcpy(&hdr->parsed_pkt.smac, eh->h_source, sizeof(eh->h_source));

  hdr->parsed_pkt.eth_type = ntohs(eh->h_proto);
  hdr->parsed_pkt.pkt_detail.offset.eth_offset = -skb_displ;

  /* 处理 vlan 报文*/
  if(hdr->parsed_pkt.eth_type == 0x8100 /* 802.1q (VLAN) */ ) {
    hdr->parsed_pkt.pkt_detail.offset.vlan_offset =
      hdr->parsed_pkt.pkt_detail.offset.eth_offset + sizeof(struct ethhdr);
    hdr->parsed_pkt.vlan_id =
```

```
    (skb->data[hdr->parsed_pkt.pkt_detail.offset.vlan_offset] & 15) * 256 +
    skb->data[hdr->parsed_pkt.pkt_detail.offset.vlan_offset + 1];
  hdr->parsed_pkt.eth_type =
    (skb->data[hdr->parsed_pkt.pkt_detail.offset.vlan_offset + 2]) * 256 +
    skb->data[hdr->parsed_pkt.pkt_detail.offset.vlan_offset + 3];
  displ = 4;
} else {
  displ = 0;
  hdr->parsed_pkt.vlan_id = 0;    /* Any VLAN */
}
/* l3 报文 */
if(hdr->parsed_pkt.eth_type == 0x0800 /* IP */ ) {
  hdr->parsed_pkt.pkt_detail.offset.l3_offset =
    hdr->parsed_pkt.pkt_detail.offset.eth_offset + displ +
    sizeof(struct ethhdr);
  ip = (struct iphdr *)(skb->data +
             hdr->parsed_pkt.pkt_detail.offset.
             l3_offset);

  hdr->parsed_pkt.ipv4_src =
    ntohl(ip->saddr), hdr->parsed_pkt.ipv4_dst =
    ntohl(ip->daddr), hdr->parsed_pkt.l3_proto = ip->protocol;
  hdr->parsed_pkt.ipv4_tos = ip->tos;
  hdr->parsed_pkt.pkt_detail.offset.l4_offset =
    hdr->parsed_pkt.pkt_detail.offset.l3_offset + ip->ihl * 4;

/* l4 报文 */
  if((ip->protocol == IPPROTO_TCP)
   || (ip->protocol == IPPROTO_UDP)) {
    if(ip->protocol == IPPROTO_TCP) {
   struct tcphdr *tcp =
     (struct tcphdr *)(skb->data +
              hdr->parsed_pkt.
              pkt_detail.offset.
              l4_offset);
   hdr->parsed_pkt.l4_src_port =
     ntohs(tcp->source),
     hdr->parsed_pkt.l4_dst_port =
     ntohs(tcp->dest);
   hdr->parsed_pkt.pkt_detail.offset.
     payload_offset =
     hdr->parsed_pkt.pkt_detail.offset.
     l4_offset + (tcp->doff * 4);
   hdr->parsed_pkt.tcp_flags =
```

```
     (tcp->fin * TH_FIN_MULTIPLIER) +
     (tcp->syn * TH_SYN_MULTIPLIER) +
     (tcp->rst * TH_RST_MULTIPLIER) +
     (tcp->psh * TH_PUSH_MULTIPLIER) +
     (tcp->ack * TH_ACK_MULTIPLIER) +
     (tcp->urg * TH_URG_MULTIPLIER);
    } else if(ip->protocol == IPPROTO_UDP) {
   struct udphdr *udp =
     (struct udphdr *)(skb->data +
              hdr->parsed_pkt.
              pkt_detail.offset.
              l4_offset);
   hdr->parsed_pkt.l4_src_port =
     ntohs(udp->source),
     hdr->parsed_pkt.l4_dst_port =
     ntohs(udp->dest);
   hdr->parsed_pkt.pkt_detail.offset.
     payload_offset =
     hdr->parsed_pkt.pkt_detail.offset.
     l4_offset + sizeof(struct udphdr);
    } else
   hdr->parsed_pkt.pkt_detail.offset.
     payload_offset =
     hdr->parsed_pkt.pkt_detail.offset.l4_offset;
   } else
     hdr->parsed_pkt.l4_src_port =
   hdr->parsed_pkt.l4_dst_port = 0;

   hdr->parsed_pkt.pkt_detail.offset.eth_offset = skb_displ;

   return(1);  /* IP */
 }
 /* TODO: handle IPv6 */
 return(0);       /* No IP */
}
```

该函数提取 l2-l4 层的信息并存储进 pfring_pkthdr, 值得一提的是，支持对 vlan 协议的 支持。

回到 skb_ring_handler()中，接下来提供了 ip 分片重组功能，一般都在用户态 e 完成该功能，此处略去分析，而后 是 skb_ring_handler()的核心功能，遍历查找合适的 ring，并将 sk_buff 插入其中。

packet_rcv() -> skb_ring_handler()

```
 read_lock_bh(&ring_mgmt_lock);
```

```
  /* [1] Check unclustered sockets */
  list_for_each(ptr, &ring_table) {
    struct ring_opt *pfr;
    struct ring_element *entry;

    entry = list_entry(ptr, struct ring_element, list);

    skElement = entry->sk;
    pfr = ring_sk(skElement);

    if((pfr != NULL)
     && (pfr->cluster_id == 0 /* No cluster */ )
     && (pfr->ring_slots != NULL)
     && is_valid_skb_direction(pfr->direction, recv_packet)
     && ((pfr->ring_netdev == skb->dev)
         || ((skb->dev->flags & IFF_SLAVE)
         && (pfr->ring_netdev == skb->dev->master)))) {
      /* We've found the ring where the packet can be stored */
      int old_caplen = hdr.caplen;  /* Keep old lenght */
      hdr.caplen = min(hdr.caplen, pfr->bucket_len);
      add_skb_to_ring(skb,   pfr,   &hdr,   is_ip_pkt,   displ,   channel_id,
num_rx_channels);
      hdr.caplen = old_caplen;
      rc = 1;    /* Ring found: we've done our job */
    }
  }
```

遍历ring_table，由于每个元素是 struct ring_element ,所以通过sock找到ptr(struct ring_opt *)，并作一系列过滤，最重要的是cluster_id必须为0,且网卡 设备必须匹配，匹配成功后调用add_skb_to_ring()插入该ring。

packet_rcv() -> skb_ring_handler() -> add_skb_to_ring()
该函数主要进行基于 BPF 的过滤以及 plugin 的调用，这里略去不作分析，如果该 skb 需要被传递给应用程序，则调用 add_pkt_to_ring()将 skb 插入到 ring 中。

packet_rcv() -> skb_ring_handler() -> add_skb_to_ring() -> add_pkt_to_ring()

```
static void add_pkt_to_ring(struct sk_buff *skb,
               struct ring_opt *pfr,
               struct pfring_pkthdr *hdr,
               int displ, u_int8_t channel_id,
               int offset, void *plugin_mem)
{
  char *ring_bucket;
  int idx;
```

```
  FlowSlot *theSlot;
  int32_t the_bit = 1 << channel_id;

#if defined(RING_DEBUG)
  printk("[PF_RING]                -->                add_pkt_to_ring(len=%d)
[pfr->channel_id=%d][channel_id=%d]\n",
     hdr->len, pfr->channel_id, channel_id);
#endif

  if(!pfr->ring_active)
    return;

  if((pfr->channel_id != RING_ANY_CHANNEL)
     && (channel_id != RING_ANY_CHANNEL)
     && ((pfr->channel_id & the_bit) != the_bit))
    return; /* Wrong channel */

  write_lock_bh(&pfr->ring_index_lock);
  /* 获取当前可以插入的 slot */
  idx = pfr->slots_info->insert_idx;
  idx++, theSlot = get_insert_slot(pfr);
  pfr->slots_info->tot_pkts++;

  /* 无 slot 或该 slot 还未被应用程序取走，丢弃 */
  if((theSlot == NULL) || (theSlot->slot_state != 0)) {
    /* No room left */
    pfr->slots_info->tot_lost++;
    write_unlock_bh(&pfr->ring_index_lock);
    return;
  }

  ring_bucket = &theSlot->bucket;

  if((plugin_mem != NULL) && (offset > 0))
    memcpy(&ring_bucket[sizeof(struct pfring_pkthdr)], plugin_mem, offset);

  if(skb != NULL) {
    hdr->caplen = min(pfr->bucket_len - offset, hdr->caplen);

    if(hdr->caplen > 0) {
#if defined(RING_DEBUG)
     printk("[PF_RING]                                                    -->
[caplen=%d][len=%d][displ=%d][parsed_header_len=%d][bucket_len=%d][sizeof=%d
]\n",
```

```
     hdr->caplen, hdr->len, displ,
        hdr->parsed_header_len, pfr->bucket_len,
        sizeof(struct pfring_pkthdr));
#endif
    // 将数据从 skb 拷贝到 ring_bucket[offset]处
     skb_copy_bits(skb, -displ,
           &ring_bucket[sizeof(struct pfring_pkthdr) + offset], hdr->caplen);
   } else {
     if(hdr->parsed_header_len >= pfr->bucket_len) {
    static u_char print_once = 0;

    if(!print_once) {
      printk("[PF_RING] WARNING: the bucket len is [%d] shorter than the plugin
parsed header [%d]\n",
        pfr->bucket_len, hdr->parsed_header_len);
      print_once = 1;
    }
     }
   }
  }
    // 拷贝数据包头部信息至每个 bucket 头部
  memcpy(ring_bucket, hdr, sizeof(struct pfring_pkthdr)); /* Copy extended packet
header */

  if(idx == pfr->slots_info->tot_slots)
    pfr->slots_info->insert_idx = 0; // 双向循环队列重新指向数组的头部
  else
    pfr->slots_info->insert_idx = idx;

#if defined(RING_DEBUG)
  printk("[PF_RING] ==> insert_idx=%d\n", pfr->slots_info->insert_idx);
#endif

  pfr->slots_info->tot_insert++;
  theSlot->slot_state = 1; // 表示报文已插入至 ring 中
  write_unlock_bh(&pfr->ring_index_lock);

  /* wakeup in case of poll() */
  if(waitqueue_active(&pfr->ring_slots_waitqueue))
    wake_up_interruptible(&pfr->ring_slots_waitqueue);
}
```

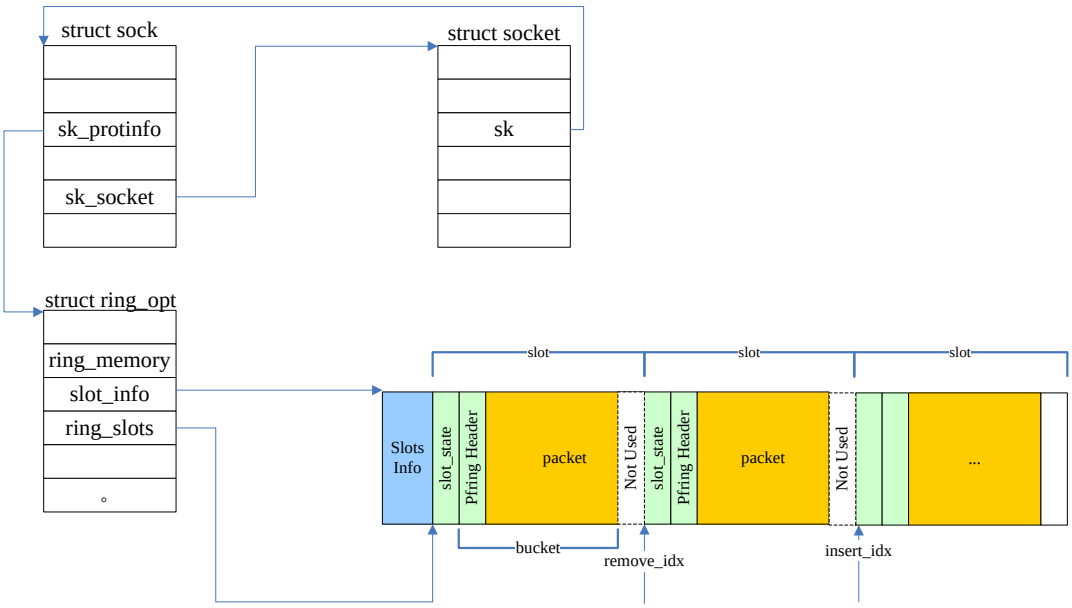
add_pkt_to_ring() 完成具体包拷贝，将报文以及分析出的头部信息插入到一个空闲的 slot 中，并唤醒调用 poll()睡眠的用户程序。

这里，对 write_lock_bh()的理解不是很清楚，为防止 ring 被删除(rmmod)，加锁

可以理解的，但是为什么需要 disable 下半部呢？

具体的插入的插入过程不再分析，仅一张结构图描述各个结构的内存布局：

sock socket ring slot bucket 关系图



回到 skb_ring_handler()中，目前只处理了 ring_table 中没有被 cluster 的 ring，接下来，处理已被 cluster 的 ring。

packet_rcv() -> skb_ring_handler()

```
/* [2] Check socket clusters */
  list_for_each(ptr, &ring_cluster_list) {
    ring_cluster_element *cluster_ptr;
    struct ring_opt *pfr;

    cluster_ptr = list_entry(ptr, ring_cluster_element, list);

    if(cluster_ptr->cluster.num_cluster_elements > 0) {
      u_int skb_hash = hash_pkt_cluster(cluster_ptr, &hdr);

      skElement = cluster_ptr->cluster.sk[skb_hash];

      if(skElement != NULL) {
     pfr = ring_sk(skElement);

     if((pfr != NULL)
```

```
      && (pfr->ring_slots != NULL)
      && ((pfr->ring_netdev == skb->dev)
       || ((skb->dev->flags & IFF_SLAVE)
           && (pfr->ring_netdev ==
            skb->dev->master)))
      && is_valid_skb_direction(pfr->direction, recv_packet)
      ) {
     /* We've found the ring where the packet can be stored */
     add_skb_to_ring(skb, pfr, &hdr,
            is_ip_pkt, displ,
            channel_id, num_rx_channels);
     rc = 1;   /* Ring found: we've done our job */
   }
    }
  }
 }
```

处理 ring_cluster_list，与 ring_table 大致相同，唯一的区别是，对于每个 cluster, 只选出 cluster 其中的一个 ring，而非 cluster 的 ring，符合条件都需要插入，现有的 PF_RING 提供了两种 hash 算法:
cluster_round_robin：轮转，针对报文，同一条连接报文会被散列到不同的 ring 中，这是一种最细粒度的分派，但在某些需要维护连接的应用中不适用。
per_flow：基于流的分派，hash 的元素包括：vlan_id、protocol、传输层 4 元组，可以根据应用场景改写 hash 算法。
接下来也调用了 add_skb_to_ring()将 skb 插入到 ring 中，过程与非 cluster 一致。

至此，skb 已被插入到 ring 中，pakcet_rcv()的过程全部结束。

四、模块退出

模块退出时，过程基本与模块初始化时相反。

```
static void __exit ring_exit(void)
{
  struct list_head *ptr, *tmp_ptr;
  struct ring_element *entry;
  struct pfring_hooks *hook;

  pfring_enabled = 0;
 /* 注销网络层协议，不再接收报文 */
  unregister_device_handler();
  /* 清理 ring_table*/
  list_for_each_safe(ptr, tmp_ptr, &ring_table) {
    entry = list_entry(ptr, struct ring_element, list);
```

```
    list_del(ptr);
    kfree(entry);
  }
  /* 清理 ring_aware_device_list */
  list_for_each_safe(ptr, tmp_ptr, &ring_aware_device_list) {
    ring_device_element *dev_ptr;

    dev_ptr = list_entry(ptr, ring_device_element, list);
    hook = (struct pfring_hooks*)dev_ptr->dev->pfring_ptr;
    if(hook->magic == PF_RING) {
      printk("[PF_RING] Unregister hook for %s\n", dev_ptr->dev->name);
      dev_ptr->dev->pfring_ptr = NULL; /* Unhook PF_RING */
    }
    list_del(ptr);
    kfree(dev_ptr);
  }
  /* 清理 ring_cluster_list */
  list_for_each_safe(ptr, tmp_ptr, &ring_cluster_list) {
    ring_cluster_element *cluster_ptr;

    cluster_ptr = list_entry(ptr, ring_cluster_element, list);

    list_del(ptr);
    kfree(cluster_ptr);
  }
  /* 清理 ring_dna_devices_list */
  list_for_each_safe(ptr, tmp_ptr, &ring_dna_devices_list) {
    dna_device_list *elem;

    elem = list_entry(ptr, dna_device_list, list);

    list_del(ptr);
    kfree(elem);
  }
  /* 注销 socket PF_RING*/
  sock_unregister(PF_RING);
  proto_unregister(&ring_proto);
  unregister_netdevice_notifier(&ring_netdev_notifier);
  /* 删除相应的 proc 目录*/
  ring_proc_term();

  printk("[PF_RING] unloaded\n");
}
```

五、测试

笔者使用标准的 libpcap-mmap、libpcap-ring、ring 进行的了测试，
硬件环境：
cpu ：Intel(R) Xeon(R) CPU E5430 @ 2.66GHz
mem：4G
nic: intel 千兆
压力仪：smartbit
首先测试了一组单网卡的数据，发现 3 种情况下，都达到了网卡的阀值。
再测试了一组双网卡组成 bridge 的数据，发现在小包情况下，libpcap-mmap 和 libpcap-ring 大致相当，但 ring 的抓包效果明显较差。
最后配置了网卡的 NAPI 模式，发现软中断全部被分配到其中一个核上，该核的负荷一直是 100%，抓包效果也奇差无比。

因此，polling 技术在 smp 机器上的运用首先需要解决软中断分配不均衡的问题，否则性能反而更差，貌似网上有人提出过这个问题，但没有给出相应解决思路。
另外 ring 在小包的情况下性能没有达到预期这个问题，目前原因不明。
欣喜的是，PF_RING 相比 libpcap 有自己的优点：

1. 抓包点可以调整;
2. 基于每个网卡分配缓冲，利用 poll()的多路 io 的特性，完美支持多网卡抓包;
3. 有支持多进程的机制(cluster),有相应的负载均衡机制;
4. 实现简洁，代码量比 libpcap 小的多，易于修改和维护;