图解 Linux 网络包发送过程

开发内功修炼之网络篇

张彦飞

最后更新时间：2021-05

公众号：「开发内功修炼」

了解你的每一比特、用好你的每一纳秒！

公众号

作者微信

大家好，我是飞哥!

半年前我以源码的方式描述了网络包的接收过程。之后不断有粉丝提醒我，飞哥飞哥，你还没聊发送过程呢。好，安排!

在开始今天的文章之前，我先来请大家思考几个小问题。

- 问1：我们在查看内核发送数据消耗的 CPU 时，是应该看 sy 还是 si ?
- 问2：为什么你服务器上的 /proc/softirqs 里 NET_RX 要比 NET_TX 大的多的多?
- 问3：发送网络数据的时候都涉及到哪些内存拷贝操作?

这些问题虽然在线上经常看到，但我们似乎很少去深究。如果真的能透彻地把这些问题理解到位，我们对性能的掌控能力将会变得更强。

带着这三个问题，我们开始今天对 Linux 内核网络发送过程的深度剖析。还是按照我们之前的传统，先从一段简单的代码作为切入。如下代码是一个典型服务器程序的典型的缩微代码：

```
int main(){
  fd = socket(AF_INET, SOCK_STREAM, 0);
  bind(fd, ...);
  listen(fd, ...);

  cfd = accept(fd, ...);

  // 接收用户请求
  read(cfd, ...);

  // 用户请求处理
  dosometing();

  // 给用户返回结果
  send(cfd, buf, sizeof(buf), 0);
}
```

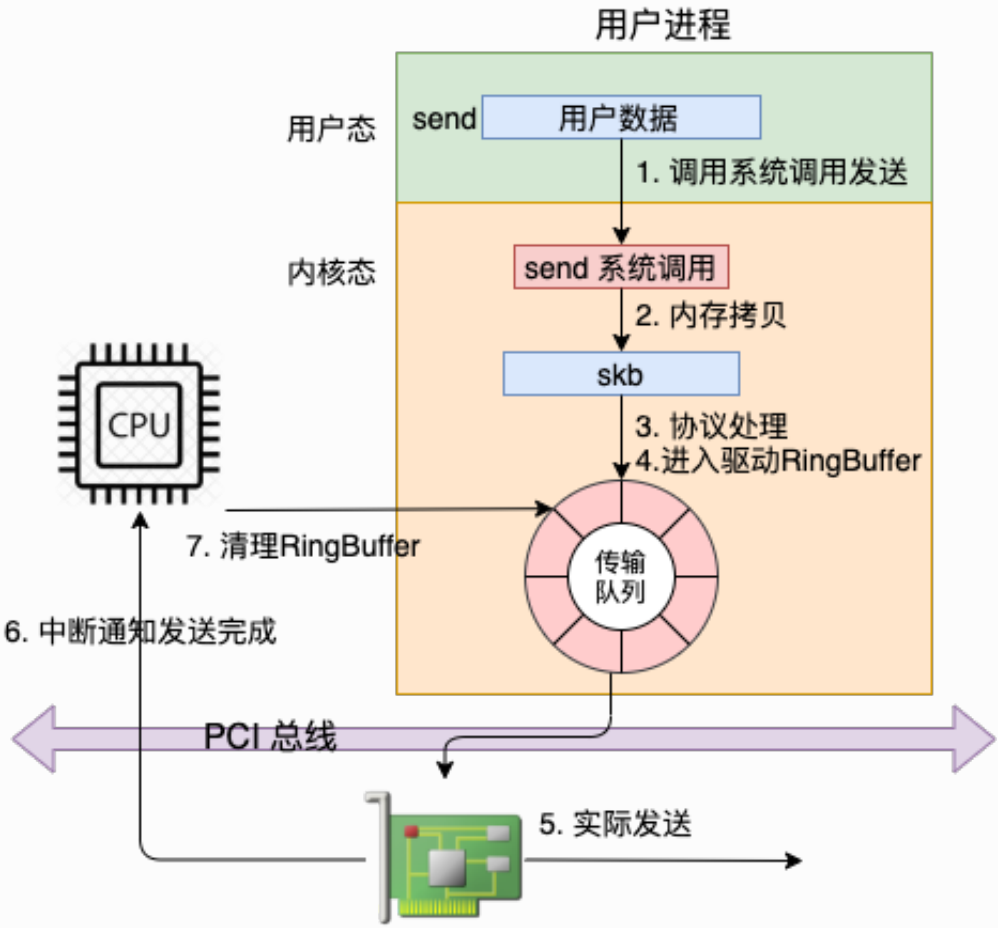
今天我们来讨论上述代码中，调用 send 之后内核是怎么样把数据包发送出去的。本文基于Linux 3.10，网卡驱动采用Intel的igb网卡举例。

预警：本文共有两万六千字，20 多张图，长文慎入!

一、LINUX 网络发送过程总览

飞哥觉得看 Linux 源码最重要的是得有整体上的把握，而不是一开始就陷入各种细节。

我这里先给大家准备了一个总的流程图，简单阐述下 send 发送了的数据是如何一步一步被发送到网卡的。



在这幅图中，我们看到用户数据被拷贝到内核态，然后经过协议栈处理后进入到了 RingBuffer 中。随后网卡驱动真正将数据发送了出去。当发送完成的时候，是通过硬中断来通知 CPU，然后清理 RingBuffer。

因为文章后面要进入源码，所以我们再从源码的角度给出一个流程图。

应用层

```c
int main(){
    // 给用户返回结果
    send(cfd, buf, sizeof(buf), 0);
}
```

系统调用

```c
//file: net/socket.c
SYSCALL_DEFINE6(sendto, int, fd, ...)
{
    //构造 msghdr 并赋值
    struct msghdr msg;
    ......

    //发送数据
    sock_sendmsg(sock, &msg, len);
}
```

```
//file: net/socket.c
static inline int __sock_sendmsg_nosec(...)
{
    return sock->ops->sendmsg(iocb, sock, msg, size);
}
```

协议栈

```
//file: net/ipv4/af_inet.c
int inet_sendmsg(......)
{
    return sk->sk_prot->sendmsg(iocb, sk, msg, size);
}
```

传输层

```
//file: net/ipv4/tcp.c
int tcp_sendmsg(...)
{
    ...
}

//file: net/ipv4/tcp_output.c
static int tcp_transmit_skb(......)
{
    //封装TCP头
    th = tcp_hdr(skb);
    th->source      = inet->inet_sport;
    th->dest        = inet->inet_dport;

    //调用网络层发送接口
    err = icsk->icsk_af_ops->queue_xmit(skb);
}
```

网络层

```
//file: net/ipv4/ip_output.c
int ip_queue_xmit(struct sk_buff *skb, struct flowi *fl)
{
    res = ip_local_out(skb);
}

//file: net/ipv4/ip_output.c
static inline int ip_finish_output2(struct sk_buff *skb)
{
    //继续向下层传递
    int res = dst_neigh_output(dst, neigh, skb);
}
```

链路

```
//file: include/net/dst.h
static inline int dst_neigh_output(...)
{
    ......
    return neigh_hh_output(hh, skb);
}
```

层

```
//file: include/net/neighbour.h
static inline int neigh_hh_output(...)
{
    ......
    skb_push(skb, hh_len);
    return dev_queue_xmit(skb);
}
```

网络设备子系统

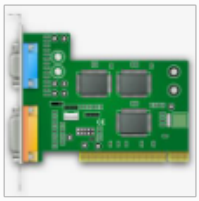
```
//file: net/core/dev.c
int dev_queue_xmit(struct sk_buff *skb)
{
    //选择发送队列并获取 qdisc
    txq = netdev_pick_tx(dev, skb);
    q = rcu_dereference_bh(txq->qdisc);

    //则调用__dev_xmit_skb 继续发送
    rc = __dev_xmit_skb(skb, q, dev, txq);
}
```

```
//file: net/core/dev.c
int dev_hard_start_xmit(...)
{
    //获取设备的回调函数集合 ops
    const struct net_device_ops *ops = dev->netdev_ops;

    //调用驱动里的发送回调函数 ndo_start_xmit 将数据包传给网卡设备
    skb_len = skb->len;
    rc = ops->ndo_start_xmit(skb, dev);
}
```

驱动程序

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static netdev_tx_t igb_xmit_frame(...)
{
    return igb_xmit_frame_ring(skb, ...);
}

netdev_tx_t igb_xmit_frame_ring(...)
{
    //获取TX Queue 中下一个可用缓冲区信息
    first = &tx_ring->tx_buffer_info[tx_ring->next_to_use];
    first->skb = skb;
    first->bytecount = skb->len;

    //igb_tx_map 函数准备给设备发送的数据。
    igb_tx_map(tx_ring, first, hdr_len);
}
```

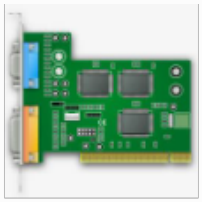
硬件



虽然数据这时已经发送完毕，但是其实还有一件重要的事情没有做，那就是释放缓存队列等内存。

那内核是如何知道什么时候才能释放内存的呢，当然是等网络发送完毕之后。网卡在发送完毕的时候，会给 CPU 发送一个硬中断来通知 CPU。更完整的流程看图：

硬件



硬中断

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static irqreturn_t igb_msix_ring(int irq, void *data)
{
    napi_schedule(&q_vector->napi);
}

static inline void ____napi_schedule(...)
{
    list_add_tail(&napi->poll_list, &sd->poll_list);
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

软中断

```
static void net_rx_action(struct softirq_action *h)
{
    while (!list_empty(&sd->poll_list)) {
        ......
        n = list_first_entry(&sd->poll_list, ...);
        work = n->poll(n, weight);
    }
}
```

驱动

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static int igb_poll(struct napi_struct *napi, int budget)
{
    //performs the transmit completion operations
    if (q_vector->tx.ring)
        clean_complete = igb_clean_tx_irq(q_vector);
    ...
}
```

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static bool igb_clean_tx_irq(struct igb_q_vector *q_vector)
{
    //释放 skb
    dev_kfree_skb_any(tx_buffer->skb);

    //清除 tx_buffer
    tx_buffer->skb = NULL;
    dma_unmap_len_set(tx_buffer, len, 0);

    //清理 DMA 区域
    while (tx_desc != eop_desc) {
        ...
    }
}
```

结束

注意，我们今天的主题虽然是发送数据，但是硬中断最终触发的软中断却是 NET_RX_SOFTIRQ，而并不是 NET_TX_SOFTIRQ！！！（T 是 transmit 的缩写，R 表示 receive）

意不意外，惊不惊喜？？？

所以这就是开篇问题 1 的一部分的原因（注意，这只是一部分原因）。

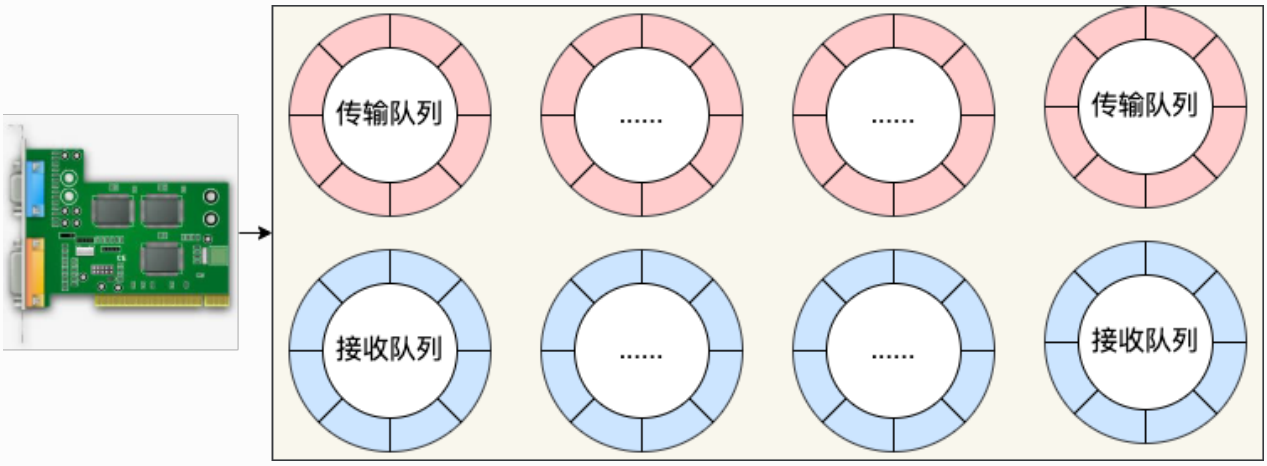
> 问1：在服务器上查看 /proc/softirqs，为什么 NET_RX 要比 NET_TX 大的多的多?

传输完成最终会触发 NET_RX，而不是 NET_TX。 所以自然你观测 /proc/softirqs 也就能看到 NET_RX 更多了。

好，现在你已经对内核是怎么发送网络包的有一个全局上的把握了。不要得意，我们需要了解的细节才是更有价值的地方，让我们继续！！

二、网卡启动准备

现在的服务器上的网卡一般都是支持多队列的。每一个队列上都是由一个 RingBuffer 表示的，开启了多队列以后的的网卡就会对应有多个 RingBuffer。



网卡在启动时最重要的任务之一就是分配和初始化 RingBuffer，理解了 RingBuffer 将会非常有助于后面我们掌握发送。因为今天的主题是发送，所以就以传输队列为例，我们来看下网卡启动时分配 RingBuffer 的实际过程。

在网卡启动的时候，会调用到 __igb_open 函数，RingBuffer 就是在这里分配的。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static int __igb_open(struct net_device *netdev, bool resuming)
{
  struct igb_adapter *adapter = netdev_priv(netdev);

  //分配传输描述符数组
  err = igb_setup_all_tx_resources(adapter);
```

```
  //分配接收描述符数组
  err = igb_setup_all_rx_resources(adapter);

  //开启全部队列
  netif_tx_start_all_queues(netdev);
}
```

在上面 __igb_open 函数调用 igb_setup_all_tx_resources 分配所有的传输 RingBuffer，调用 igb_setup_all_rx_resources 创建所有的接收 RingBuffer。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static int igb_setup_all_tx_resources(struct igb_adapter *adapter)
{
  //有几个队列就构造几个 RingBuffer
  for (i = 0; i < adapter->num_tx_queues; i++) {
    igb_setup_tx_resources(adapter->tx_ring[i]);
  }
}
```

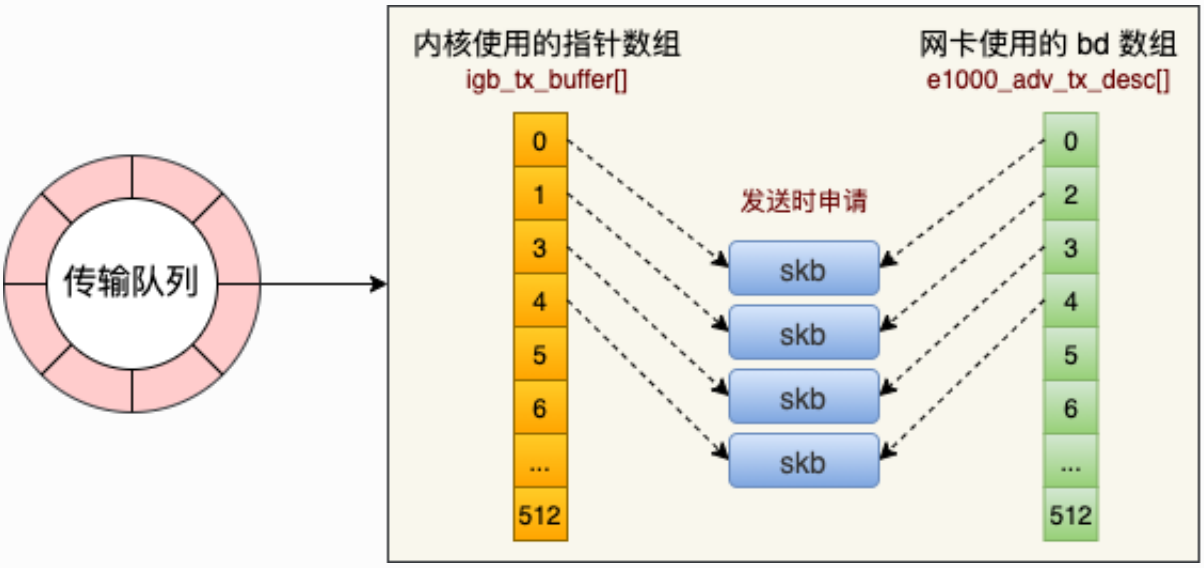
真正的 RingBuffer 构造过程是在 igb_setup_tx_resources 中完成的。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
int igb_setup_tx_resources(struct igb_ring *tx_ring)
{
  //1.申请 igb_tx_buffer 数组内存
  size = sizeof(struct igb_tx_buffer) * tx_ring->count;
  tx_ring->tx_buffer_info = vzalloc(size);

  //2.申请 e1000_adv_tx_desc DMA 数组内存
  tx_ring->size = tx_ring->count * sizeof(union e1000_adv_tx_desc);
  tx_ring->size = ALIGN(tx_ring->size, 4096);
  tx_ring->desc = dma_alloc_coherent(dev, tx_ring->size,
            &tx_ring->dma, GFP_KERNEL);

  //3.初始化队列成员
  tx_ring->next_to_use = 0;
  tx_ring->next_to_clean = 0;
}
```

从上述源码可以看到，实际上一个 RingBuffer 的内部不仅仅是一个环形队列数组，而是有两个。

1）igb_tx_buffer 数组：这个数组是内核使用的，通过 vzalloc 申请的。
2）e1000_adv_tx_desc 数组：这个数组是网卡硬件使用的，硬件是可以通过 DMA 直接访问这块内存，通过 dma_alloc_coherent 分配。

这个时候它们之间还没有啥联系。将来在发送的时候，这两个环形数组中相同位置的指针将都将指向同一个 skb。这样，内核和硬件就能共同访问同样的数据了，内核往 skb 里写数据，网卡硬件负责发送。



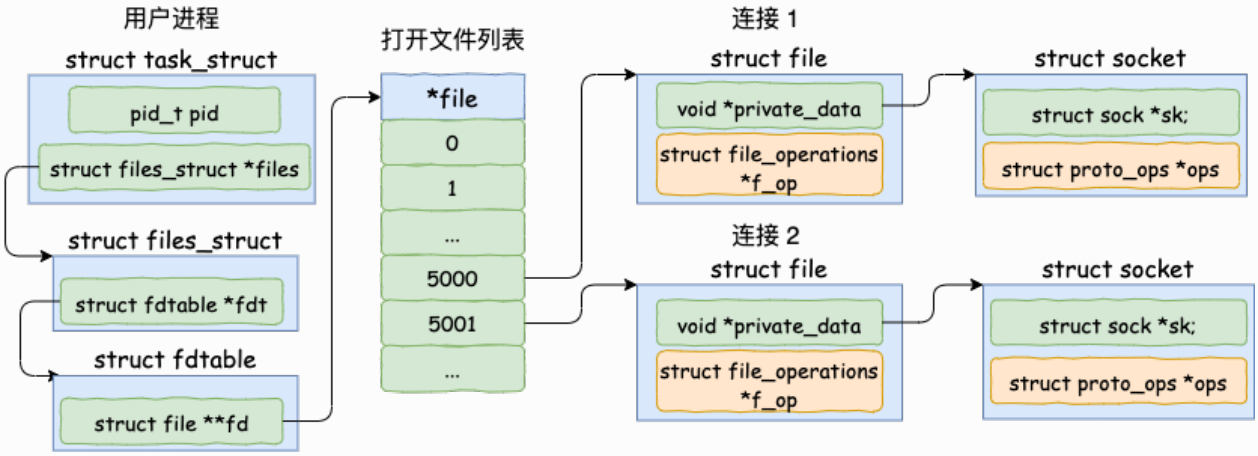
最后调用 netif_tx_start_all_queues 开启队列。另外，对于硬中断的处理函数 igb_msix_ring 其实也是在 __igb_open 中注册的。

三、ACCEPT 创建新 SOCKET

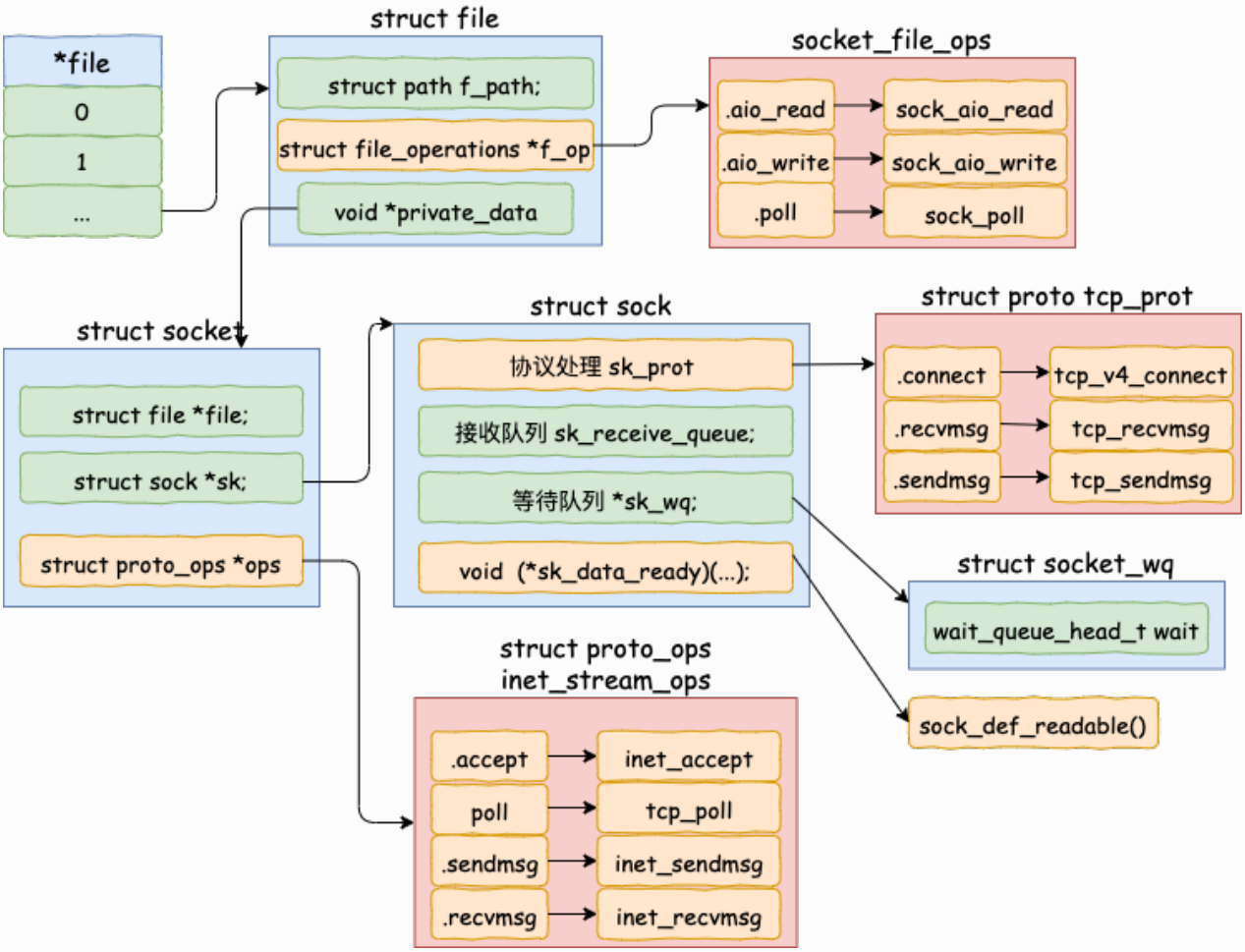
在发送数据之前，我们往往还需要一个已经建立好连接的 socket。

我们就以开篇服务器缩微源代码中提到的 accept 为例，当 accept 之后，进程会创建一个新的 socket 出来，然后把它放到当前进程的打开文件列表中，专门用于和对应的客户端通信。

假设服务器进程通过 accept 和客户端建立了两条连接，我们来简单看一下这两条连接和进程的关联关系。



其中代表一条连接的 socket 内核对象更为具体一点的结构图如下。



为了避免喧宾夺主，accept 详细的源码过程这里就不介绍了，感兴趣请参考 《图解 | 深入揭秘 epoll 是如何实现 IO 多路复用的！》。一文中的第一部分。

今天我们还是把重点放到数据发送过程上。

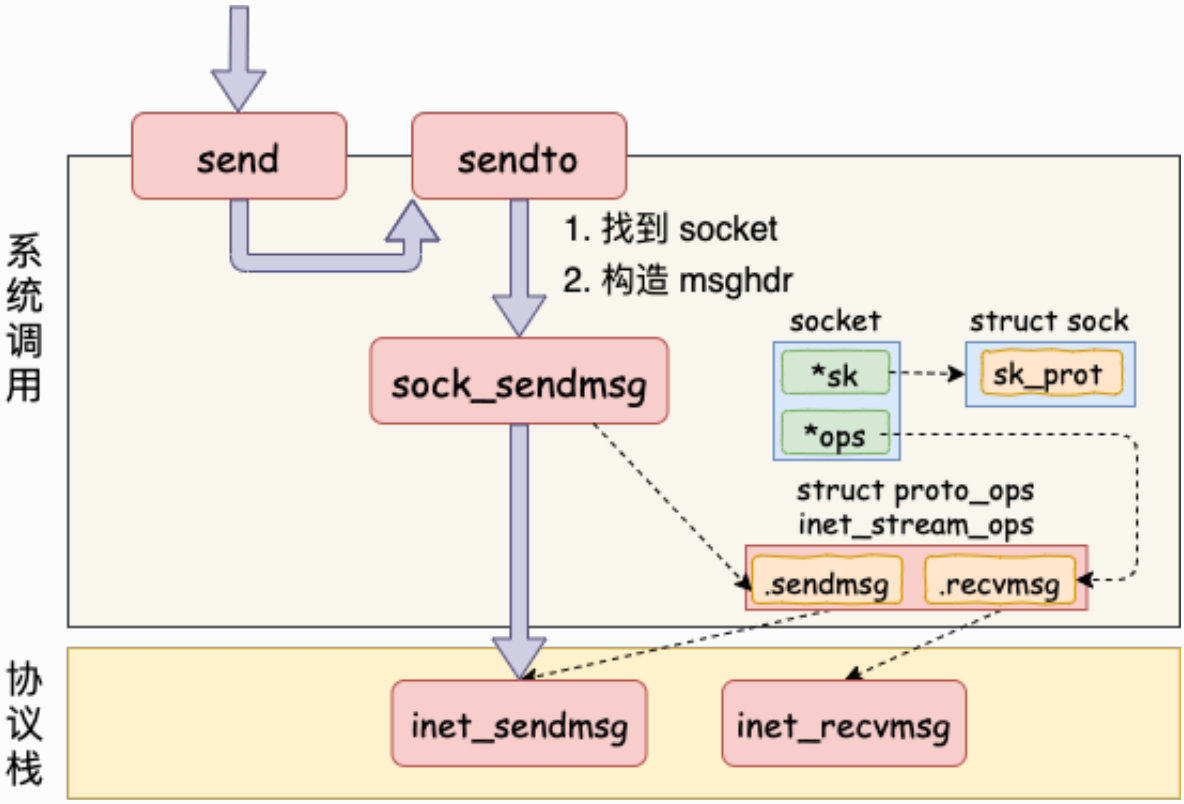
四、发送数据真正开始

4.1 send 系统调用实现

send 系统调用的源码位于文件 net/socket.c 中。在这个系统调用里，内部其实真正使用的是 sendto 系统调用。整个调用链条虽然不短，但其实主要只干了两件简单的事情，

- 第一是在内核中把真正的 socket 找出来，在这个对象里记录着各种协议栈的函数地址。
- 第二是构造一个 struct msghdr 对象，把用户传入的数据，比如 buffer地址、数据长度啥的，统统都装进去.

剩下的事情就交给下一层，协议栈里的函数 inet_sendmsg 了，其中 inet_sendmsg 函数的地址是通过 socket 内核对象里的 ops 成员找到的。大致流程如图。



有了上面的了解，我们再看起源码就要容易许多了。源码如下：

```
//file: net/socket.c
SYSCALL_DEFINE4(send, int, fd, void __user *, buff, size_t, len,
    unsigned int, flags)
{
  return sys_sendto(fd, buff, len, flags, NULL, 0);
}

SYSCALL_DEFINE6(......)
{
  //1.根据 fd 查找到 socket
  sock = sockfd_lookup_light(fd, &err, &fput_needed);

  //2.构造 msghdr
  struct msghdr msg;
  struct iovec iov;

  iov.iov_base = buff;
  iov.iov_len = len;
  msg.msg_iovlen = 1;

  msg.msg_iov = &iov;
  msg.msg_flags = flags;
  ......

  //3.发送数据
  sock_sendmsg(sock, &msg, len);
}
```

从源码可以看到，我们在用户态使用的 send 函数和 sendto 函数其实都是 sendto 系统调用实现的。send 只是为了方便，封装出来的一个更易于调用的方式而已。

在 sendto 系统调用里，首先根据用户传进来的 socket 句柄号来查找真正的 socket 内核对象。接着把用户请求的 buff、len、flag 等参数都统统打包到一个 struct msghdr 对象中。

接着调用了 sock_sendmsg => __sock_sendmsg ==> __sock_sendmsg_nosec。
在__sock_sendmsg_nosec 中，调用将会由系统调用进入到协议栈，我们来看它的源码。

```
//file: net/socket.c
static inline int __sock_sendmsg_nosec(...)
{
  ......
  return sock->ops->sendmsg(iocb, sock, msg, size);
}
```

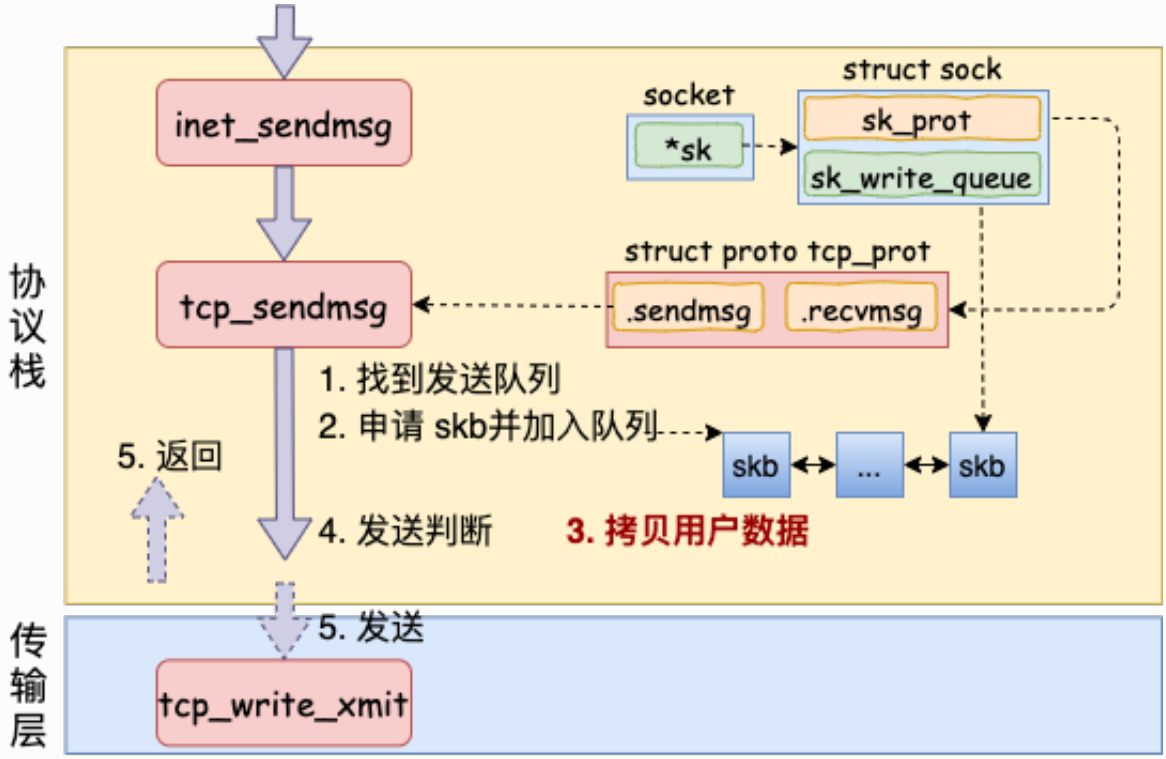
通过第三节里的 socket 内核对象结构图，我们可以看到，这里调用的是 sock->ops->sendmsg 实际执行的是 inet_sendmsg。这个函数是 AF_INET 协议族提供的通用发送函数。

4.2 传输层处理

1）传输层拷贝

在进入到协议栈 inet_sendmsg 以后，内核接着会找到 socket 上的具体协议发送函数。对于 TCP 协议来说，那就是 tcp_sendmsg（同样也是通过 socket 内核对象找到的）。

在这个函数中，内核会申请一个内核态的 skb 内存，将用户待发送的数据拷贝进去。注意这个时候不一定会真正开始发送，如果没有达到发送条件的话很可能这次调用直接就返回了。大概过程如图：



我们来看 inet_sendmsg 函数的源码。

```
//file: net/ipv4/af_inet.c
int inet_sendmsg(......)
{
  ......
  return sk->sk_prot->sendmsg(iocb, sk, msg, size);
}
```

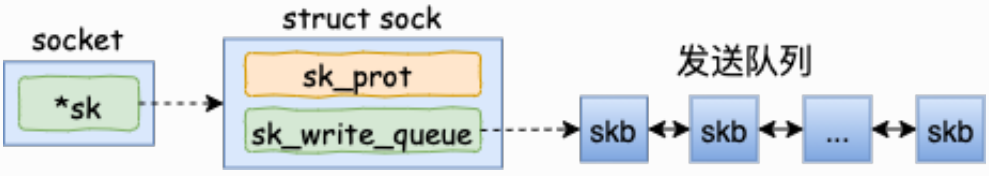
在这个函数中会调用到具体协议的发送函数。同样参考第三节里的 socket 内核对象结构图，我们看到对于 TCP 协议下的 socket 来说，来说 sk->sk_prot->sendmsg 指向的是 tcp_sendmsg（对于 UPD 来说是 udp_sendmsg）。

tcp_sendmsg 这个函数比较长，我们分多次来看它。 先看这一段

```
//file: net/ipv4/tcp.c
int tcp_sendmsg(...)
{
  while(...){
    while(...){
      //获取发送队列
      skb = tcp_write_queue_tail(sk);

      //申请skb 并拷贝
      ......
    }
  }
}
```

```
//file: include/net/tcp.h
static inline struct sk_buff *tcp_write_queue_tail(const struct sock
*sk)
{
  return skb_peek_tail(&sk->sk_write_queue);
}
```

理解对 socket 调用 tcp_write_queue_tail 是理解发送的前提。如上所示，这个函数是在获取 socket 发送队列中的最后一个 skb。 skb 是 struct sk_buff 对象的简称，用户的发送队列就是该对象组成的一个链表。

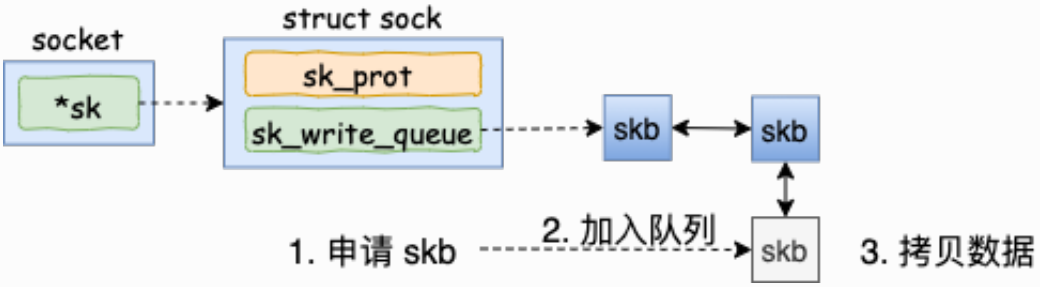


我们再接着看 tcp_sendmsg 的其它部分。

```
//file: net/ipv4/tcp.c
```

```
int tcp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr
*msg,
    size_t size)
{
  //获取用户传递过来的数据和标志
  iov = msg->msg_iov; //用户数据地址
  iovlen = msg->msg_iovlen; //数据块数为1
  flags = msg->msg_flags; //各种标志

  //遍历用户层的数据块
  while (--iovlen >= 0) {

    //待发送数据块的地址
    unsigned char __user *from = iov->iov_base;

    while (seglen > 0) {

      //需要申请新的 skb
      if (copy <= 0) {

        //申请 skb，并添加到发送队列的尾部
        skb = sk_stream_alloc_skb(sk,
                select_size(sk, sg),
                sk->sk_allocation);

        //把 skb 挂到socket的发送队列上
        skb_entail(sk, skb);
      }

      // skb 中有足够的空间
      if (skb_availroom(skb) > 0) {
        //拷贝用户空间的数据到内核空间，同时计算校验和
        //from是用户空间的数据地址
        skb_add_data_nocache(sk, skb, from, copy);
      }
      ......
```

这个函数比较长，不过其实逻辑并不复杂。其中 msg->msg_iov 存储的是用户态内存的要发送的数据的 buffer。接下来在内核态申请内核内存，比如 skb，并把用户内存里的数据拷贝到内核态内存中。**这就会涉及到一次或者几次内存拷贝的开销。**



至于内核什么时候真正把 skb 发送出去。在 tcp_sendmsg 中会进行一些判断。

```
//file: net/ipv4/tcp.c
int tcp_sendmsg(...)
{
  while(...){
    while(...){
      //申请内核内存并进行拷贝

      //发送判断
      if (forced_push(tp)) {
        tcp_mark_push(tp, skb);
        __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
      } else if (skb == tcp_send_head(sk))
        tcp_push_one(sk, mss_now);
      }
      continue;
    }
  }
}
```

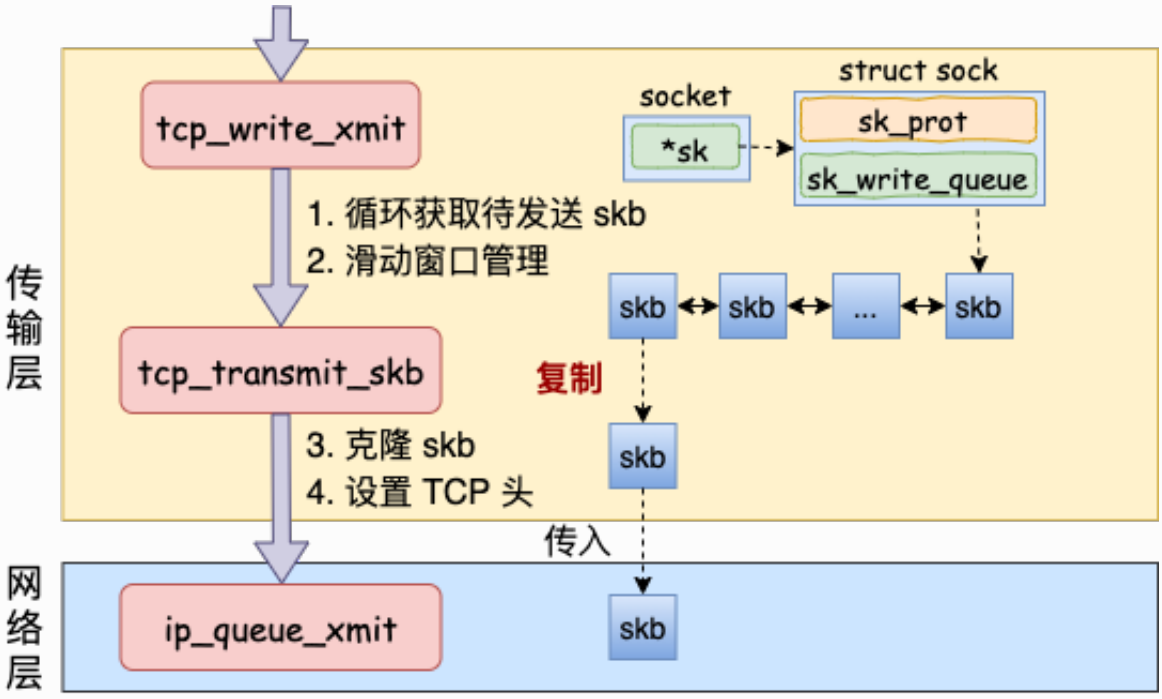
只有满足 forced_push(tp) 或者 skb == tcp_send_head(sk) 成立的时候，内核才会真正启动发送数据包。其中 forced_push(tp) 判断的是未发送的数据数据是否已经超过最大窗口的一半了。

条件都不满足的话，**这次的用户要发送的数据只是拷贝到内核就算完事了！**

2）传输层发送

假设现在内核发送条件已经满足了，我们再来跟踪一下实际的发送过程。 对于上小节函数中，当满足真正发送条件的时候，无论调用的是 __tcp_push_pending_frames 还是 tcp_push_one 最终都实际会执行到 tcp_write_xmit。

所以我们直接从 tcp_write_xmit 看起，这个函数处理了传输层的拥塞控制、滑动窗口相关的工作。满足窗口要求的时候，设置一下 TCP 头然后将 skb 传到更低的网络层进行处理。



我们来看下 tcp_write_xmit 的源码。

```
//file: net/ipv4/tcp_output.c
static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now, int
nonagle,
          int push_one, gfp_t gfp)
{
  //循环获取待发送 skb
  while ((skb = tcp_send_head(sk)))
  {
    //滑动窗口相关
    cwnd_quota = tcp_cwnd_test(tp, skb);
    tcp_snd_wnd_test(tp, skb, mss_now);
    tcp_mss_split_point(...);
    tso_fragment(sk, skb, ...);
    ......

    //真正开启发送
    tcp_transmit_skb(sk, skb, 1, gfp);
  }
}
```

可以看到我们之前在网络协议里学的滑动窗口、拥塞控制就是在这个函数中完成的，这部分就不过多展开了，感兴趣同学自己找这段源码来读。我们今天只看发送主过程，那就走到了 tcp_transmit_skb。

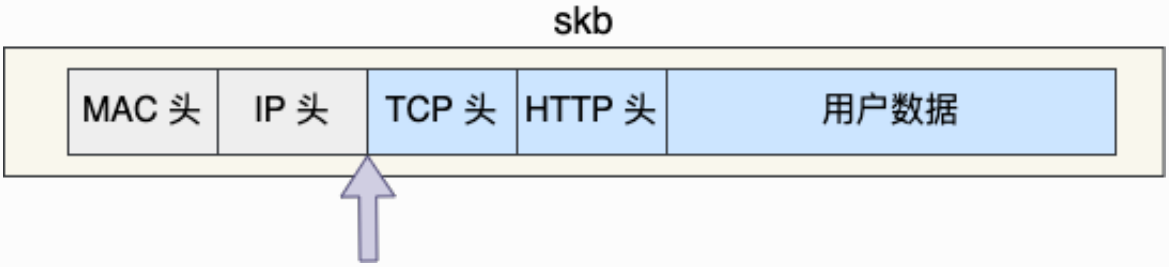
```
//file: net/ipv4/tcp_output.c
static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int
clone_it,
        gfp_t gfp_mask)
{
  //1.克隆新 skb 出来
```

```
  if (likely(clone_it)) {
    skb = skb_clone(skb, gfp_mask);
    ......
  }

  //2.封装 TCP 头
  th = tcp_hdr(skb);
  th->source    = inet->inet_sport;
  th->dest    = inet->inet_dport;
  th->window    = ...;
  th->urg     = ...;
  ......

  //3.调用网络层发送接口
  err = icsk->icsk_af_ops->queue_xmit(skb, &inet->cork.fl);
}
```

第一件事是先克隆一个新的 skb，这里重点说下为什么要复制一个 skb 出来呢?

是因为 skb 后续在调用网络层，最后到达网卡发送完成的时候，这个 skb 会被释放掉。而我们知道 TCP 协议是支持丢失重传的，在收到对方的 ACK 之前，这个 skb 不能被删除。所以内核的做法就是每次调用网卡发送的时候，实际上传递出去的是 skb 的一个拷贝。等收到 ACK 再真正删除。

第二件事是修改 skb 中的 TCP header，根据实际情况把 TCP 头设置好。这里要介绍一个小技巧，skb 内部其实包含了网络协议中所有的 header。在设置 TCP 头的时候，只是把指针指向 skb 的合适位置。后面再设置 IP 头的时候，在把指针挪一挪就行，避免频繁的内存申请和拷贝，效率很高。



tcp_transmit_skb 是发送数据位于传输层的最后一步，接下来就可以进入到网络层进行下一层的操作了。调用了网络层提供的发送接口icsk->icsk_af_ops->queue_xmit()。

在下面的这个源码中，我们的知道了 queue_xmit 其实指向的是 ip_queue_xmit 函数。

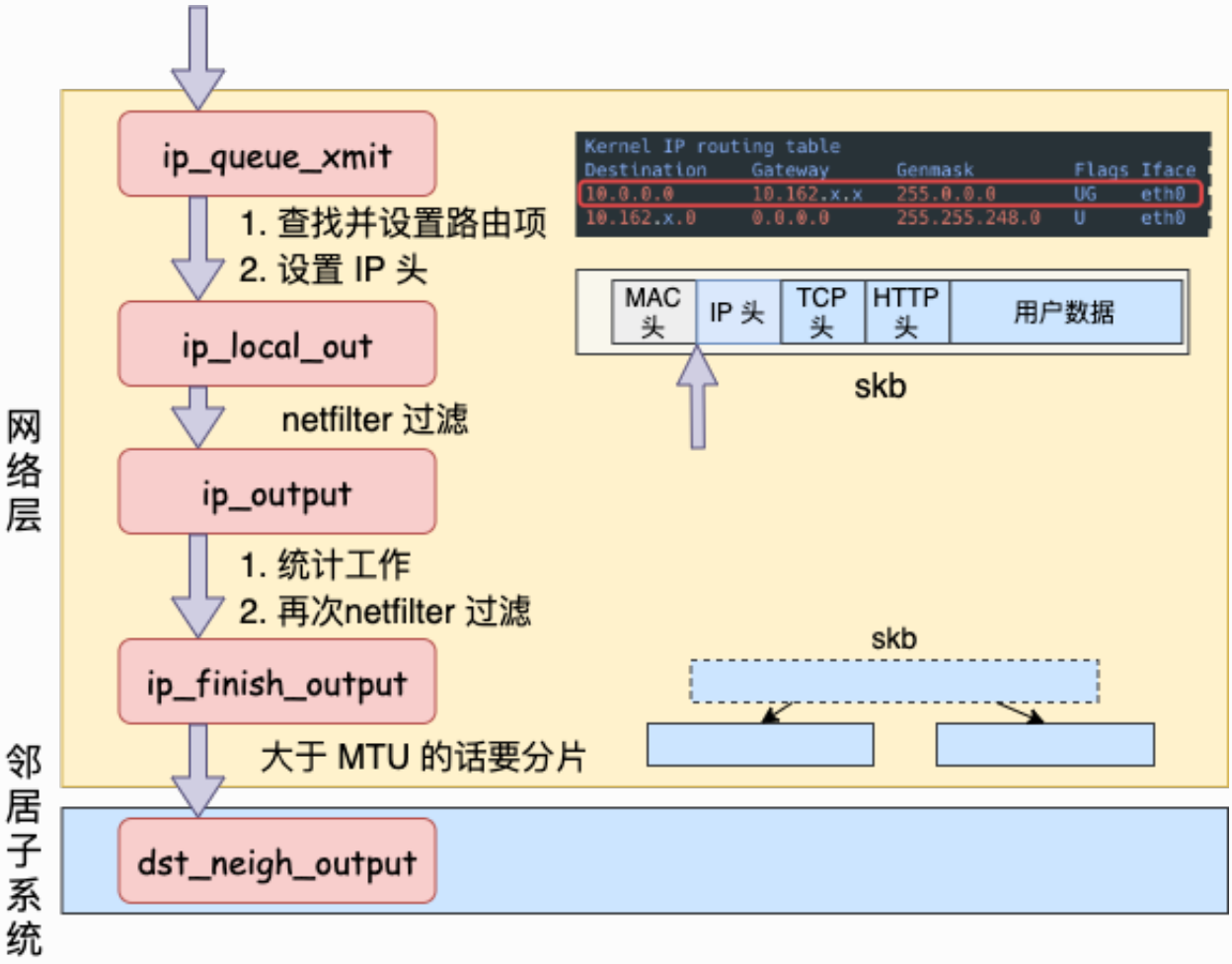
```
//file: net/ipv4/tcp_ipv4.c
const struct inet_connection_sock_af_ops ipv4_specific = {
  .queue_xmit     = ip_queue_xmit,
  .send_check     = tcp_v4_send_check,
  ...
}
```

自此，传输层的工作也就都完成了。 数据离开了传输层，接下来将会进入到内核在网络层的实现里。

4.3 网络层发送处理

Linux 内核网络层的发送的实现位于 net/ipv4/ip_output.c 这个文件。传输层调用到的 ip_queue_xmit 也在这里。（从文件名上也能看出来进入到 IP 层了，源文件名已经从 tcp_xxx 变成了 ip_xxx。）

在网络层里主要处理路由项查找、IP 头设置、netfilter 过滤、skb 切分（大于 MTU 的话）等几项工作，处理完这些工作后会交给更下层的邻居子系统来处理。



我们来看网络层入口函数 ip_queue_xmit 的源码：

```
//file: net/ipv4/ip_output.c
int ip_queue_xmit(struct sk_buff *skb, struct flowi *fl)
{
  //检查 socket 中是否有缓存的路由表
  rt = (struct rtable *)__sk_dst_check(sk, 0);
  if (rt == NULL) {
    //没有缓存则展开查找
    //则查找路由项， 并缓存到 socket 中
    rt = ip_route_output_ports(...);
    sk_setup_caps(sk, &rt->dst);
  }

  //为 skb 设置路由表
  skb_dst_set_noref(skb, &rt->dst);

  //设置 IP header
  iph = ip_hdr(skb);
  iph->protocol = sk->sk_protocol;
  iph->ttl      = ip_select_ttl(inet, &rt->dst);
  iph->frag_off = ...;

  //发送
  ip_local_out(skb);
```

```
}
```

ip_queue_xmit 已经到了网络层，在这个函数里我们看到了网络层相关的功能路由项查找，如果找到了则设置到 skb 上（没有路由的话就直接报错返回了）。

在 Linux 上通过 route 命令可以看到你本机的路由配置。

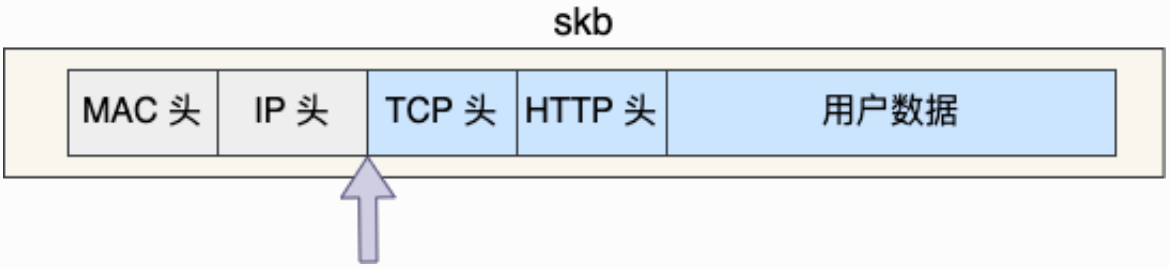
```
[                    ~]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.0.0        10.             255.0.0.0       UG    0      0        0 eth0
10.             0.0.0.0         255.255.248.0   U     0      0        0 eth0
169             0.0.0.0         255.255.0.0     U     1002   0        0 eth0
```

在路由表中，可以查到某个目的网络应该通过哪个 Iface（网卡），哪个 Gateway（网卡）发送出去。查找出来以后缓存到 socket 上，下次再发送数据就不用查了。

接着把路由表地址也放到 skb 里去。

```
//file: include/linux/skbuff.h
struct sk_buff {
  //保存了一些路由相关信息
  unsigned long   _skb_refdst;
}
```

接下来就是定位到 skb 里的 IP 头的位置上，然后开始按照协议规范设置 IP header。



再通过 ip_local_out 进入到下一步的处理。

```
//file: net/ipv4/ip_output.c
int ip_local_out(struct sk_buff *skb)
{
  //执行 netfilter 过滤
  err = __ip_local_out(skb);

  //开始发送数据
  if (likely(err == 1))
    err = dst_output(skb);
  ......
```

在 ip_local_out => __ip_local_out => nf_hook 会执行 netfilter 过滤。如果你使用 iptables 配置了一些规则，那么这里将检测是否命中规则。
如果你设置了非常复杂的 netfilter 规则，在这个函数这里将会导致你的进程 CPU 开销会极大增加。

还是不多展开说，继续只聊和发送有关的过程 dst_output。

```
//file: include/net/dst.h
static inline int dst_output(struct sk_buff *skb)
{
  return skb_dst(skb)->output(skb);
}
```

此函数找到到这个 skb 的路由表（dst 条目），然后调用路由表的 output 方法。这又是一个函数指针，指向的是 ip_output 方法。

```
//file: net/ipv4/ip_output.c
int ip_output(struct sk_buff *skb)
{
  //统计
  .....

  //再次交给 netfilter，完毕后回调 ip_finish_output
  return NF_HOOK_COND(NFPROTO_IPV4, NF_INET_POST_ROUTING, skb, NULL,
dev,
        ip_finish_output,
        !(IPCB(skb)->flags & IPSKB_REROUTED));
}
```

在 ip_output 中进行一些简单的，统计工作，再次执行 netfilter 过滤。过滤通过之后回调 ip_finish_output。

```
//file: net/ipv4/ip_output.c
static int ip_finish_output(struct sk_buff *skb)
{
  //大于 mtu 的话就要进行分片了
  if (skb->len > ip_skb_dst_mtu(skb) && !skb_is_gso(skb))
    return ip_fragment(skb, ip_finish_output2);
  else
    return ip_finish_output2(skb);
}
```

在 ip_finish_output 中我们看到，**如果数据大于 MTU 的话，是会执行分片的。**

> 实际 MTU 大小确定依赖 MTU 发现，以太网帧为 1500 字节。之前 QQ 团队在早期的时候，会尽量控制自己数据包尺寸小于 MTU，通过这种方式来优化网络性能。因为分片会带来两个问题：1、需要进行额外的切分处理，有额外性能开销。2、只要一个分片丢失，整个包都得重传。所以避免分片既杜绝了分片开销，也大大降低了重传率。

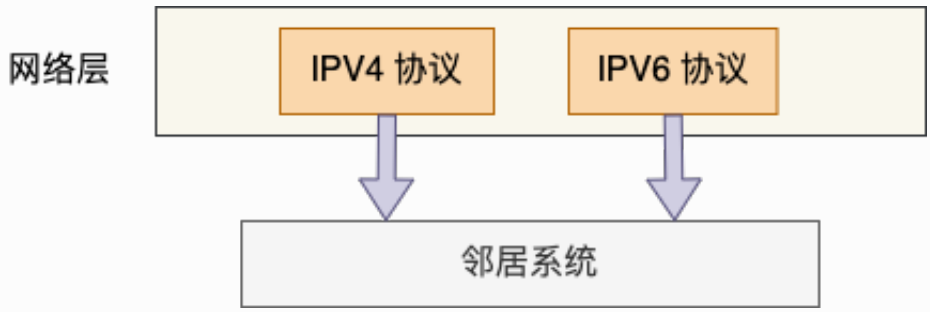
在 ip_finish_output2 中，终于发送过程会进入到下一层，邻居子系统中。

```
//file: net/ipv4/ip_output.c
static inline int ip_finish_output2(struct sk_buff *skb)
{
  //根据下一跳 IP 地址查找邻居项，找不到就创建一个
  nexthop = (__force u32) rt_nexthop(rt, ip_hdr(skb)->daddr);
  neigh = __ipv4_neigh_lookup_noref(dev, nexthop);
  if (unlikely(!neigh))
    neigh = __neigh_create(&arp_tbl, &nexthop, dev, false);

  //继续向下层传递
  int res = dst_neigh_output(dst, neigh, skb);
}
```

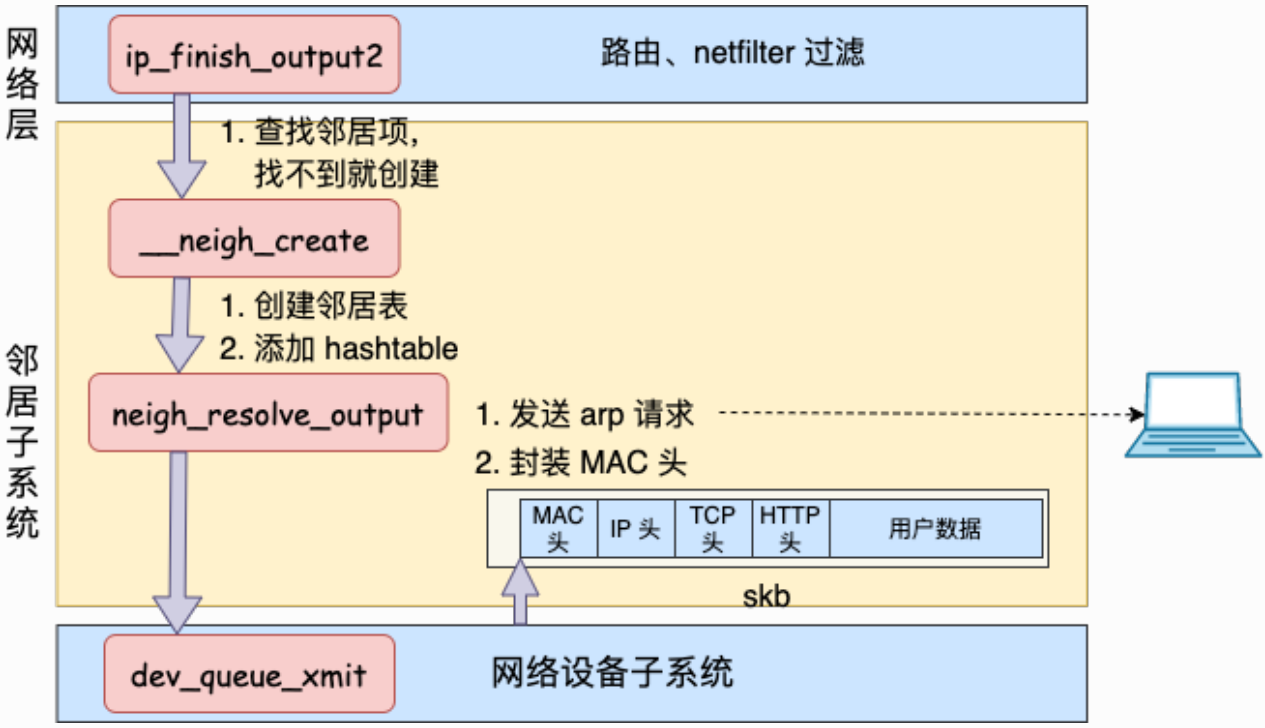
4.4 邻居子系统

邻居子系统是位于网络层和数据链路层中间的一个系统，其作用是对网络层提供一个封装，让网络层不必关心下层的地址信息，让下层来决定发送到哪个 MAC 地址。

而且这个邻居子系统并不位于协议栈 net/ipv4/ 目录内，而是位于 net/core/neighbour.c。因为无论是对于 IPv4 还是 IPv6 ，都需要使用该模块。



在邻居子系统里主要是查找或者创建邻居项，在创造邻居项的时候，有可能会发出实际的 arp 请求。然后封装一下 MAC 头，将发送过程再传递到更下层的网络设备子系统。大致流程如图。



理解了大致流程，我们再回头看源码。在上面小节 ip_finish_output2 源码中调用了 __ipv4_neigh_lookup_noref。它是在 arp 缓存中进行查找，其第二个参数传入的是路由下一跳 IP 信息。

```
//file: include/net/arp.h
extern struct neigh_table arp_tbl;
static inline struct neighbour *__ipv4_neigh_lookup_noref(
  struct net_device *dev, u32 key)
{
  struct neigh_hash_table *nht = rcu_dereference_bh(arp_tbl.nht);

  //计算 hash 值，加速查找
  hash_val = arp_hashfn(......);
  for (n = rcu_dereference_bh(nht->hash_buckets[hash_val]);
     n != NULL;
     n = rcu_dereference_bh(n->next)) {
    if (n->dev == dev && *(u32 *)n->primary_key == key)
      return n;
  }
}
```

如果查找不到，则调用 __neigh_create 创建一个邻居。

```
//file: net/core/neighbour.c
struct neighbour *__neigh_create(......)
{
  //申请邻居表项
  struct neighbour *n1, *rc, *n = neigh_alloc(tbl, dev);

  //构造赋值
  memcpy(n->primary_key, pkey, key_len);
```

```
  n->dev = dev;
  n->parms->neigh_setup(n);

  //最后添加到邻居 hashtable 中
  rcu_assign_pointer(nht->hash_buckets[hash_val], n);
  ......
```

有了邻居项以后，此时仍然还不具备发送 IP 报文的能力，因为目的 MAC 地址还未获取。
调用 dst_neigh_output 继续传递 skb。

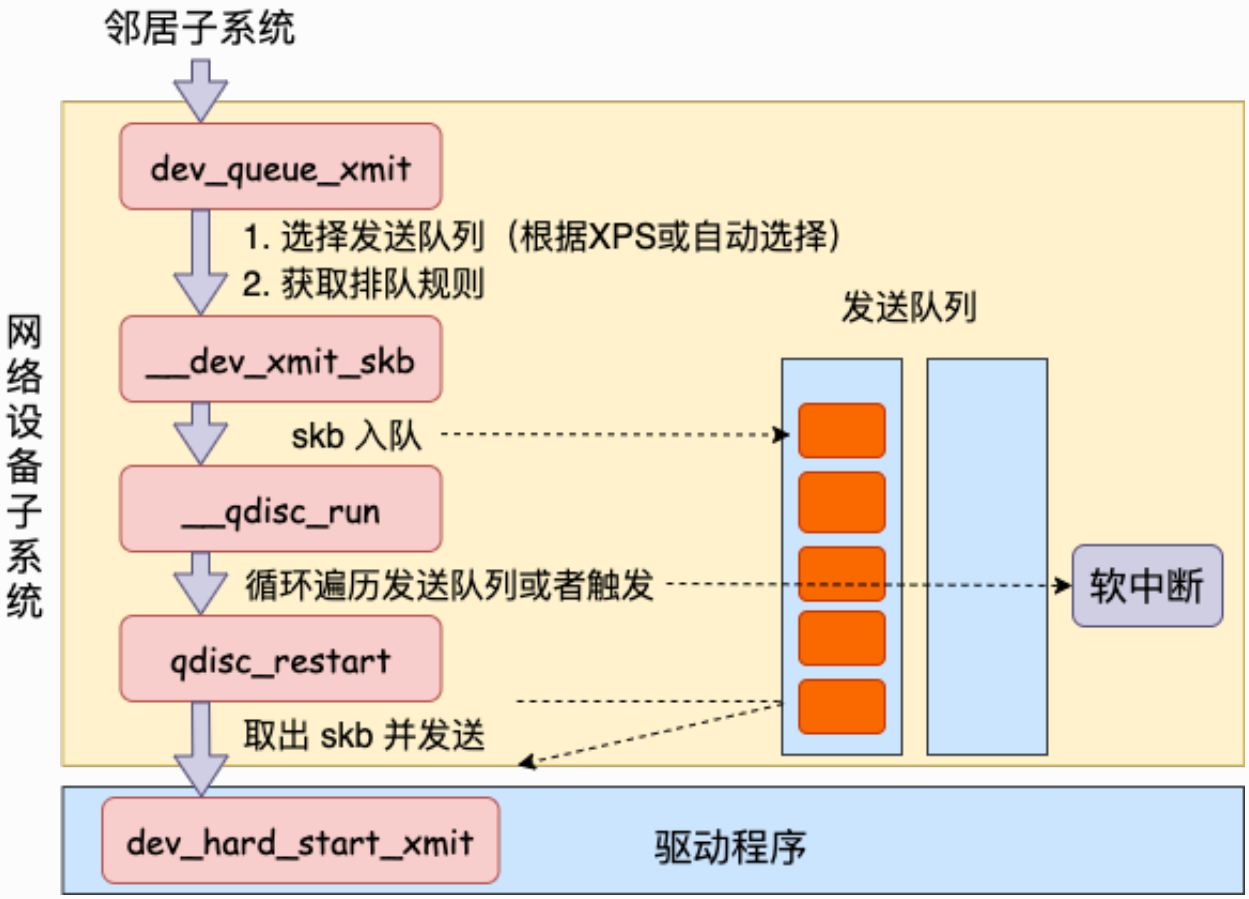
```
//file: include/net/dst.h
static inline int dst_neigh_output(struct dst_entry *dst,
      struct neighbour *n, struct sk_buff *skb)
{
  ......
  return n->output(n, skb);
}
```

调用 output，实际指向的是 neigh_resolve_output。在这个函数内部有可能会发出 arp 网络请求。

```
//file: net/core/neighbour.c
int neigh_resolve_output(){

  //注意：这里可能会触发 arp 请求
  if (!neigh_event_send(neigh, skb)) {

    //neigh->ha 是 MAC 地址
    dev_hard_header(skb, dev, ntohs(skb->protocol),
                neigh->ha, NULL, skb->len);
    //发送
    dev_queue_xmit(skb);
  }
}
```

当获取到硬件 MAC 地址以后，就可以封装 skb 的 MAC 头了。最后调用 dev_queue_xmit 将 skb 传递给 Linux 网络设备子系统。

4.5 网络设备子系统



邻居子系统通过 dev_queue_xmit 进入到网络设备子系统中来。

```
//file: net/core/dev.c
int dev_queue_xmit(struct sk_buff *skb)
{
  //选择发送队列
  txq = netdev_pick_tx(dev, skb);

  //获取与此队列关联的排队规则
  q = rcu_dereference_bh(txq->qdisc);

  //如果有队列，则调用__dev_xmit_skb 继续处理数据
  if (q->enqueue) {
    rc = __dev_xmit_skb(skb, q, dev, txq);
    goto out;
  }

  //没有队列的是回环设备和隧道设备
  .......
}
```

开篇第二节网卡启动准备里我们说过，网卡是有多个发送队列的（尤其是现在的网卡）。上面对 netdev_pick_tx 函数的调用就是选择一个队列进行发送。

netdev_pick_tx 发送队列的选择受 XPS 等配置的影响，而且还有缓存，也是一套小复杂的逻辑。这里我们只关注两个逻辑，首先会获取用户的 XPS 配置，否则就自动计算了。代码见 netdev_pick_tx => __netdev_pick_tx。

```
//file: net/core/flow_dissector.c
u16 __netdev_pick_tx(struct net_device *dev, struct sk_buff *skb)
{
  //获取 XPS 配置
  int new_index = get_xps_queue(dev, skb);

  //自动计算队列
  if (new_index < 0)
    new_index = skb_tx_hash(dev, skb);}
```

然后获取与此队列关联的 qdisc。在 linux 上通过 tc 命令可以看到 qdisc 类型，例如对于我的某台多队列网卡机器上是 mq disc。

```
#tc qdisc
qdisc mq 0: dev eth0 root
```

大部分的设备都有队列（回环设备和隧道设备除外），所以现在我们进入到 __dev_xmit_skb。

```
//file: net/core/dev.c
static inline int __dev_xmit_skb(struct sk_buff *skb, struct Qdisc *q,
         struct net_device *dev,
         struct netdev_queue *txq)
{
  //1.如果可以绕开排队系统
  if ((q->flags & TCQ_F_CAN_BYPASS) && !qdisc_qlen(q) &&
      qdisc_run_begin(q)) {
    ......
  }

  //2.正常排队
  else {

    //入队
    q->enqueue(skb, q)

    //开始发送
    __qdisc_run(q);
  }
}
```

上述代码中分两种情况，1 是可以 bypass（绕过）排队系统的，另外一种是正常排队。我们只看第二种情况。

先调用 q->enqueue 把 skb 添加到队列里。然后调用 __qdisc_run 开始发送。

```
//file: net/sched/sch_generic.c
void __qdisc_run(struct Qdisc *q)
{
  int quota = weight_p;

  //循环从队列取出一个 skb 并发送
  while (qdisc_restart(q)) {

    // 如果发生下面情况之一，则延后处理：
    // 1. quota 用尽
    // 2. 其他进程需要 CPU
    if (--quota <= 0 || need_resched()) {
      //将触发一次 NET_TX_SOFTIRQ 类型 softirq
      __netif_schedule(q);
      break;
    }
  }
}
```

在上述代码中，我们看到 while 循环不断地从队列中取出 skb 并进行发送。注意，这个时候其实都占用的是用户进程的系统态时间(sy)。 只有当 quota 用尽或者其它进程需要 CPU 的时候才触发软中断进行发送。

所以这就是为什么一般服务器上查看 /proc/softirqs，一般 NET_RX 都要比 NET_TX 大的多的第二个原因。对于读来说，都是要经过 NET_RX 软中断，而对于发送来说，只有系统态配额用尽才让软中断上。

我们来把精力在放到 qdisc_restart 上，继续看发送过程。

```
static inline int qdisc_restart(struct Qdisc *q)
{
  //从 qdisc 中取出要发送的 skb
  skb = dequeue_skb(q);
  ...

  return sch_direct_xmit(skb, q, dev, txq, root_lock);
}
```

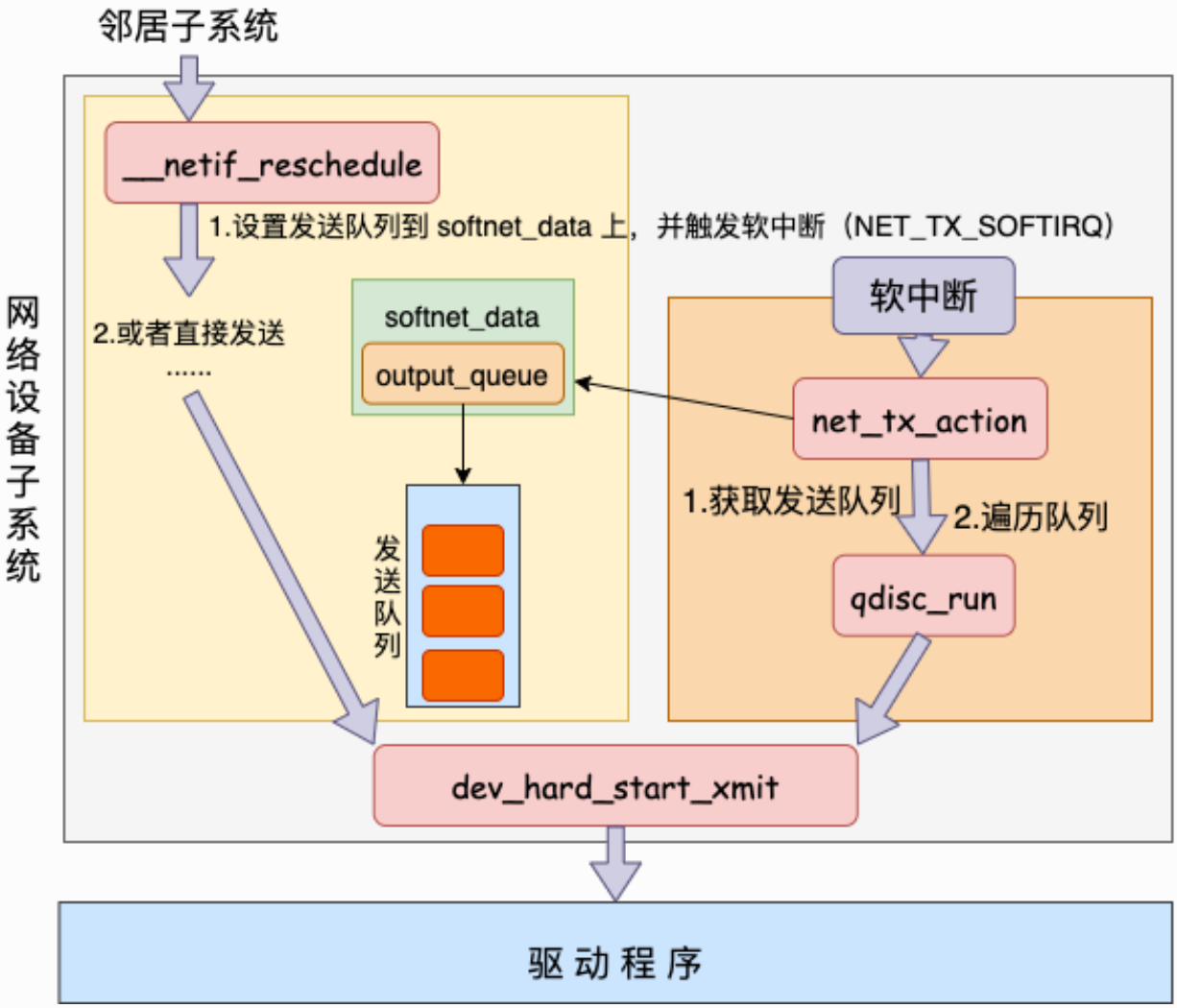
qdisc_restart 从队列中取出一个 skb，并调用 sch_direct_xmit 继续发送。

```
//file: net/sched/sch_generic.c
int sch_direct_xmit(struct sk_buff *skb, struct Qdisc *q,
      struct net_device *dev, struct netdev_queue *txq,
      spinlock_t *root_lock)
{
  //调用驱动程序来发送数据
  ret = dev_hard_start_xmit(skb, dev, txq);
}
```

4.6 软中断调度

在 4.5 咱们看到了如果系统态 CPU 发送网络包不够用的时候，会调用 __netif_schedule 触发一个软中断。该函数会进入到 __netif_reschedule，由它来实际发出 NET_TX_SOFTIRQ 类型软中断。

软中断是由内核线程来运行的，该线程会进入到 net_tx_action 函数，在该函数中能获取到发送队列，并也最终调用到驱动程序里的入口函数 dev_hard_start_xmit。




```
//file: net/core/dev.c
static inline void __netif_reschedule(struct Qdisc *q)
{
  sd = &__get_cpu_var(softnet_data);
  q->next_sched = NULL;
  *sd->output_queue_tailp = q;
  sd->output_queue_tailp = &q->next_sched;

  ......
  raise_softirq_irqoff(NET_TX_SOFTIRQ);
}
```

在该函数里在软中断能访问到的 softnet_data 里设置了要发送的数据队列，添加到了 output_queue 里了。紧接着触发了 NET_TX_SOFTIRQ 类型的软中断。（T 代表 transmit 传输）

软中断的入口代码我这里也不详细扒了，感兴趣的同学参考《图解Linux网络包接收过程》一文中的 3.2 小节 - ksoftirqd内核线程处理软中断。

我们直接从 NET_TX_SOFTIRQ softirq 注册的回调函数 net_tx_action讲起。用户态进程触发完软中断之后，会有一个软中断内核线程会执行到 net_tx_action。

牢记，这以后发送数据消耗的 CPU 就都显示在 si 这里了，不会消耗用户进程的系统时间了。

```
//file: net/core/dev.c
static void net_tx_action(struct softirq_action *h)
{
  //通过 softnet_data 获取发送队列
  struct softnet_data *sd = &__get_cpu_var(softnet_data);

  // 如果 output queue 上有 qdisc
  if (sd->output_queue) {

    // 将 head 指向第一个 qdisc
    head = sd->output_queue;

    //遍历 qdsics 列表
    while (head) {
      struct Qdisc *q = head;
      head = head->next_sched;

      //发送数据
      qdisc_run(q);
    }
  }
}
```

软中断这里会获取 softnet_data。前面我们看到进程内核态在调用 __netif_reschedule 的时候把发送队列写到 softnet_data 的 output_queue 里了。 软中断循环遍历 sd->output_queue 发送数据帧。

来看 qdisc_run，它和进程用户态一样，也会调用到 __qdisc_run。

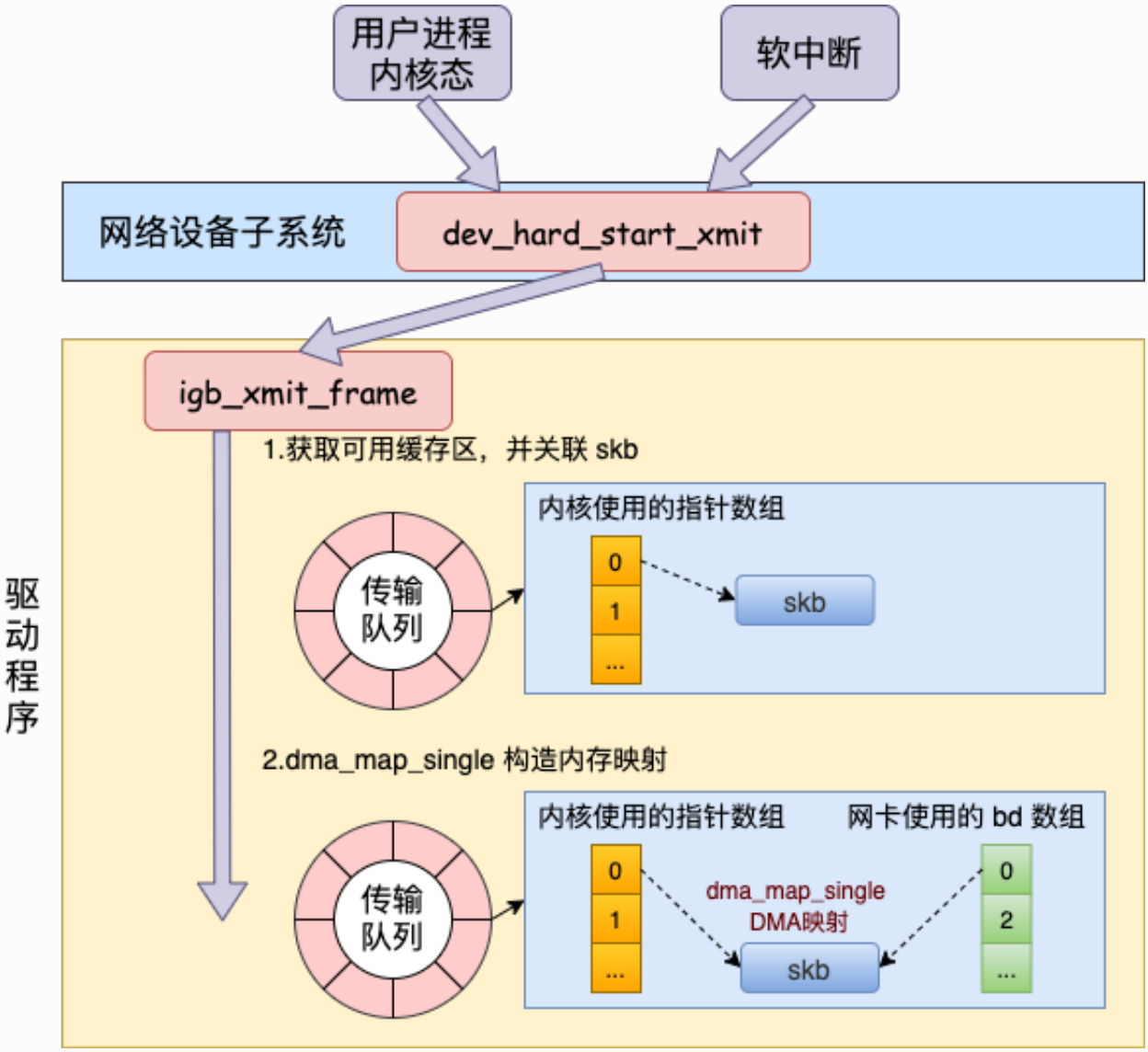
```
//file: include/net/pkt_sched.h
static inline void qdisc_run(struct Qdisc *q)
{
  if (qdisc_run_begin(q))
    __qdisc_run(q);
}
```

然后一样就是进入 qdisc_restart => sch_direct_xmit，直到驱动程序函数 dev_hard_start_xmit。

4.7 igb 网卡驱动发送

我们前面看到，无论是对于用户进程的内核态，还是对于软中断上下文，都会调用到网络设备子系统中的 dev_hard_start_xmit 函数。在这个函数中，会调用到驱动里的发送函数 igb_xmit_frame。

在驱动函数里，将 skb 会挂到 RingBuffer上，驱动调用完毕后，数据包将真正从网卡发送出去。



我们来看看实际的源码：

```
//file: net/core/dev.c
int dev_hard_start_xmit(struct sk_buff *skb, struct net_device *dev,
      struct netdev_queue *txq)
{
  //获取设备的回调函数集合 ops
  const struct net_device_ops *ops = dev->netdev_ops;

  //获取设备支持的功能列表
  features = netif_skb_features(skb);

  //调用驱动的 ops 里面的发送回调函数 ndo_start_xmit 将数据包传给网卡设备
  skb_len = skb->len;
  rc = ops->ndo_start_xmit(skb, dev);
}
```

其中 ndo_start_xmit 是网卡驱动要实现的一个函数，是在 net_device_ops 中定义的。

```
//file: include/linux/netdevice.h
struct net_device_ops {
  netdev_tx_t   (*ndo_start_xmit) (struct sk_buff *skb,
                struct net_device *dev);

}
```

在 igb 网卡驱动源码中，我们找到了。

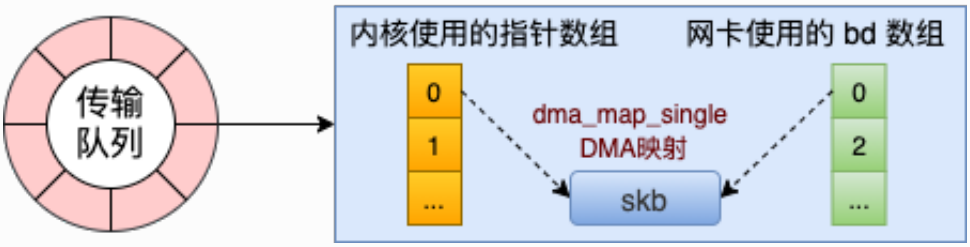
```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static const struct net_device_ops igb_netdev_ops = {
  .ndo_open    = igb_open,
  .ndo_stop    = igb_close,
  .ndo_start_xmit    = igb_xmit_frame,
  ...
};
```

也就是说，对于网络设备层定义的 ndo_start_xmit， igb 的实现函数是 igb_xmit_frame。这个函数是在网卡驱动初始化的时候被赋值的。具体初始化过程参见《图解Linux网络包接收过程》一文中的 2.4 节，网卡驱动初始化。

所以在上面网络设备层调用 ops->ndo_start_xmit 的时候，会实际上进入 igb_xmit_frame 这个函数中。我们进入这个函数来看看驱动程序是如何工作的。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static netdev_tx_t igb_xmit_frame(struct sk_buff *skb,
          struct net_device *netdev)
{
  ......
  return igb_xmit_frame_ring(skb, igb_tx_queue_mapping(adapter, skb));
}

netdev_tx_t igb_xmit_frame_ring(struct sk_buff *skb,
        struct igb_ring *tx_ring)
{
  //获取TX Queue 中下一个可用缓冲区信息
  first = &tx_ring->tx_buffer_info[tx_ring->next_to_use];
  first->skb = skb;
  first->bytecount = skb->len;
  first->gso_segs = 1;

  //igb_tx_map 函数准备给设备发送的数据。
  igb_tx_map(tx_ring, first, hdr_len);
}
```

在这里从网卡的发送队列的 RingBuffer 中取下来一个元素，并将 skb 挂到元素上。



igb_tx_map 函数处理将 skb 数据映射到网卡可访问的内存 DMA 区域。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static void igb_tx_map(struct igb_ring *tx_ring,
          struct igb_tx_buffer *first,
          const u8 hdr_len)
{
  //获取下一个可用描述符指针
  tx_desc = IGB_TX_DESC(tx_ring, i);

  //为 skb->data 构造内存映射，以允许设备通过 DMA 从 RAM 中读取数据
  dma = dma_map_single(tx_ring->dev, skb->data, size, DMA_TO_DEVICE);

  //遍历该数据包的所有分片,为 skb 的每个分片生成有效映射
  for (frag = &skb_shinfo(skb)->frags[0];; frag++) {

    tx_desc->read.buffer_addr = cpu_to_le64(dma);
    tx_desc->read.cmd_type_len = ...;
    tx_desc->read.olinfo_status = 0;
  }

  //设置最后一个descriptor
  cmd_type |= size | IGB_TXD_DCMD;
  tx_desc->read.cmd_type_len = cpu_to_le32(cmd_type);

  /* Force memory writes to complete before letting h/w know there
   * are new descriptors to fetch
   */
  wmb();
}
```

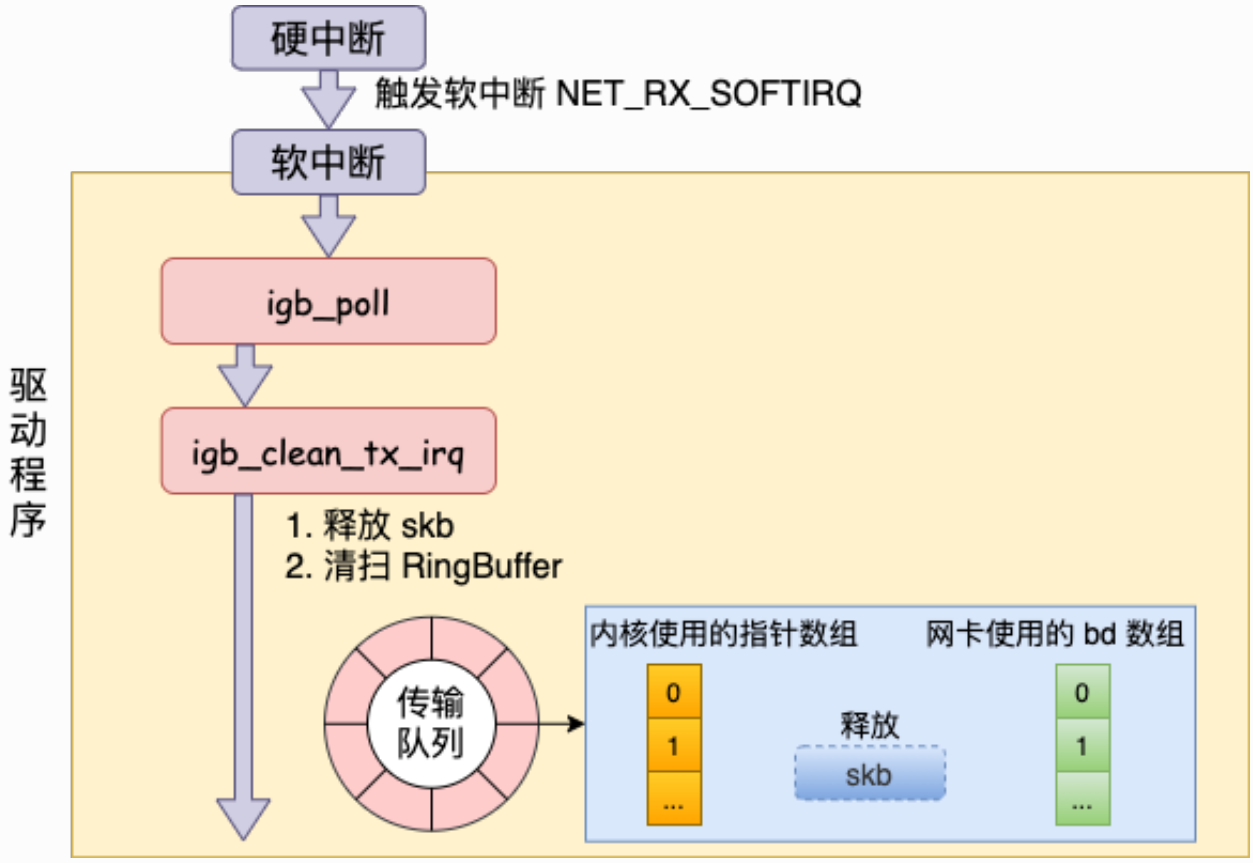
当所有需要的描述符都已建好，且 skb 的所有数据都映射到 DMA 地址后，驱动就会进入到它的最后一步，触发真实的发送。

4.8 发送完成硬中断

当数据发送完成以后，其实工作并没有结束。因为内存还没有清理。当发送完成的时候，网卡设备会触发一个硬中断来释放内存。

在《图解Linux网络包接收过程》一文中的 3.1 和 3.2 小节，我们详细讲述过硬中断和软中断的处理过程。

在发送硬中断里，会执行 RingBuffer 内存的清理工作，如图。



再回头看一下硬中断触发软中断的源码。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static inline void ____napi_schedule(...){
  list_add_tail(&napi->poll_list, &sd->poll_list);
  __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

这里有个很有意思的细节，无论硬中断是因为是有数据要接收，还是说发送完成通知，**从硬中断触发的软中断都是 NET_RX_SOFTIRQ**。这个我们在第一节说过了，这是软中断统计中 RX 要高于 TX 的一个原因。

好我们接着进入软中断的回调函数 igb_poll。在这个函数里，我们注意到有一行 igb_clean_tx_irq，参见源码：

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static int igb_poll(struct napi_struct *napi, int budget)
{
  //performs the transmit completion operations
  if (q_vector->tx.ring)
    clean_complete = igb_clean_tx_irq(q_vector);
  ...
}
```

我们来看看当传输完成的时候，igb_clean_tx_irq 都干啥了。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static bool igb_clean_tx_irq(struct igb_q_vector *q_vector)
{
  //free the skb
  dev_kfree_skb_any(tx_buffer->skb);

  //clear tx_buffer data
  tx_buffer->skb = NULL;
  dma_unmap_len_set(tx_buffer, len, 0);

  // clear last DMA location and unmap remaining buffers */
  while (tx_desc != eop_desc) {
  }
}
```

无非就是清理了 skb，解除了 DMA 映射等等。 到了这一步，传输才算是基本完成了。

为啥我说是基本完成，而不是全部完成了呢？因为传输层需要保证可靠性，所以 skb 其实还没有删除。它得等收到对方的 ACK 之后才会真正删除，那个时候才算是彻底的发送完毕。

公众号：「开发内功修炼」

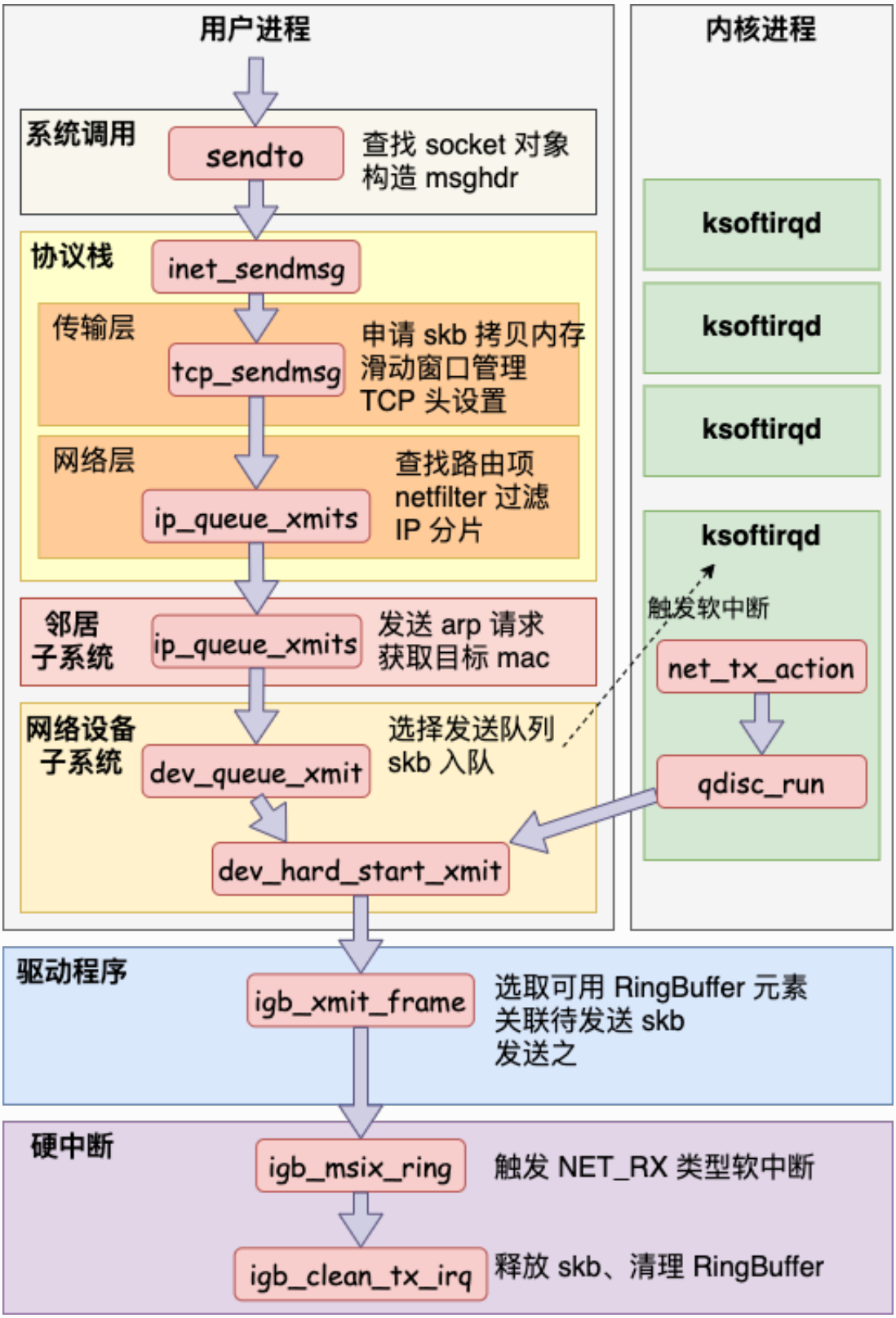
了解你的每一比特、用好你的每一纳秒！

公众号

作者微信

最后

用一张图总结一下整个发送过程



了解了整个发送过程以后，我们回头再来回顾开篇提到的几个问题。

1.我们在监控内核发送数据消耗的 CPU 时，是应该看 sy 还是 si？

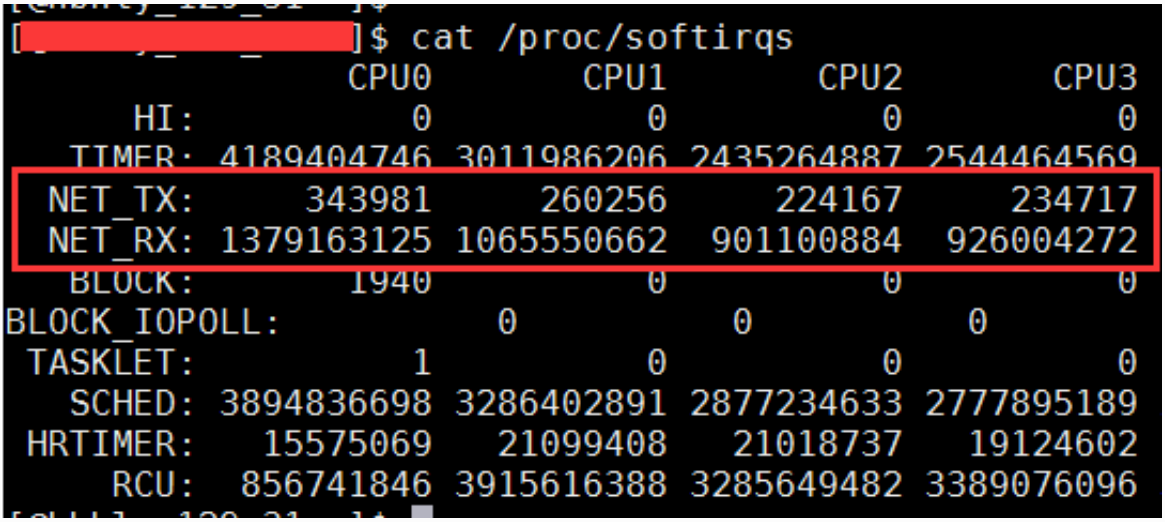
在网络包的发送过程中，用户进程（在内核态）完成了绝大部分的工作，甚至连调用驱动的事情都干了。 只有当内核态进程被切走前才会发起软中断。 发送过程中，绝大部分（90%）以上的开销都是在用户进程内核态消耗掉的。

只有一少部分情况下才会触发软中断（NET_TX 类型），由软中断 ksoftirqd 内核进程来发送。

所以，在监控网络 IO 对服务器造成的 CPU 开销的时候，不能仅仅只看 si，而是应该把 si、sy 都考虑进来。

2. 在服务器上查看 /proc/softirqs，为什么 NET_RX 要比 NET_TX 大的多的多?

之前我认为 NET_RX 是读取，NET_TX 是传输。对于一个既收取用户请求，又给用户返回的 Server 来说。 这两块的数字应该差不多才对，至少不会有数量级的差异。但事实上，飞哥手头的一台服务器是这样的：



经过今天的源码分析，发现这个问题的原因有两个。

第一个原因是当数据发送完成以后，通过硬中断的方式来通知驱动发送完毕。但是硬中断无论是有数据接收，还是对于发送完毕，触发的软中断都是 NET_RX_SOFTIRQ，而并不是 NET_TX_SOFTIRQ。

第二个原因是对于读来说，都是要经过 NET_RX 软中断的，都走 ksoftirqd 内核进程。而对于发送来说，绝大部分工作都是在用户进程内核态处理了，只有系统态配额用尽才会发出 NET_TX，让软中断上。

综上两个原因，那么在机器上查看 NET_RX 比 NET_TX 大的多就不难理解了。

3.发送网络数据的时候都涉及到哪些内存拷贝操作?

这里的内存拷贝，我们只特指待发送数据的内存拷贝。

第一次拷贝操作是内核申请完 skb 之后，这时候会将用户传递进来的 buffer 里的数据内容都拷贝到 skb 中。如果要发送的数据量比较大的话，这个拷贝操作开销还是不小的。

第二次拷贝操作是从传输层进入网络层的时候，每一个 skb 都会被克隆一个新的副本出来。网络层以及下面的驱动、软中断等组件在发送完成的时候会将这个副本删除。传输层保存着原始的 skb，在当网络对方没有 ack 的时候，还可以重新发送，以实现 TCP 中要求的可靠传输。

第三次拷贝不是必须的，只有当 IP 层发现 skb 大于 MTU 时才需要进行。会再申请额外的 skb，并将原来的 skb 拷贝为多个小的 skb。

> 这里插入个题外话，大家在网络性能优化中经常听到的零拷贝，我觉得这是吹出来的一个大牛逼。TCP 为了保证可靠性，第二次的拷贝根本就没法省。如果包再大于 MTU 的话，分片时的拷贝同样也避免不了。

看到这里，相信内核发送数据包对于你来说，已经不再是一个完全不懂的黑盒了。本文哪怕你只看懂十分之一，你也已经掌握了这个黑盒的打开方式。这在你将来优化网络性能时你就会知道从哪儿下手了。

最后，还愣着干啥，赶紧帮飞哥**赞**、**再看**、**转发**三连走起！

参考

- Monitoring and Tuning the Linux Networking Stack: Sending Data
- 上述文章的一个翻译版本,只翻译了一半
- 翻译的是另外一半，neighboring subsystem 浅析
- LINUX内核网络数据包发送（四）——LINUX NETDEVICE 子系统
- LINUX内核网络数据包发送（三）——IP协议层分析
- LINUX网络子系统中DMA机制的实现
- Linux socket 数据发送过程深入分析
- send 函数
- Linux msghdr结构讲解
- TCP的发送系列 — tcp_sendmsg()的实现（一）
- 参考：send和recv背后数据的收发过程
- linux net子系统-协议层（传输层与网络层）
- LINUX内核网络（一）——初探内核网络
- XPS选择发送队列描述详细

公众号: 「开发内功修炼」

了解你的每一比特、 用好你的每一纳秒！

公众号

作者微信