图解Linux网络包接收过程

原创 张彦飞allen 开发内功修炼 2020-09-24 09:03

因为要对百万、千万、甚至是过亿的用户提供各种网络服务，所以在一线互联网企业里面试和晋升后端开发同学的其中一个重点要求就是要能支撑高并发，要理解性能开销，会进行性能优化。而很多时候，如果你对Linux底层的理解不深的话，遇到很多线上性能瓶颈你会觉得狗拿刺猬，无从下手。

我们今天用图解的方式，来深度理解一下在Linux下网络包的接收过程。还是按照惯例来借用一段最简单的代码开始思考。为了简单起见，我们用udp来举例，如下：

```
int main(){
      int serverSocketFd = socket(AF_INET, SOCK_DGRAM, 0);
      bind(serverSocketFd, ...);

      char buff[BUFFSIZE];
      int readCount = recvfrom(serverSocketFd, buff, BUFFSIZE, 0, ...);
      buff[readCount] = '\0';
      printf("Receive from client:%s\n", buff);
}
```

上面代码是一段udp server接收收据的逻辑。当在开发视角看的时候，只要客户端有对应的数据发送过来，服务器端执行 `recv_from` 后就能收到它，并把它打印出来。我们现在想知道的是，当网络包达到网卡，直到我们的 `recvfrom` 收到数据，这中间，究竟都发生过什么？

通过本文，你将深入理解Linux网络系统内部是如何实现的，以及各个部分之间如何交互。相信这对你的工作将会有非常大的帮助。本文基于Linux 3.10，源代码参见https://mirrors.edge.kernel.org/pub/linux/kernel/v3.x/，网卡驱动采用Intel的igb网卡举例。

友情提示，本文略长，可以先Mark后看！

一 Linux网络收包总览

在TCP/IP网络分层模型里，整个协议栈被分成了物理层、链路层、网络层，传输层和应用层。物理层对应的是网卡和网线，应用层对应的是我们常见的Nginx，FTP等等各种应用。Linux实现的是链路层、网络层和传输层这三层。

在Linux内核实现中，链路层协议靠网卡驱动来实现，内核协议栈来实现网络层和传输层。内核对更上层的应用层提供socket接口来供用户进程访问。我们用Linux的视角来看到的TCP/IP网络分层模型应该是下面这个样子的。

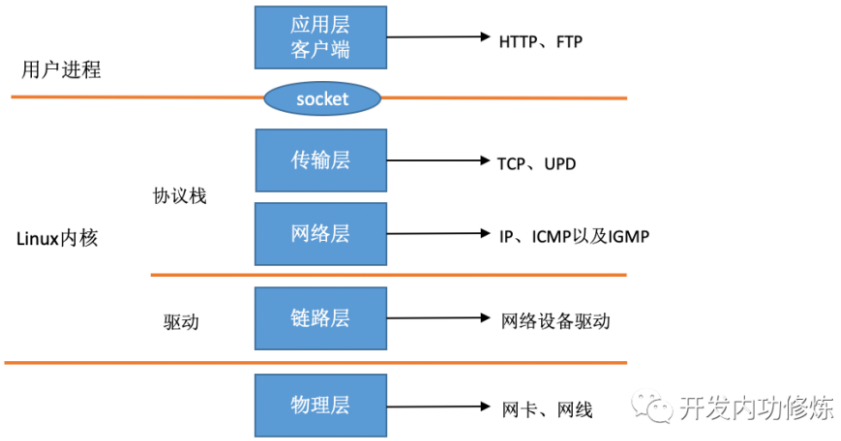

图1 Linux视角的网络协议栈

在Linux的源代码中，网络设备驱动对应的逻辑位于`driver/net/ethernet`，其中intel系列网卡的驱动在`driver/net/ethernet/intel`目录下。协议栈模块代码位于`kernel`和`net`目录。

内核和网络设备驱动是通过中断的方式来处理的。当设备上有数据到达的时候，会给CPU的相关引脚上触发一个电压变化，以通知CPU来处理数据。对于网络模块来说，由于处理过程比较复杂和耗时，如果在中断函数中完成所有的处理，将会导致中断处理函数（优先级过高）将过度占据CPU，将导致CPU无法响应其它设备，例如鼠标和键盘的消息。因此Linux中断处理函数是分上半部和下半部的。上半部是只进行最简单的工作，快速处理然后释放CPU，接着CPU就可以允许其它中断进来。剩下将绝大部分的工作都放到下半部中，可以慢慢从容处理。2.4以后的内核版本采用的下半部实现方式是软中断，由ksoftirqd内核线程全权处理。和硬中断不同的是，硬中断是通过给CPU物理引脚施加电压变化，而软中断是通过给内存中的一个变量的二进制值以通知软中断处理程序。

好了，大概了解了网卡驱动、硬中断、软中断和ksoftirqd线程之后，我们在这几个概念的基础上给出一个内核收包的路径示意：

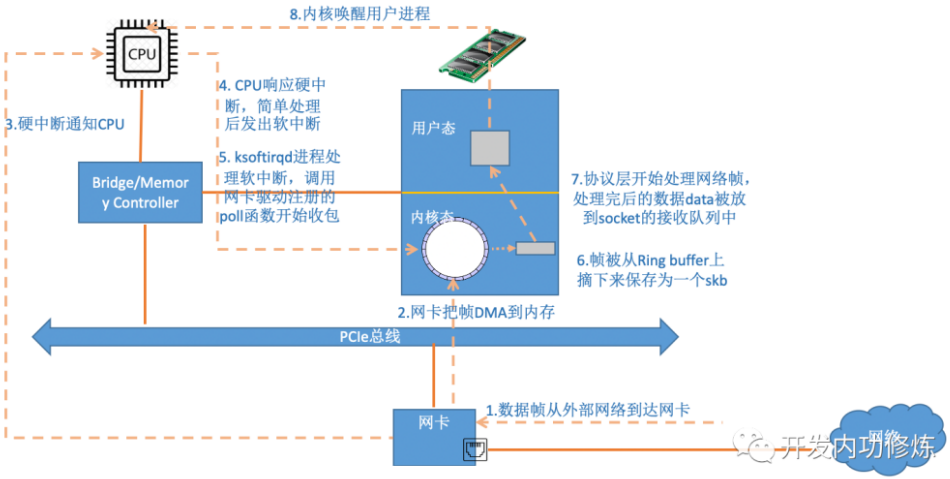

图2 Linux内核网络收包总览

当网卡上收到数据以后，Linux中第一个工作的模块是网络驱动。网络驱动会以DMA的方式把网卡上收到的帧写到内存里。再向CPU发起一个中断，以通知CPU有数据到达。第二，当CPU收到中断请求后，会去调用网络驱动注册的中断处理函数。网卡的中断处理函数并不做过多工作，发出软中断请求，然后尽快释放CPU。ksoftirqd检测到有软中断请求到达，调用poll开始轮询收包，收到后交由各级协议栈处理。对于UDP包来说，会被放到用户socket的接收队列中。

我们从上面这张图中已经从整体上把握到了Linux对数据包的处理过程。但是要想了解更多网络模块工作的细节，我们还得往下看。

二 Linux启动

Linux驱动，内核协议栈等等模块在具备接收网卡数据包之前，要做很多的准备工作才行。比如要提前创建好ksoftirqd内核线程，要注册好各个协议对应的处理函数，网络设备子系统要提前初始化好，网卡要启动好。只有这些都Ready之后，我们才能真正开始接收数据包。那么我们现在来看看这些准备工作都是怎么做的。

2.1 创建ksoftirqd内核线程

Linux的软中断都是在专门的内核线程（ksoftirqd）中进行的，因此我们非常有必要看一下这些进程是怎么初始化的，这样我们才能在后面更准确地了解收包过程。该进程数量不是1个，而是N个，其中N等于你的机器的核数。

系统初始化的时候在kernel/smpboot.c中调用了smpboot_register_percpu_thread， 该函数进一步会执行到spawn_ksoftirqd（位于kernel/softirq.c）来创建出softirqd进程。

图3 创建ksoftirqd内核线程

相关代码如下：

```
//file: kernel/softirq.c
static struct smp_hotplug_thread softirq_threads = {
      .store              = &ksoftirqd,
      .thread_should_run    = ksoftirqd_should_run,
      .thread_fn         = run_ksoftirqd,
      .thread_comm            = "ksoftirqd/%u",};
```

```
static __init int spawn_ksoftirqd(void){
      register_cpu_notifier(&cpu_nfb);

      BUG_ON(smpboot_register_percpu_thread(&softirq_threads));
       return 0;
}
early_initcall(spawn_ksoftirqd);
```

当ksoftirqd被创建出来以后，它就会进入自己的线程循环函数ksoftirqd_should_run和run_ksoftirqd了。不停地判断有没有软中断需要被处理。这里需要注意的一点是，软中断不仅仅只有网络软中断，还有其它类型。

```
//file: include/linux/interrupt.h
```

```
enum{
      HI_SOFTIRQ=0,
```

```
        TIMER_SOFTIRQ,
        NET_TX_SOFTIRQ,
        NET_RX_SOFTIRQ,
        BLOCK_SOFTIRQ,
        BLOCK_IOPOLL_SOFTIRQ,
        TASKLET_SOFTIRQ,
        SCHED_SOFTIRQ,
        HRTIMER_SOFTIRQ,
        RCU_SOFTIRQ,
};
```

2.2 网络子系统初始化

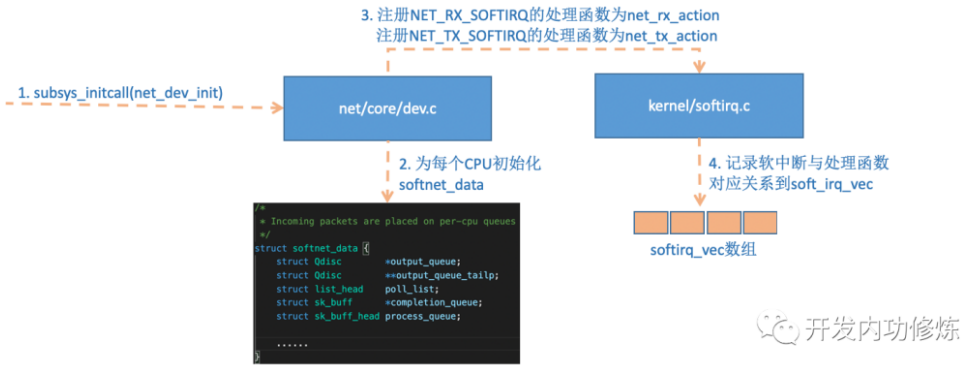

图4 网络子系统初始化

linux内核通过调用 `subsys_initcall` 来初始化各个子系统，在源代码目录里你可以grep出许多对这个函数的调用。这里我们要说的是网络子系统的初始化，会执行到 `net_dev_init` 函数。

```
//file: net/core/dev.c
static int __init net_dev_init(void){
      ......

      for_each_possible_cpu(i) {
```

```
            struct softnet_data *sd = &per_cpu(softnet_data, i);

            memset(sd, 0, sizeof(*sd));
            skb_queue_head_init(&sd->input_pkt_queue);
            skb_queue_head_init(&sd->process_queue);
            sd->completion_queue = NULL;
            INIT_LIST_HEAD(&sd->poll_list);
            ......
      }
      ......
      open_softirq(NET_TX_SOFTIRQ, net_tx_action);
      open_softirq(NET_RX_SOFTIRQ, net_rx_action);
}
subsys_initcall(net_dev_init);
```

在这个函数里，会为每个CPU都申请一个 `softnet_data` 数据结构，在这个数据结构里的 `poll_list` 是等待驱动程序将其poll函数注册进来，稍后网卡驱动初始化的时候我们可以看到这一过程。

另外open_softirq注册了每一种软中断都注册一个处理函数。NET_TX_SOFTIRQ的处理函数为net_tx_action，NET_RX_SOFTIRQ的为net_rx_action。继续跟踪 `open_softirq` 后发现这个注册的方式是记录在 `softirq_vec` 变量里的。后面ksoftirqd线程收到软中断的时候，也会使用这个变量来找到每一种软中断对应的处理函数。

```
//file: kernel/softirq.c
void open_softirq(int nr, void (*action)(struct softirq_action *)){
      softirq_vec[nr].action = action;
}
```

2.3 协议栈注册

内核实现了网络层的ip协议，也实现了传输层的tcp协议和udp协议。这些协议对应的实现函数分别是ip_rcv(),tcp_v4_rcv()和udp_rcv()。和我们平时写代码的方式不一样的是，内核是通过注册的方式来实现的。Linux内核中的 `fs_initcall` 和 `subsys_initcall` 类似，也是初始化模块的入口。 `fs_initcall` 调用 `inet_init` 后开始网络协议栈注册。通过 `inet_init` ，将这些函数注册到了inet_protos和ptype_base数据结构中了。如下图:

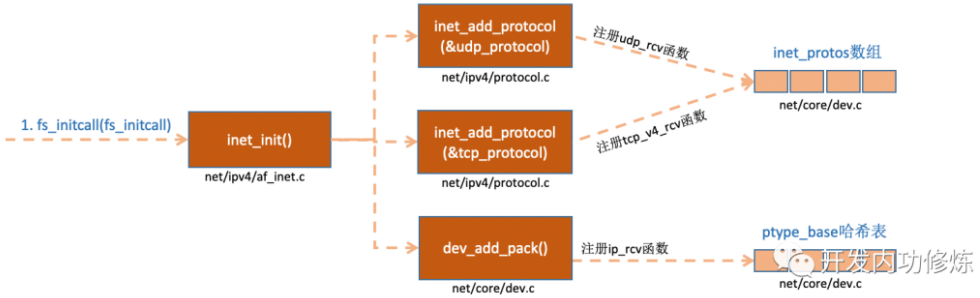

图5 AF_INET协议栈注册

相关代码如下

```
//file: net/ipv4/af_inet.c
static struct packet_type ip_packet_type __read_mostly = {
      .type = cpu_to_be16(ETH_P_IP),
      .func = ip_rcv,};static const struct net_protocol udp_protocol = {
      .handler =   udp_rcv,
      .err_handler =   udp_err,
      .no_policy =      1,
      .netns_ok = 1,};static const struct net_protocol tcp_protocol = {
      .early_demux      =    tcp_v4_early_demux,
      .handler      =    tcp_v4_rcv,
      .err_handler      =    tcp_v4_err,
      .no_policy    =    1,
      .netns_ok     =    1,
};
static int __init inet_init(void){
      ......
      if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
            pr_crit("%s: Cannot add ICMP protocol\n", __func__);
      if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
            pr_crit("%s: Cannot add UDP protocol\n", __func__);
      if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
```

```
            pr_crit("%s: Cannot add TCP protocol\n", __func__);
      ......
      dev_add_pack(&ip_packet_type);
}
```

上面的代码中我们可以看到，udp_protocol结构体中的handler是udp_rcv，tcp_protocol结构体中的handler是tcp_v4_rcv，通过inet_add_protocol被初始化了进来。

```
int inet_add_protocol(const struct net_protocol *prot, unsigned char protocol){
      if (!prot->netns_ok) {
            pr_err("Protocol %u is not namespace aware, cannot register.\n",
                  protocol);
            return -EINVAL;
      }

      return !cmpxchg((const struct net_protocol **)&inet_protos[protocol],
                  NULL, prot) ? 0 : -1;
}
```

`inet_add_protocol` 函数将tcp和udp对应的处理函数都注册到了inet_protos数组中了。再看 `dev_add_pack(&ip_packet_type);` 这一行，ip_packet_type结构体中的type是协议名，func是ip_rcv函数，在dev_add_pack中会被注册到ptype_base哈希表中。

```
//file: net/core/dev.c
void dev_add_pack(struct packet_type *pt){
      struct list_head *head = ptype_head(pt);
      ......
}
static inline struct list_head *ptype_head(const struct packet_type *pt){
      if (pt->type == htons(ETH_P_ALL))
            return &ptype_all;
      else
            return &ptype_base[ntohs(pt->type) & PTYPE_HASH_MASK];
```

```
}
```

这里我们需要记住inet_protos记录着udp，tcp的处理函数地址，ptype_base存储着ip_rcv()函数的处理地址。后面我们会看到软中断中会通过ptype_base找到ip_rcv函数地址，进而将ip包正确地送到ip_rcv()中执行。在ip_rcv中将会通过inet_protos找到tcp或者udp的处理函数，再而把包转发给udp_rcv()或tcp_v4_rcv()函数。

扩展一下，如果看一下ip_rcv和udp_rcv等函数的代码能看到很多协议的处理过程。例如，ip_rcv中会处理netfilter和iptable过滤，如果你有很多或者很复杂的 netfilter 或 iptables 规则，这些规则都是在软中断的上下文中执行的，会加大网络延迟。再例如，udp_rcv中会判断socket接收队列是否满了。对应的相关内核参数是net.core.rmem_max和net.core.rmem_default。如果有兴趣，建议大家好好读一下 `inet_init` 这个函数的代码。

2.4 网卡驱动初始化

每一个驱动程序（不仅仅只是网卡驱动）会使用 module_init 向内核注册一个初始化函数，当驱动被加载时，内核会调用这个函数。比如igb网卡驱动的代码位于 `drivers/net/ethernet/intel/igb/igb_main.c`

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static struct pci_driver igb_driver = {
      .name       = igb_driver_name,
      .id_table = igb_pci_tbl,
      .probe      = igb_probe,
      .remove     = igb_remove,
      ......
};
static int __init igb_init_module(void){
      ......
      ret = pci_register_driver(&igb_driver);
      return ret;
}
```

驱动的 `pci_register_driver` 调用完成后，Linux内核就知道了该驱动的相关信息，比如igb网卡驱动的 `igb_driver_name` 和 `igb_probe` 函数地址等等。当网卡设备被识别以后，内核会调用其驱动的probe方法（igb_driver的probe方法是igb_probe）。驱动probe方法执行的目的就是让设备ready，对于igb网

卡，其 igb_probe 位于drivers/net/ethernet/intel/igb/igb_main.c下。主要执行的操作如下：

图6 网卡驱动初始化

第5步中我们看到，网卡驱动实现了ethtool所需要的接口，也在这里注册完成函数地址的注册。当 ethtool 发起一个系统调用之后，内核会找到对应操作的回调函数。对于igb网卡来说，其实现函数都在drivers/net/ethernet/intel/igb/igb_ethtool.c下。相信你这次能彻底理解ethtool的工作原理了吧？这个命令之所以能查看网卡收发包统计、能修改网卡自适应模式、能调整RX 队列的数量和大小，是因为ethtool命令最终调用到了网卡驱动的相应方法，而不是ethtool本身有这个超能力。

第6步注册的igb_netdev_ops中包含的是igb_open等函数，该函数在网卡被启动的时候会被调用。

```
//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,
   .ndo_get_stats64            = igb_get_stats64,
   .ndo_set_rx_mode            = igb_set_rx_mode,
   .ndo_set_mac_address      = igb_set_mac,
   .ndo_change_mtu             = igb_change_mtu,
   .ndo_do_ioctl               = igb_ioctl,
  ......
```

第7步中，在igb_probe初始化过程中，还调用到了 igb_alloc_q_vector 。他注册了一个NAPI机制所必须的poll函数，对于igb网卡驱动来说，这个函数就是igb_poll,如下代码所示。

```
static int igb_alloc_q_vector(struct igb_adapter *adapter,
                              int v_count, int v_idx,
                              int txr_count, int txr_idx,
                              int rxr_count, int rxr_idx){
      ......
      /* initialize NAPI */
      netif_napi_add(adapter->netdev, &q_vector->napi,
                     igb_poll, 64);
}
```

2.5 启动网卡

当上面的初始化都完成以后，就可以启动网卡了。回忆前面网卡驱动初始化时，我们提到了驱动向内核注册了 structure net_device_ops 变量，它包含着网卡启用、发包、设置mac 地址等回调函数（函数指针）。当启用一个网卡时（例如，通过 ifconfig eth0 up），net_device_ops 中的 igb_open方法会被调用。它通常会做以下事情：

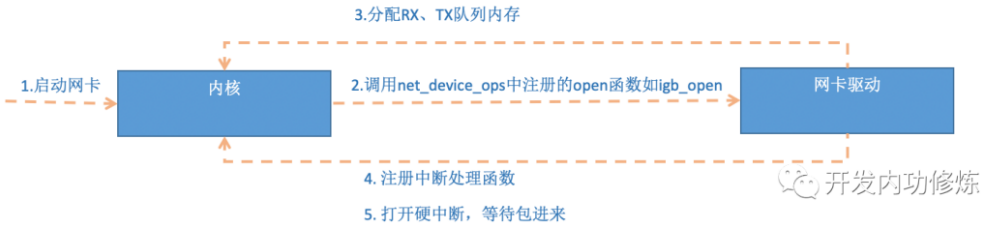

图7 启动网卡

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

      /* allocate receive descriptors */
      err = igb_setup_all_rx_resources(adapter);
```

```
	/* 注册中断处理函数 */
	err = igb_request_irq(adapter);
	if (err)
		goto err_req_irq;

	/* 启用NAPI */
	for (i = 0; i < adapter->num_q_vectors; i++)
		napi_enable(&(adapter->q_vector[i]->napi));
	......
}
```

在上面 `__igb_open` 函数调用了igb_setup_all_tx_resources,和igb_setup_all_rx_resources。在 `igb_setup_all_rx_resources` 这一步操作中，分配了RingBuffer，并建立内存和Rx队列的映射关系。（Rx Tx 队列的数量和大小可以通过 ethtool 进行配置）。我们再接着看中断函数注册 `igb_request_irq`：

```
static int igb_request_irq(struct igb_adapter *adapter){
	if (adapter->msix_entries) {
		err = igb_request_msix(adapter);
		if (!err)
			goto request_done;
		......
	}
}
static int igb_request_msix(struct igb_adapter *adapter){
	......
	for (i = 0; i < adapter->num_q_vectors; i++) {
		...
		err = request_irq(adapter->msix_entries[vector].vector,
				  igb_msix_ring, 0, q_vector->name,
	}
```

在上面的代码中跟踪函数调用， `__igb_open` => `igb_request_irq` => `igb_request_msix`，在 `igb_request_msix` 中我们看到了，对于多队列的网卡，为每一个队列都注册了中断，其对应的中断处理函数是igb_msix_ring（该函数也在drivers/net/ethernet/intel/igb/igb_main.c下）。我们也可以

看到，msix方式下，每个 RX 队列有独立的MSI-X 中断，从网卡硬件中断的层面就可以设置让收到的包被不同的 CPU处理。（可以通过 irqbalance ，或者修改 /proc/irq/IRQ_NUMBER/smp_affinity能够修改和CPU的绑定行为）。

当做好以上准备工作以后，就可以开门迎客（数据包）了！

三 迎接数据的到来

3.1 硬中断处理

首先当数据帧从网线到达网卡上的时候，第一站是网卡的接收队列。网卡在分配给自己的RingBuffer中寻找可用的内存位置，找到后DMA引擎会把数据DMA到网卡之前关联的内存里，这个时候CPU都是无感的。当DMA操作完成以后，网卡会像CPU发起一个硬中断，通知CPU有数据到达。

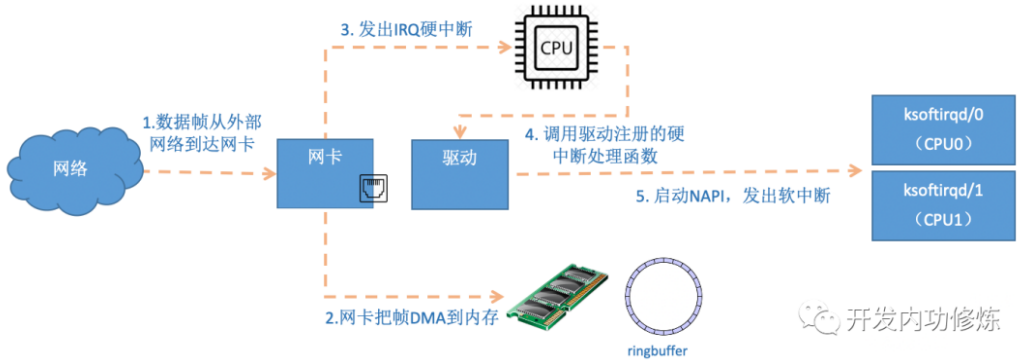

图8 网卡数据硬中断处理过程

> 注意：当RingBuffer满的时候，新来的数据包将给丢弃。ifconfig查看网卡的时候，可以里面有个overruns，表示因为环形队列满被丢弃的包。如果发现有丢包，可能需要通过ethtool命令来加大环形队列的长度。

在启动网卡一节，我们说到了网卡的硬中断注册的处理函数是igb_msix_ring。

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static irqreturn_t igb_msix_ring(int irq, void *data){
      struct igb_q_vector *q_vector = data;
```

```
        /* Write the ITR value calculated from the previous interrupt. */
        igb_write_itr(q_vector);

        napi_schedule(&q_vector->napi);
        return IRQ_HANDLED;
}
```

`igb_write_itr` 只是记录一下硬件中断频率（据说目的是在减少对CPU的中断频率时用到）。顺着napi_schedule调用一路跟踪下去，`__napi_schedule` => `____napi_schedule`

```
/* Called with irq disabled */
static inline void ____napi_schedule(struct softnet_data *sd,
                                      struct napi_struct *napi){
        list_add_tail(&napi->poll_list, &sd->poll_list);
        __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

这里我们看到，`list_add_tail` 修改了CPU变量softnet_data里的poll_list，将驱动napi_struct传过来的poll_list添加了进来。其中softnet_data中的poll_list是一个双向列表，其中的设备都带有输入帧等着被处理。紧接着 `__raise_softirq_irqoff` 触发了一个软中断NET_RX_SOFTIRQ， 这个所谓的触发过程只是对一个变量进行了一次或运算而已。

```
void __raise_softirq_irqoff(unsigned int nr){
        trace_softirq_raise(nr);
        or_softirq_pending(1UL << nr);
}
//file: include/linux/irq_cpustat.h
#define or_softirq_pending(x)   (local_softirq_pending() |= (x))
```

我们说过，Linux在硬中断里只完成简单必要的工作，剩下的大部分的处理都是转交给软中断的。通过上面代码可以看到，硬中断处理过程真的是非常短。只是记录了一个寄存器，修改了一下下CPU的poll_list，然后发出个软中断。就这么简单，硬中断工作就算是完成了。

3.2 ksoftirqd内核线程处理软中断

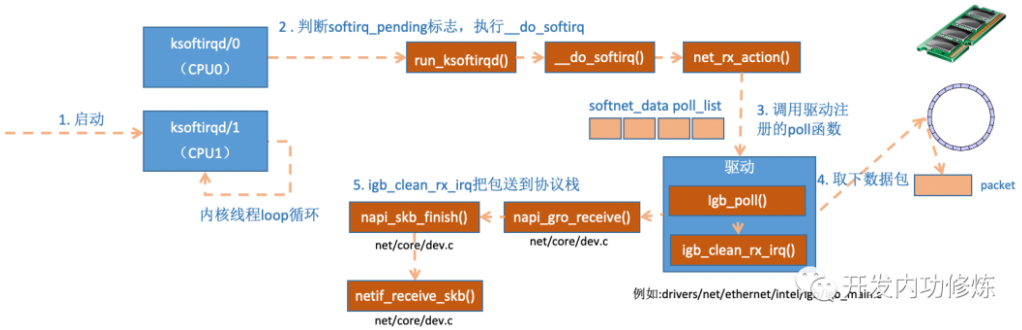

图9 ksoftirqd内核线程

内核线程初始化的时候，我们介绍了ksoftirqd中两个线程函数`ksoftirqd_should_run`和`run_ksoftirqd`。其中`ksoftirqd_should_run`代码如下：

```
static int ksoftirqd_should_run(unsigned int cpu){
      return local_softirq_pending();
}
#define local_softirq_pending() \      __IRQ_STAT(smp_processor_id(), __softirq_pending)
```

这里看到和硬中断中调用了同一个函数`local_softirq_pending`。使用方式不同的是硬中断位置是为了写入标记，这里仅仅只是读取。如果硬中断中设置了`NET_RX_SOFTIRQ`,这里自然能读取的到。接下来会真正进入线程函数中`run_ksoftirqd`处理：

```
static void run_ksoftirqd(unsigned int cpu){
      local_irq_disable();
      if (local_softirq_pending()) {
            __do_softirq();
            rcu_note_context_switch(cpu);
            local_irq_enable();
            cond_resched();
            return;
      }
      local_irq_enable();
```

```
}
```

在 `__do_softirq` 中，判断根据当前CPU的软中断类型，调用其注册的action方法。

```
asmlinkage void __do_softirq(void){
      do {
            if (pending & 1) {
                  unsigned int vec_nr = h - softirq_vec;
                  int prev_count = preempt_count();
                  ...
                  trace_softirq_entry(vec_nr);
                  h->action(h);
                  trace_softirq_exit(vec_nr);
                  ...
            }
            h++;
            pending >>= 1;
      } while (pending);
}
```

在网络子系统初始化小节， 我们看到我们为NET_RX_SOFTIRQ注册了处理函数net_rx_action。所以 `net_rx_action` 函数就会被执行到了。

这里需要注意一个细节，硬中断中设置软中断标记，和ksoftirq的判断是否有软中断到达，都是基于smp_processor_id()的。这意味着只要硬中断在哪个CPU上被响应，那么软中断也是在这个CPU上处理的。所以说，如果你发现你的Linux软中断CPU消耗都集中在一个核上的话，做法是要把调整硬中断的CPU亲和性，来将硬中断打散到不同的CPU核上去。

我们再来把精力集中到这个核心函数 `net_rx_action` 上来。

```
static void net_rx_action(struct softirq_action *h){
      struct softnet_data *sd = &__get_cpu_var(softnet_data);
      unsigned long time_limit = jiffies + 2;
      int budget = netdev_budget;
```

```
        void *have;

        local_irq_disable();
        while (!list_empty(&sd->poll_list)) {
                ......
                n = list_first_entry(&sd->poll_list, struct napi_struct, poll_list);

                work = 0;
                if (test_bit(NAPI_STATE_SCHED, &n->state)) {
                        work = n->poll(n, weight);
                        trace_napi_poll(n);
                }
                budget -= work;
        }
}
```

函数开头的time_limit和budget是用来控制net_rx_action函数主动退出的，目的是保证网络包的接收不霸占CPU不放。等下次网卡再有硬中断过来的时候再处理剩下的接收数据包。其中budget可以通过内核参数调整。这个函数中剩下的核心逻辑是获取到当前CPU变量softnet_data，对其poll_list进行遍历，然后执行到网卡驱动注册到的poll函数。对于igb网卡来说，就是igb驱动力的 `igb_poll` 函数了。

```
static int igb_poll(struct napi_struct *napi, int budget){

        ...
        if (q_vector->tx.ring)
                clean_complete = igb_clean_tx_irq(q_vector);

        if (q_vector->rx.ring)
                clean_complete &= igb_clean_rx_irq(q_vector, budget);
        ...
}
```

在读取操作中， `igb_poll` 的重点工作是对 `igb_clean_rx_irq` 的调用。

```
static bool igb_clean_rx_irq(struct igb_q_vector *q_vector, const int budget){
      ...
      do {
            /* retrieve a buffer from the ring */
            skb = igb_fetch_rx_buffer(rx_ring, rx_desc, skb);

            /* fetch next buffer in frame if non-eop */
            if (igb_is_non_eop(rx_ring, rx_desc))
                  continue;
            }

            /* verify the packet layout is correct */
            if (igb_cleanup_headers(rx_ring, rx_desc, skb)) {
                  skb = NULL;
                  continue;
            }

            /* populate checksum, timestamp, VLAN, and protocol */
            igb_process_skb_fields(rx_ring, rx_desc, skb);

            napi_gro_receive(&q_vector->napi, skb);

}
```

`igb_fetch_rx_buffer` 和 `igb_is_non_eop` 的作用就是把数据帧从RingBuffer上取下来。为什么需要两个函数呢？因为有可能帧要占多多个RingBuffer，所以是在一个循环中获取的，直到帧尾部。获取下来的一个数据帧用一个sk_buff来表示。收取完数据以后，对其进行一些校验，然后开始设置sbk变量的timestamp, VLAN id, protocol等字段。接下来进入到napi_gro_receive中：

```
//file: net/core/dev.c
gro_result_t napi_gro_receive(struct napi_struct *napi, struct sk_buff *skb){
      skb_gro_reset_offset(skb);
      return napi_skb_finish(dev_gro_receive(napi, skb), skb);
```

```
}
```

`dev_gro_receive`这个函数代表的是网卡GRO特性，可以简单理解成把相关的小包合并成一个大包就行，目的是减少传送给网络栈的包数，这有助于减少 CPU 的使用量。我们暂且忽略，直接看`napi_skb_finish`，这个函数主要就是调用了`netif_receive_skb`。

```
//file: net/core/dev.c
static gro_result_t napi_skb_finish(gro_result_t ret, struct sk_buff *skb){
      switch (ret) {
      case GRO_NORMAL:
            if (netif_receive_skb(skb))
                  ret = GRO_DROP;
            break;
      ......
}
```

在`netif_receive_skb`中，数据包将被送到协议栈中。声明，以下的3.3， 3.4， 3.5也都属于软中断的处理过程，只不过由于篇幅太长，单独拿出来成小节。

3.3 网络协议栈处理

`netif_receive_skb`函数会根据包的协议，假如是udp包，会将包依次送到ip_rcv(),udp_rcv()协议处理函数中进行处理。

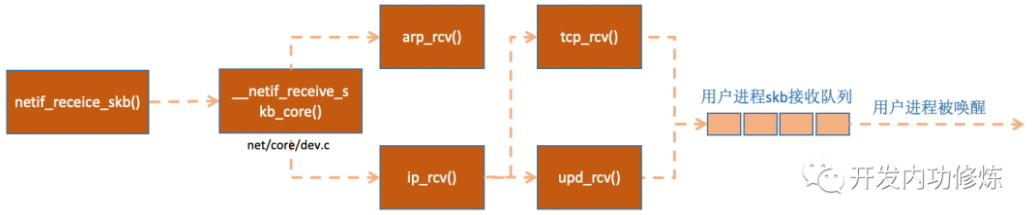

图10 网络协议栈处理

```
//file: net/core/dev.c
int netif_receive_skb(struct sk_buff *skb){
```

```
    //RPS处理逻辑，先忽略      ......
    return __netif_receive_skb(skb);
}
static int __netif_receive_skb(struct sk_buff *skb){
    ......
    ret = __netif_receive_skb_core(skb, false);}static int __netif_receive_skb_core(struct sk_buff *skb, bool pfmemalloc){
    ......

    //pcap逻辑，这里会将数据送入抓包点。tcpdump就是从这个入口获取包的      list_for_each_entry_rcu(ptype, &ptype_all, list) {
        if (!ptype->dev || ptype->dev == skb->dev) {
            if (pt_prev)
                ret = deliver_skb(skb, pt_prev, orig_dev);
            pt_prev = ptype;
        }
    }
    ......
    list_for_each_entry_rcu(ptype,
            &ptype_base[ntohs(type) & PTYPE_HASH_MASK], list) {
        if (ptype->type == type &&
            (ptype->dev == null_or_dev || ptype->dev == skb->dev ||
             ptype->dev == orig_dev)) {
            if (pt_prev)
                ret = deliver_skb(skb, pt_prev, orig_dev);
            pt_prev = ptype;
        }
    }
}
```

在 `__netif_receive_skb_core` 中，我看着原来经常使用的tcpdump的抓包点，很是激动，看来读一遍源代码时间真的没白浪费。接着 `__netif_receive_skb_core` 取出protocol，它会从数据包中取出协议信息，然后遍历注册在这个协议上的回调函数列表。`ptype_base` 是一个 hash table，在协议注册小节我们提到过。ip_rcv 函数地址就是存在这个 hash table中的。

```
//file: net/core/dev.c
```

```
static inline int deliver_skb(struct sk_buff *skb,
                          struct packet_type *pt_prev,
                          struct net_device *orig_dev){
      ......
      return pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
}
```

`pt_prev->func` 这一行就调用到了协议层注册的处理函数了。对于ip包来讲，就会进入到 `ip_rcv` （如果是arp包的话，会进入到arp_rcv）。

3.4 IP协议层处理

我们再来大致看一下linux在ip协议层都做了什么，包又是怎么样进一步被送到udp或tcp协议处理函数中的。

```
//file: net/ipv4/ip_input.c
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device *orig_dev){
      ......
      return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, skb, dev, NULL,
                     ip_rcv_finish);
}
```

这里 `NF_HOOK` 是一个钩子函数，当执行完注册的钩子后就会执行到最后一个参数指向的函数 `ip_rcv_finish` 。

```
static int ip_rcv_finish(struct sk_buff *skb){
      ......
      if (!skb_dst(skb)) {
            int err = ip_route_input_noref(skb, iph->daddr, iph->saddr,
                                           iph->tos, skb->dev);
            ...
      }
      ......
      return dst_input(skb);
```

```
}
```

跟踪 `ip_route_input_noref` 后看到它又调用了 `ip_route_input_mc` 。在 `ip_route_input_mc` 中，函数 `ip_local_deliver` 被赋值给了 `dst.input` ，如下：

```
//file: net/ipv4/route.c
static int ip_route_input_mc(struct sk_buff *skb, __be32 daddr, __be32 saddr,u8 tos, struct net_device *dev, int our){
      if (our) {
            rth->dst.input= ip_local_deliver;
            rth->rt_flags |= RTCF_LOCAL;
      }
}
```

所以回到 `ip_rcv_finish` 中的 `return dst_input(skb);` 。

```
/* Input packet from network to transport.   */
static inline int dst_input(struct sk_buff *skb){
      return skb_dst(skb)->input(skb);
}
```

`skb_dst(skb)->input` 调用的input方法就是路由子系统赋的ip_local_deliver。

```
//file: net/ipv4/ip_input.c
int ip_local_deliver(struct sk_buff *skb){
      /*      *   Reassemble IP fragments.       */
      if (ip_is_fragment(ip_hdr(skb))) {
            if (ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER))
                  return 0;
      }

      return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, skb, skb->dev, NULL,
```

```
                    ip_local_deliver_finish);
}
static int ip_local_deliver_finish(struct sk_buff *skb){
      ......
      int protocol = ip_hdr(skb)->protocol;
      const struct net_protocol *ipprot;

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

如协议注册小节看到inet_protos中保存着tcp_rcv()和udp_rcv()的函数地址。这里将会根据包中的协议类型选择进行分发,在这里skb包将会进一步被派送到更上层的协议中，udp和tcp。

3.5 UDP协议层处理

在协议注册小节的时候我们说过，udp协议的处理函数是 `udp_rcv` 。

```
//file: net/ipv4/udp.c
int udp_rcv(struct sk_buff *skb){
      return __udp4_lib_rcv(skb, &udp_table, IPPROTO_UDP);
}
int __udp4_lib_rcv(struct sk_buff *skb, struct udp_table *udptable,
                int proto){
      sk = __udp4_lib_lookup_skb(skb, uh->source, uh->dest, udptable);

      if (sk != NULL) {
            int ret = udp_queue_rcv_skb(sk, skb
      }
      icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0);
```

```
}
```

`__udp4_lib_lookup_skb` 是根据skb来寻找对应的socket，当找到以后将数据包放到socket的缓存队列里。如果没有找到，则发送一个目标不可达的icmp包。

```
//file: net/ipv4/udp.c
int udp_queue_rcv_skb(struct sock *sk, struct sk_buff *skb){
      ......
      if (sk_rcvqueues_full(sk, skb, sk->sk_rcvbuf))
            goto drop;

      rc = 0;

      ipv4_pktinfo_prepare(skb);
      bh_lock_sock(sk);
      if (!sock_owned_by_user(sk))
            rc = __udp_queue_rcv_skb(sk, skb);
      else if (sk_add_backlog(sk, skb, sk->sk_rcvbuf)) {
            bh_unlock_sock(sk);
            goto drop;
      }
      bh_unlock_sock(sk);
      return rc;
}
```

sock_owned_by_user判断的是用户是不是正在这个socker上进行系统调用（socket被占用），如果没有，那就可以直接放到socket的接收队列中。如果有，那就通过 `sk_add_backlog` 把数据包添加到backlog队列。当用户释放的socket的时候，内核会检查backlog队列，如果有数据再移动到接收队列中。

`sk_rcvqueues_full` 接收队列如果满了的话，将直接把包丢弃。接收队列大小受内核参数net.core.rmem_max和net.core.rmem_default影响。

四 recvfrom系统调用

花开两朵，各表一枝。上面我们说完了整个Linux内核对数据包的接收和处理过程，最后把数据包放到socket的接收队列中了。那么我们再回头看用户进程调用 `recvfrom` 后是发生了什么。我们在代码里调用的 `recvfrom` 是一个glibc的库函数，该函数在执行后会将用户进行陷入到内核态，进入到Linux实现的系统调用 `sys_recvfrom` 。在理解Linux对 `sys_revvfrom` 之前，我们先来简单看一下 `socket` 这个核心数据结构。这个数据结构太大了，我们只把对和我们今天主题相关的内容画出来，如下：

图11 socket内核数据机构

`socket` 数据结构中的 `const struct proto_ops` 对应的是协议的方法集合。每个协议都会实现不同的方法集，对于IPv4 Internet协议族来说，每种协议都有对应的处理方法，如下。对于udp来说，是通过 `inet_dgram_ops` 来定义的，其中注册了 `inet_recvmsg` 方法。

```
//file: net/ipv4/af_inet.c
const struct proto_ops inet_stream_ops = {
      ......
      .recvmsg          = inet_recvmsg,
      .mmap             = sock_no_mmap,
      ......
}
const struct proto_ops inet_dgram_ops = {
      ......
      .sendmsg          = inet_sendmsg,
      .recvmsg          = inet_recvmsg,
      ......
```

```
}
```

socket 数据结构中的另一个数据结构 struct sock *sk 是一个非常大，非常重要的子结构体。其中的 sk_prot 又定义了二级处理函数。对于UDP协议来说，会被设置成UDP协议实现的方法集 udp_prot 。

```
//file: net/ipv4/udp.c
struct proto udp_prot = {
      .name               = "UDP",
      .owner              = THIS_MODULE,
      .close              = udp_lib_close,
      .connect            = ip4_datagram_connect,
      ......
      .sendmsg            = udp_sendmsg,
      .recvmsg            = udp_recvmsg,
      .sendpage           = udp_sendpage,
      ......
}
```

看完了 socket 变量之后，我们再来看 sys_revvfrom 的实现过程。

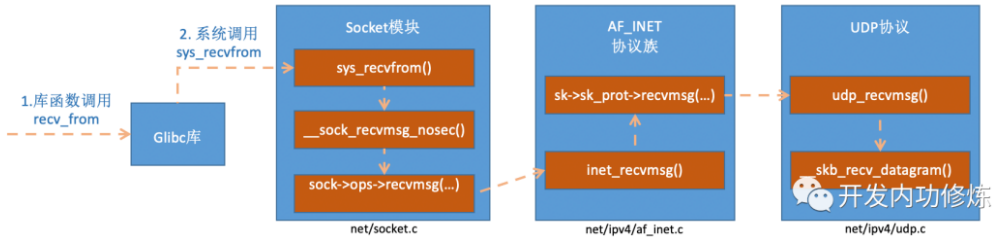

图12 recvfrom函数内部实现过程

在 inet_recvmsg 调用了 sk->sk_prot->recvmsg 。

```
//file: net/ipv4/af_inet.c
int inet_recvmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,size_t size, int flags){
```

```
      ......
      err = sk->sk_prot->recvmsg(iocb, sk, msg, size, flags & MSG_DONTWAIT,
                                 flags & ~MSG_DONTWAIT, &addr_len);
      if (err >= 0)
            msg->msg_namelen = addr_len;
      return err;
}
```

上面我们说过这个对于udp协议的socket来说，这个 `sk_prot` 就是 `net/ipv4/udp.c` 下的 `struct proto udp_prot` 。由此我们找到了 `udp_recvmsg` 方法。

```
//file:net/core/datagram.c:EXPORT_SYMBOL(__skb_recv_datagram);
struct sk_buff *__skb_recv_datagram(struct sock *sk, unsigned int flags,int *peeked, int *off, int *err){
      ......
      do {
            struct sk_buff_head *queue = &sk->sk_receive_queue;
            skb_queue_walk(queue, skb) {
                  ......
            }

            /* User doesn't want to wait */
            error = -EAGAIN;
            if (!timeo)
                  goto no_packet;
      } while (!wait_for_more_packets(sk, err, &timeo, last));
}
```

终于我们找到了我们想要看的重点，在上面我们看到了所谓的读取过程，就是访问 `sk->sk_receive_queue` 。如果没有数据，且用户也允许等待，则将调用wait_for_more_packets()执行等待操作，它加入会让用户进程进入睡眠状态。

五 总结

网络模块是Linux内核中最复杂的模块了，看起来一个简简单单的收包过程就涉及到许多内核组件之间的交互，如网卡驱动、协议栈，内核ksoftirqd线程等。看起来很复杂，本文想通过图示的方式，尽量以容易理解的方式来将内核收包过程讲清楚。现在让我们再串一串整个收包过程。

当用户执行完 `recvfrom` 调用后，用户进程就通过系统调用进行到内核态工作了。如果接收队列没有数据，进程就进入睡眠状态被操作系统挂起。这块相对比较简单，剩下大部分的戏份都是由Linux内核其它模块来表演了。

首先在开始收包之前，Linux要做许多的准备工作：

- 1. 创建ksoftirqd线程，为它设置好它自己的线程函数，后面指望着它来处理软中断呢
- 2. 协议栈注册，linux要实现许多协议，比如arp，icmp，ip，udp，tcp，每一个协议都会将自己的处理函数注册一下，方便包来了迅速找到对应的处理函数
- 3. 网卡驱动初始化，每个驱动都有一个初始化函数，内核会让驱动也初始化一下。在这个初始化过程中，把自己的DMA准备好，把NAPI的poll函数地址告诉内核
- 4. 启动网卡，分配RX，TX队列，注册中断对应的处理函数

以上是内核准备收包之前的重要工作，当上面都ready之后，就可以打开硬中断，等待数据包的到来了。

当数据到来了以后，第一个迎接它的是网卡（我去，这不是废话么）：

- 1. 网卡将数据帧DMA到内存的RingBuffer中，然后向CPU发起中断通知
- 2. CPU响应中断请求，调用网卡启动时注册的中断处理函数
- 3. 中断处理函数几乎没干啥，就发起了软中断请求
- 4. 内核线程ksoftirqd线程发现有软中断请求到来，先关闭硬中断
- 5. ksoftirqd线程开始调用驱动的poll函数收包
- 6. poll函数将收到的包送到协议栈注册的ip_rcv函数中
- 7. ip_rcv函数再讲包送到udp_rcv函数中（对于tcp包就送到tcp_rcv）

现在我们可以回到开篇的问题了，我们在用户层看到的简单一行 `recvfrom`,Linux内核要替我们做如此之多的工作，才能让我们顺利收到数据。这还是简简单单的UDP，如果是TCP，内核要做的工作更多，不由得感叹内核的开发者们真的是用心良苦。

理解了整个收包过程以后，我们就能明确知道Linux收一个包的CPU开销了。首先第一块是用户进程调用系统调用陷入内核态的开销。第二块是CPU响应包的硬中断的CPU开销。第三块是ksoftirqd内核线程的软中断上下文花费的。后面我们再专门发一篇文章实际观察一下这些开销。

另外网络收发中有很多末支细节咱们并没有展开了说，比如说no NAPI， GRO，RPS等。因为我觉得说的太对了反而会影响大家对整个流程的把握，所以尽量只保留主框架了，少即是多！