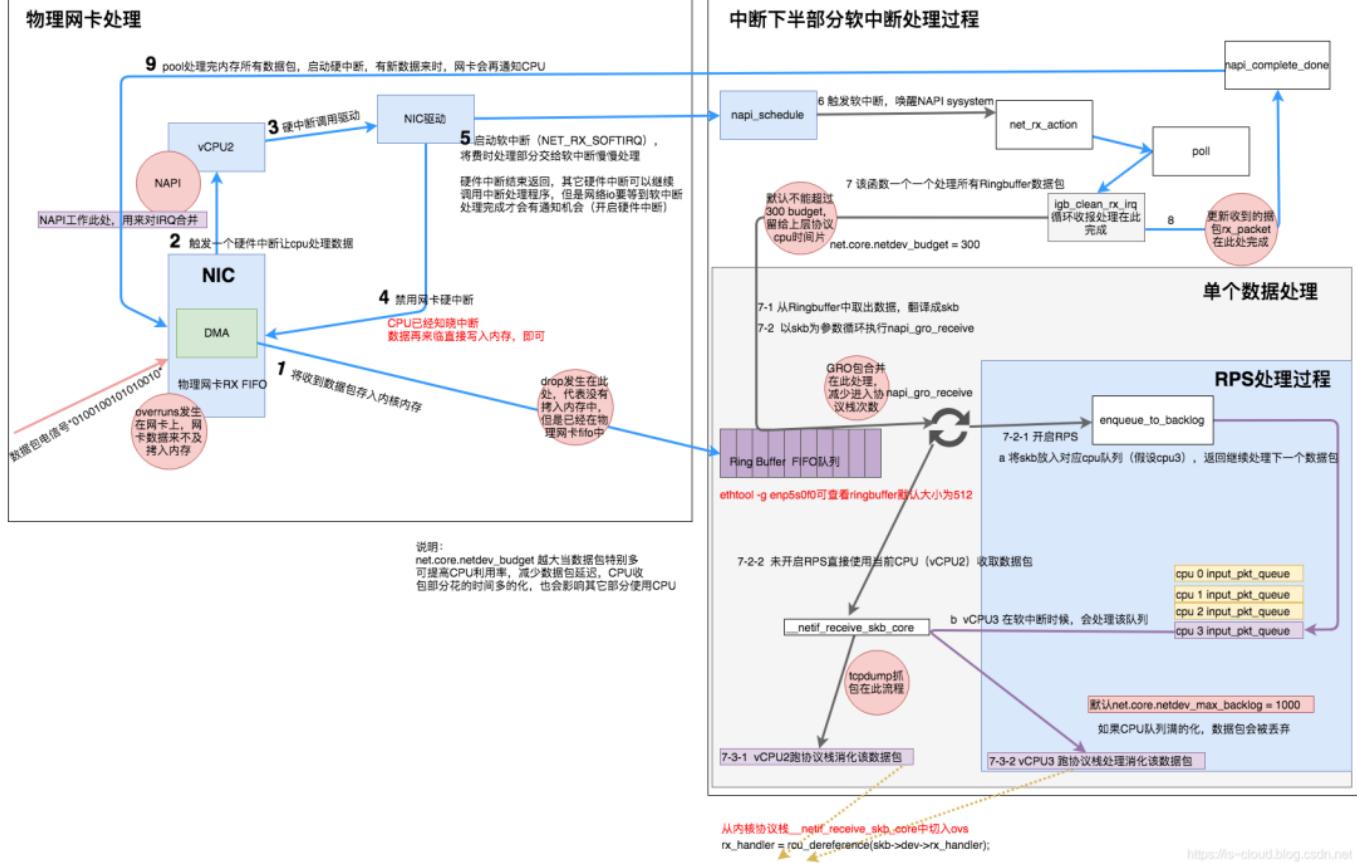


# Linux网络子系统

xholic 极客重生 2021-08-25 09:01

收录于话题

#深入理解网络 31 #深入理解Linux系统 29



今天分享一篇经典Linux协议栈文章，主要讲解Linux网络子系统，看完相信大家对协议栈又会加深不少，不光可以了解协议栈处理流程，方便定位问题，还可以学习一下怎么去设计一个可扩展的子系统，屏蔽不同层次的差异。

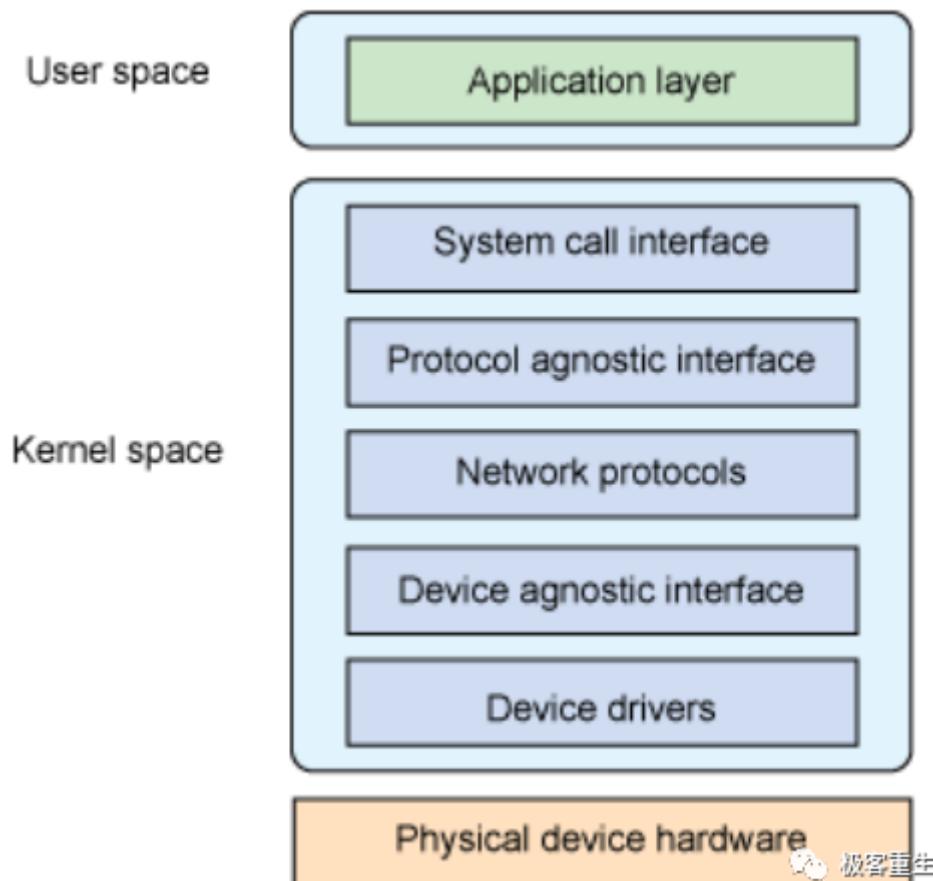
## 目录

1. Linux网络子系统的分层
2. TCP/IP分层模型
3. Linux 网络协议栈
4. Linux 网卡收包时的中断处理问题
5. Linux 网络启动的准备工作
6. Linux网络包：中断到网络层接收
7. 总结

## Linux网络子系统的分层

Linux网络子系统实现需要：

- 支持不同的协议族 (INET, INET6, UNIX, NETLINK...)
- 支持不同的网络设备
- 支持统一的BSD socket API
- 需要屏蔽协议、硬件、平台(API)的差异，因而采用分层结构：

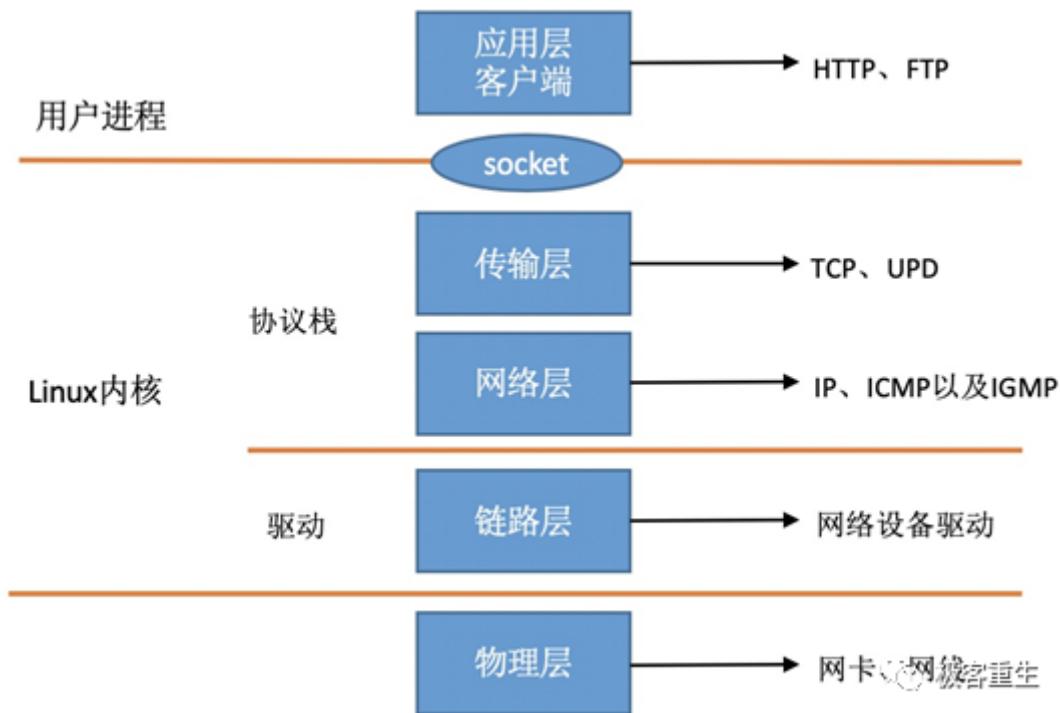


系统调用提供用户的应用程序访问内核的唯一途径。协议无关接口由socket layer来实现的，其提供一组通用功能，以支持各种不同的协议。网络协议层为socket层提供具体协议接口——proto{}，实现具体的协议细节。设备无关接口，提供一组通用函数供底层网络设备驱动程序使用。设备驱动与特定网卡设备相关，定义了具体的协议细节，会分配一个net\_device结构，然后用其必需的例程进行初始化。

## TCP/IP分层模型

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。



首先我们梳理一下每层模型的职责：

**链路层**：对0和1进行分组，定义数据帧，确认主机的物理地址，传输数据；

**网络层**：定义IP地址，确认主机所在的网络位置，并通过IP进行MAC寻址，对外网数据包进行路由转发；

**传输层**：定义端口，确认主机上应用程序的身份，并将数据包交给对应的应用程序；

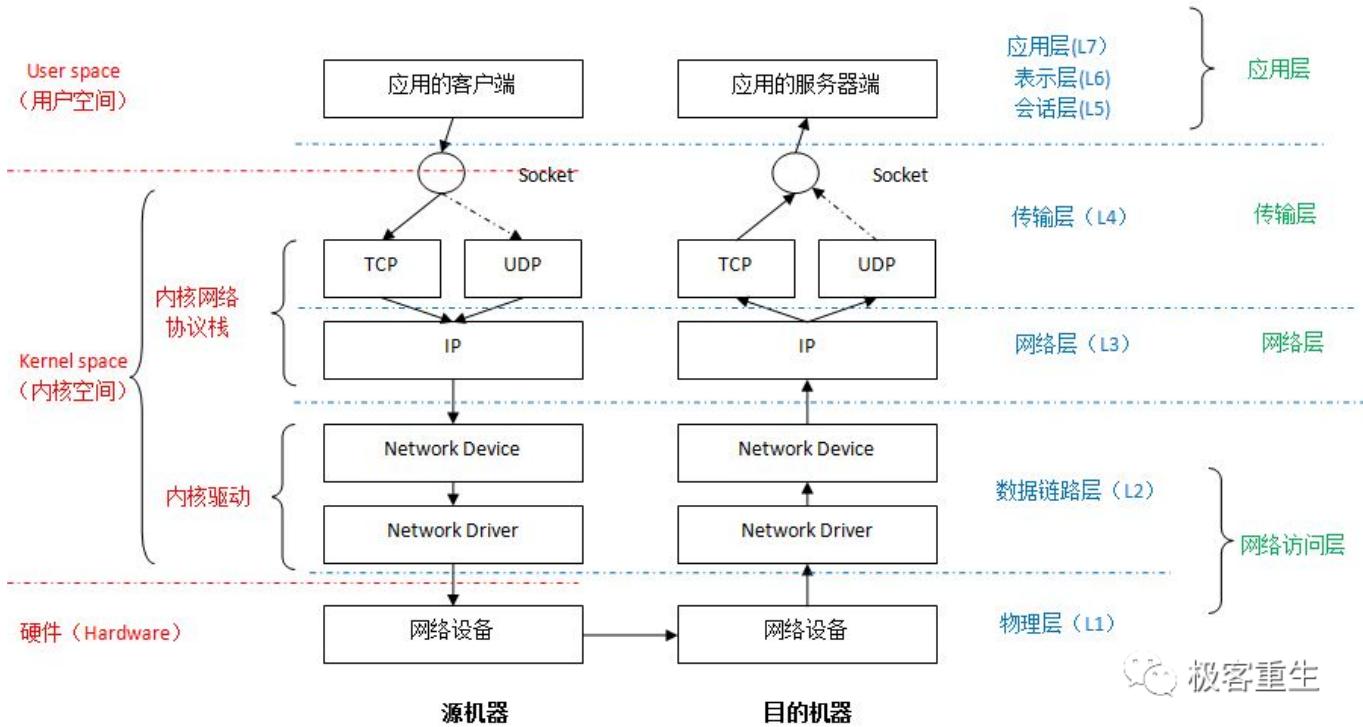
**应用层**：定义数据格式，并按照对应的格式解读数据。

然后再把每层模型的职责串联起来，用一句通俗易懂的话讲就是：

当你输入一个网址并按下回车键的时候，首先，应用层协议对该请求包做了格式定义；紧接着传输层协议加上了双方的端口号，确认了双方通信的应用程序；然后网络协议加上了双方的IP地址，确认了双方的网络位置；最后链路层协议加上了双方的MAC地址，确认了双方的物理位置，同时将数据进行分组，形成数据帧，采用广播方式，通过传输介质发送给对方主机。而对于不同网段，该数据包首先会转发给网关路由器，经过多次转发后，最终被发送到目标主机。目标机接收到数据包后，采用对应的协议，对帧数据进行组装，然后再通过一层一层的协议进行解析，最终被应用层的协议解析并交给服务器处理。

## Linux 网络协议栈

## 客户端发到服务器的 message 经过的完整路径



基于TCP/IP协议栈的send/recv在应用层，传输层，网络层和链路层中具体函数调用过程已经有很多人研究，本文引用一张比较完善的图如下：



以上说明基本大致说明了TCP/IP中TCP， UDP协议包在网络子系统中的实现流程。本文主要在链路层中，即关于网卡收报触发中断到进入网络层之间的过程探究。

## Linux 网卡收包时的中断处理问题

中断，一般指硬件中断，多由系统自身或与之链接的外设（如键盘、鼠标、网卡等）产生。中断首先是处理器提供的一种响应外设请求的机制，是处理器硬件支持的特性。一个外设通过产生一种电信号通知中断控制器，中断控制器再向处理器发送相应的信号。处理器检测到了这个信号后就会打断自己目前正在做的工作，转而去处理这次中断（所以才叫中断）。当然在转去处理中断和中断返回时都有保护现场和返回现场的操作，这里不赘述。

那软中断又是什么呢？我们知道在中断处理时CPU没法处理其它事物，对于网卡来说，如果每次网卡收包时中断的时间都过长，那很可能造成丢包的可能性。当然我们不能完全避免丢包的可能性，以太包的传输是没有100%保证的，所以网络才有协议栈，通过高层的协议来保证连续数据传输的数据完整性（比如在协议发现丢包时要求重传）。但是即使有协议保证，那我们也不能肆无忌惮的使用中断，中断的时间越短越好，尽快放开处理器，让它可以去响应下次中断甚至进行调度工作。**基于这样的考虑，我们将中断分成了上下两部分，上半部分就是上面说**

的中断部分，需要快速及时响应，同时需要越快结束越好。而下半部分就是完成一些可以推后执行的工作。对于网卡收包来说，网卡收到数据包，通知内核数据包到了，中断处理将数据包存入内存这些都是急切需要完成的工作，放到上半部完成。而解析处理数据包的工作则可以放到下半部去执行。

软中断就是下半部使用的一种机制，它通过软件模仿硬件中断的处理过程，但是和硬件没有关系，单纯的通过软件达到一种异步处理的方式。其它下半部的处理机制还包括tasklet，工作队列等。依据所处理的场合不同，选择不同的机制，网卡收包一般使用软中断。对应NET\_RX\_SOFTIRQ这个软中断，软中断的类型如下：

```

1 enum
2 {
3     HI_SOFTIRQ=0,
4     TIMER_SOFTIRQ,
5     NET_TX_SOFTIRQ,
6     NET_RX_SOFTIRQ,
7     BLOCK_SOFTIRQ,
8     IRQ_POLL_SOFTIRQ,
9     TASKLET_SOFTIRQ,
10    SCHED_SOFTIRQ,
11    HRTIMER_SOFTIRQ,
12    RCU_SOFTIRQ,      /* Preferable RCU should always be the last soft
13    NR_SOFTIRQS
14 };

```

通过以上可以了解到，Linux中断注册显然应该包括网卡的硬中断，包处理的软中断两个步骤。

## 注册网卡中断

我们以一个具体的网卡驱动为例，比如e1000。其模块初始化函数就是：

```

1 static int __init e1000_init_module(void)
2 {
3     int ret;
4     pr_info("%s - version %s\n", e1000_driver_string, e1000_driver_v
5     pr_info("%s\n", e1000_copyright);
6     ret = pci_register_driver(&e1000_driver);

```

```

7 ...
8     return ret;
9
10 }

```

其中e1000\_driver这个结构体是一个关键，这个结构体中很主要的一个方法就是.probe方法，也就是e1000\_probe()：

```

1 /**
2
3 * e1000_probe - Device Initialization Routine
4 * @pdev: PCI device information struct
5 * @ent: entry in e1000_pci_tbl
6 *
7 * Returns 0 on success, negative on failure
8 *
9 * e1000_probe initializes an adapter identified by a pci_dev structure.
10 * The OS initialization, configuring of the adapter private structure,
11 * and a hardware reset occur.
12 */
13 static int e1000_probe(struct pci_dev *pdev, const struct pci_device_id
14 {
15 ...
16 ...
17     netdev->netdev_ops = &e1000_netdev_ops;
18     e1000_set_ethtool_ops(netdev);
19 ...
20 ...
21 }

```

这个函数很长，我们不都列出来，这是e1000主要的初始化函数，即使从注释都能看出来。我们留意其注册了netdev的netdev\_ops，用的是e1000\_netdev\_ops这个结构体：

```

1 static const struct net_device_ops e1000_netdev_ops = {
2     .ndo_open          = e1000_open,
3     .ndo_stop         = e1000_close,

```

```

4     .ndo_start_xmit      = e1000_xmit_frame,
5     .ndo_set_rx_mode     = e1000_set_rx_mode,
6     .ndo_set_mac_address = e1000_set_mac,
7     .ndo_tx_timeout       = e1000_tx_timeout,
8 ...
9 ...
10 };

```

这个e1000的方法集里有一个重要的方法，e1000\_open，我们要说的中断的注册就从这里开始：

```

1 /**
2  * e1000_open - Called when a network interface is made active
3  * @netdev: network interface device structure
4  *
5  * Returns 0 on success, negative value on failure
6  *
7  * The open entry point is called when a network interface is made
8  * active by the system (IFF_UP). At this point all resources needed
9  * for transmit and receive operations are allocated, the interrupt
10 * handler is registered with the OS, the watchdog task is started,
11 * and the stack is notified that the interface is ready.
12 */
13 int e1000_open(struct net_device *netdev)
14 {
15     struct e1000_adapter *adapter = netdev_priv(netdev);
16     struct e1000_hw *hw = &adapter->hw;
17 ...
18 ...
19     err = e1000_request_irq(adapter);
20 ...
21 }

```

e1000在这里注册了中断：

```

1 static int e1000_request_irq(struct e1000_adapter *adapter)
2 {
3     struct net_device *netdev = adapter->netdev;

```

```

4     irq_handler_t handler = e1000_intr;
5     int irq_flags = IRQF_SHARED;
6     int err;
7     err = request_irq(adapter->pdev->irq, handler, irq_flags, netdev
8 ...
9 ...
10 }
11

```

如上所示，这个被注册的中断处理函数，也就是handler，就是e1000\_intr()。我们不展开这个中断处理函数看了，我们知道中断处理函数在这里被注册了，在网络包来的时候会触发这个中断函数。

## 注册软中断

内核初始化期间，softirq\_init会注册TASKLET\_SOFTIRQ以及HI\_SOFTIRQ相关联的处理函数。

```

1 void __init softirq_init(void)
2 {
3     .....
4     open_softirq(TASKLET_SOFTIRQ, tasklet_action);
5     open_softirq(HI_SOFTIRQ, tasklet_hi_action);
6
7 }

```

网络子系统分两种soft IRQ。NET\_TX\_SOFTIRQ和NET\_RX\_SOFTIRQ，分别处理发送数据包和接收数据包。这两个soft IQ在net\_dev\_init函数 (net/core/dev.c) 中注册：

```

1 open_softirq(NET_TX_SOFTIRQ, net_tx_action);
2 open_softirq(NET_RX_SOFTIRQ, net_rx_action);

```

收发数据包的软中断处理函数被注册为net\_rx\_action和net\_tx\_action。其中open\_softirq实现为：

```

1 void open_softirq(int nr, void (*action)(struct softirq_action *))
2 {
3     softirq_vec[nr].action = action;

```

```

4
5  }

```

```

Breakpoint 4, softirq_init () at kernel/softirq.c:586
586          open_softirq(TASKLET_SOFTIRQ, tasklet_action);
(gdb) bt
#0  softirq_init () at kernel/softirq.c:586
#1  0xffffffff829aed8f in start_kernel () at init/main.c:680
#2  0xffffffff810000d4 in secondary_startup_64 () at arch/x86/kernel/head_64.S:241
#3  0x0000000000000000 in ?? ()
(gdb) li
581          &per_cpu(tasklet_vec, cpu).head;
582          per_cpu(tasklet_hi_vec, cpu).tail =
583              &per_cpu(tasklet_hi_vec, cpu).head;
584      }
585
586          open_softirq(TASKLET_SOFTIRQ, tasklet_action);
587          open_softirq(HI_SOFTIRQ, tasklet_hi_action);
588      }
589
590 static int ksoftirqd_should_run(unsigned int cpu)

```

 极客重生

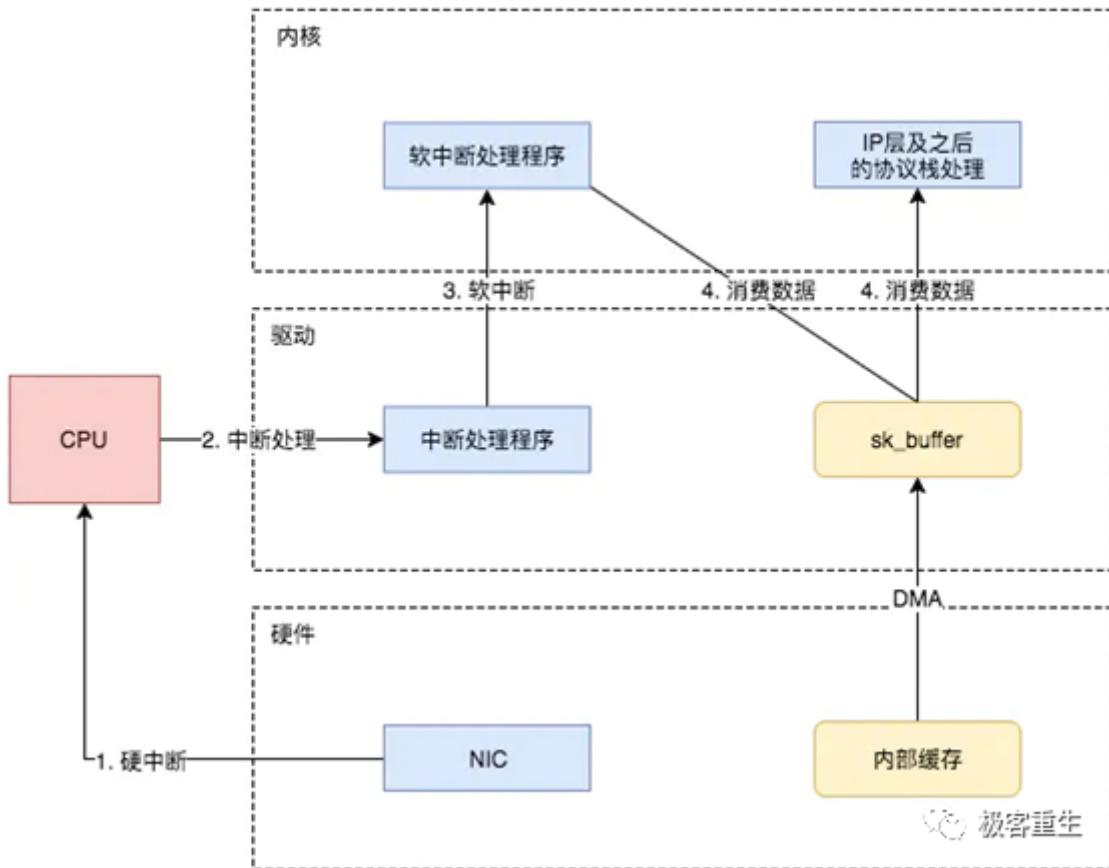
```

(gdb) li
452          or_softirq_pending(1UL << nr);
453      }
454
455 void open_softirq(int nr, void (*action)(struct softirq_action *))
456 {
457     softirq_vec[nr].action = action;
458 }
459
460 /*
461  * Tasklets
(gdb) bt
#0  open_softirq (nr=7, action=0xffffffff810a0060 <run_rebalance_domains>) at kernel/softirq.c:457
#1  0xffffffff829cc2c8 in init_sched_fair_class () at kernel/sched/fair.c:10543
#2  0xffffffff829cc202 in sched_init () at kernel/sched/core.c:6714
#3  0xffffffff829aecf0 in start_kernel () at init/main.c:640
#4  0xffffffff810000d4 in secondary_startup_64 () at arch/x86/kernel/head_64.S:241
#5  0x0000000000000000 in ?? ()

```

 极客重生

- 从硬中断到软中断



## Linux 网络启动的准备工作

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，后面就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

### 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。



相关代码如下：

```
1 //file: kernel/softirq.c
2 static struct smp_hotplug_thread softirq_threads = {
3     .store          = &ksoftirqd,
4     .thread_should_run = ksoftirqd_should_run,
5     .thread_fn      = run_ksoftirqd,
6     .thread_comm    = "ksoftirqd/%u",
7 };
```

```
Breakpoint 1, smpboot_register_percpu_thread (plug_thread=0xffffffff824531c0 <cpu_stop_threads>) at kernel/smpboot.c:296
296 {
(gdb) li
285     * @plug_thread:      Hotplug thread descriptor
286     *
287     * Creates and starts the threads on all online cpus.
288     */
289     int smpboot_register_percpu_thread(struct smp_hotplug_thread *plug_thread)
290 {
291     unsigned int cpu;
292     int ret = 0;
293
294     get_online_cpus();
(gdb)
295     mutex_lock(&smpboot_threads_lock);
296     for_each_online_cpu(cpu) {
297         ret = __smpboot_create_thread(plug_thread, cpu);
298         if (ret) {
299             smpboot_destroy_threads(plug_thread);
300             goto out;
301         }
302         smpboot_unpark_thread(plug_thread, cpu);
303     }
304     list_add(&plug_thread->list, &hotplug_threads);
(gdb)
305 out:
306     mutex_unlock(&smpboot_threads_lock);
307     put_online_cpus();
308     return ret;
309 }
310 EXPORT_SYMBOL_GPL(smpboot_register_percpu_thread);
311
312 /**
313  * smpboot_unregister_percpu_thread - Unregister a per_cpu thread related to hotplug
314  * @plug_thread:      Hotplug thread descriptor
Breakpoint 2, spawn_ksoftirqd () at kernel/softirq.c:682
682         cpuhp_setup_state_nocalls(CPUHP_SOFTIRQ_DEAD, "softirq:dead", NULL,
(gdb) li
677         .thread_comm          = "ksoftirqd/%u",
678     };
679
680     static __init int spawn_ksoftirqd(void)
681     {
682         cpuhp_setup_state_nocalls(CPUHP_SOFTIRQ_DEAD, "softirq:dead", NULL,
683                                 takeover_tasklets);
684         BUG_ON(smpboot_register_percpu_thread(&softirq_threads));
685
686         return 0;
(gdb)
687     }
688     early_initcall(spawn_ksoftirqd);
```

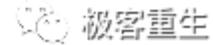
 极客重生

 极客重生

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，软中断不仅只有网络软中断，还有其它类型。

## 创建ksoftirqd内核线程

3. 注册NET\_RX\_SOFTIRQ的处理函数为net\_rx\_action  
注册NET\_TX\_SOFTIRQ的处理函数为net\_tx\_action



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。

```

Breakpoint 3, net_dev_init () at net/core/dev.c:10157
10157      BUG_ON(!dev_boot_phase);
(gdb) li
10152      */
10153  static int __init net_dev_init(void)
10154  {
10155      int i, rc = -ENOMEM;
10156
10157      BUG_ON(!dev_boot_phase);
10158
10159      if (dev_proc_init())
10160          goto out;
10161
(gdb)
10162      if (netdev_kobject_init())
10163          goto out;
10164
10165      INIT_LIST_HEAD(&ptype_all);
10166      for (i = 0; i < PTYPE_HASH_SIZE; i++)
10167          INIT_LIST_HEAD(&ptype_base[i]);
10168
10169      INIT_LIST_HEAD(&offload_base);
10170
10171      if (register_pernet_subsys(&netdev_net_ops))
(gdb)
10172          goto out;
10173
10174      /*
10175      *      Initialise the packet receive queues.
10176      */
10177
10178      for_each_possible_cpu(i) {
10179          struct work_struct *flush = per_cpu_ptr(&flush_works, i);
10180          struct softnet_data *sd = &per_cpu(softnet_data, i);
10181
(gdb)
10182          INIT_WORK(flush, flush_backlog);
10183
10184          skb_queue_head_init(&sd->input_pkt_queue);
10185          skb_queue_head_init(&sd->process_queue);

```



```

10185         skb_queue_head_init(&sd->process_queue);
10186 #ifdef CONFIG_XFRM_OFFLOAD
10187         skb_queue_head_init(&sd->xfrm_backlog);
10188 #endif
10189         INIT_LIST_HEAD(&sd->poll_list);
10190         sd->output_queue_tailp = &sd->output_queue;
10191 #ifdef CONFIG_RPS
(gdb)
10192         sd->csd.func = rps_trigger_softirq;
10193         sd->csd.info = sd;
10194         sd->cpu = i;
10195 #endif
10196
10197         init_gro_hash(&sd->backlog);
10198         sd->backlog.poll = process_backlog;
10199         sd->backlog.weight = weight_p;
10200     }
10201 (gdb)
10202     dev_boot_phase = 0;
10203
10204 /* The loopback device is special if any other network devices
10205 * is present in a network namespace the loopback device must
10206 * be present. Since we now dynamically allocate and free the
10207 * loopback device ensure this invariant is maintained by
10208 * keeping the loopback device as the first device on the
10209 * list of network devices. Ensuring the loopback devices
10210 * is the first device that appears and the last network device
10211 * that disappears.
(gdb)
10212 */
10213     if (register_pernet_device(&loopback_net_ops))
10214         goto out;
10215
10216     if (register_pernet_device(&default_device_ops))
10217         goto out;
10218
10219     open_softirq(NET_TX_SOFTIRQ, net_tx_action);
10220     open_softirq(NET_RX_SOFTIRQ, net_rx_action);
10221 (gdb)
10222     rc = cpuhp_setup_state_nocalls(CPUHP_NET_DEV_DEAD, "net/dev:dead",
10223                                     NULL, dev_cpu_dead);
10224     WARN_ON(rc < 0);
10225     rc = 0;
10226 out:

```

 极客重生

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。

## 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

相关代码如下

```

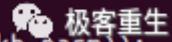
1 //file: net/ipv4/af_inet.c
2
3 static struct packet_type ip_packet_type __read_mostly = {
4     .type = cpu_to_be16(ETH_P_IP),
5     .func = ip_recv,
6 };
7
8 static const struct net_protocol udp_protocol = {
9     .handler = udp_recv,
10    .err_handler = udp_err,
11    .no_policy = 1,
12    .netns_ok = 1,
13 };
14
15 static const struct net_protocol tcp_protocol = {
16     .early_demux = tcp_v4_early_demux,
17     .handler = tcp_v4_recv,
18     .err_handler = tcp_v4_err,
19     .no_policy = 1,
20     .netns_ok = 1,
21 };

```

```

Breakpoint 10, inet_init () at net/ipv4/af_inet.c:1910
1910 {
(gdb) li
1905     .func = ip_recv,
1906     .list_func = ip_list_recv,
1907 };
1908
1909 static int __init inet_init(void)
1910 {
1911     struct inet_protosw *q;
1912     struct list_head *r;
1913     int rc = -EINVAL;
1914
(gdb) 1915     sock_skb_cb_check_size(sizeof(struct inet_skb_parm));
1916

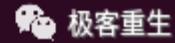
```



```

1939 #ifdef CONFIG_SYSCTL
1940     ip_static_sysctl_init();
1941 #endif
1942
1943     /*
1944      *          Add all the base protocols.
1945 */
1946
1947     if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
1948         pr_crit("%s: Cannot add ICMP protocol\n", __func__);
1949     if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
1950         pr_crit("%s: Cannot add UDP protocol\n", __func__);
1951     if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
1952         pr_crit("%s: Cannot add TCP protocol\n", __func__);
1953 #ifdef CONFIG_IP_MULTICAST
1954     if (inet_add_protocol(&igmp_protocol, IPPROTO_IGMP) < 0)
1955         pr_crit("%s: Cannot add IGMP protocol\n", __func__);
1956 #endif
1957
1958     /* Register the socket-side information for inet_create. */
1959     for (r = &inetsw[0]; r < &inetsw[SOCK_MAX]; ++r)
1960         INIT_LIST_HEAD(r);
1961
1962     for (q = inetsw_array; q < &inetsw_array[INETSW_ARRAY_LEN]; ++q)
1963         inet_register_protosw(q);
1964
1965     /*
1966      *          Set the ARP module up
1967      */
1968     arp_init();
1969
1970     /*
1971      *          Set the IP module up
1972      */
1973
1974
1975     ip_init();
1976
1977     /* Setup TCP slab cache for open requests. */
1978     tcp_init();
1979
1980     /* Setup UDP memory threshold */
1981     udp_init();
1982
1983     /* Add UDP-Lite (RFC 3828) */
1984     udplite4_register();
1985
1986     raw_init();
1987
1988     ping_init();
1989
1990     /*
1991      *          Set the ICMP layer up
1992      */
1993
1994     if (icmp_init() < 0)
1995         panic("Failed to create the ICMP control socket.\n");
1996
1997     /*
1998      *          Initialise the multicast router
1999      */
2000 #if defined(CONFIG_IP_MROUTE)
2001     if (ip_mr_init())
2002         pr_crit("%s: Cannot init ipv4 mroute\n", __func__);
2003 #endif
2004

```

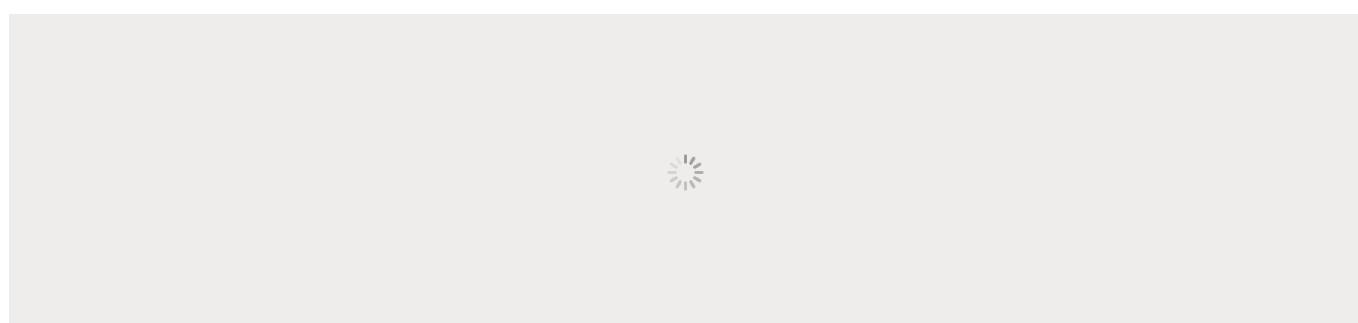


扩展一下，如果看一下ip\_rcv和udp\_rcv等函数的代码能看到很多协议的处理过程。例如，ip\_rcv中会处理netfilter和iptable过滤，如果你有很多或者很复杂的netfilter或iptables规则，这些规则都是在软中断的上下文中执行的，会加大网络延迟。再例如，udp\_rcv中会判断socket接收队列是否满了。对应的的相关内核参数是net.core.rmem\_max和net.core.rmem\_default。如果有兴趣，建议大家好好读一下inet\_init这个函数的代码。

## 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

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：



第5步中我们看到，网卡驱动实现了ethtool所需要的接口，也在那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。

```

1 //file: drivers/net/ethernet/intel/igb/igb_main.
2 .....
3 static const struct net_device_ops igb_netdev_ops = {
4     .ndo_open          = igb_open,
5     .ndo_stop         = igb_close,
```

```

6 .ndo_start_xmit          = igb_xmit_frame,
7 .ndo_get_stats64         = igb_get_stats64,
8 .ndo_set_rx_mode         = igb_set_rx_mode,
9 .ndo_set_mac_address     = igb_set_mac,
10 .ndo_change_mtu         = igb_change_mtu,
11 .ndo_do_ioctl           = igb_ioctl,.....
12 }
```

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。

```

1 static int igb_alloc_q_vector(struct igb_adapter *adapter,
2                               int v_count, int v_idx,
3                               int txr_count, int txr_idx,
4                               int rxr_count, int rxr_idx)
5 {
6     .....
7     /* initialize NAPI */
8     netif_napi_add(adapter->netdev, &q_vector->napi,
9                     igb_poll, 64);
10
11 }
```

## 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：



```

1 //file: drivers/net/ethernet/intel/igb/igb_main.c
2 static int __igb_open(struct net_device *netdev, bool resuming)
3 {
4     /* allocate transmit descriptors */
5     err = igb_setup_all_tx_resources(adapter);
6     /* allocate receive descriptors */
7     err = igb_setup_all_rx_resources(adapter);
8     /* 注册中断处理函数 */
9     err = igb_request_irq(adapter);
10    if (err)
11        goto err_req_irq;
12    /* 启用NAPI */
13    for (i = 0; i < adapter->num_q_vectors; i++)
14        napi_enable(&(adapter->q_vector[i]->napi));
15    .....
16 }

```

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:

```

1 static int igb_request_irq(struct igb_adapter *adapter)
2 {
3     if (adapter->msix_entries) {
4         err = igb_request_msix(adapter);
5         if (!err)
6             goto request_done;
7         .....
8     }
9 }
10
11 static int igb_request_msix(struct igb_adapter *adapter)
12 {
13     .....
14     for (i = 0; i < adapter->num_q_vectors; i++) {
15         ...
16         err = request_irq(adapter->msix_entries[vector].vector,

```

```

17         igb_msix_ring, 0, q_vector->name,
18     }
19

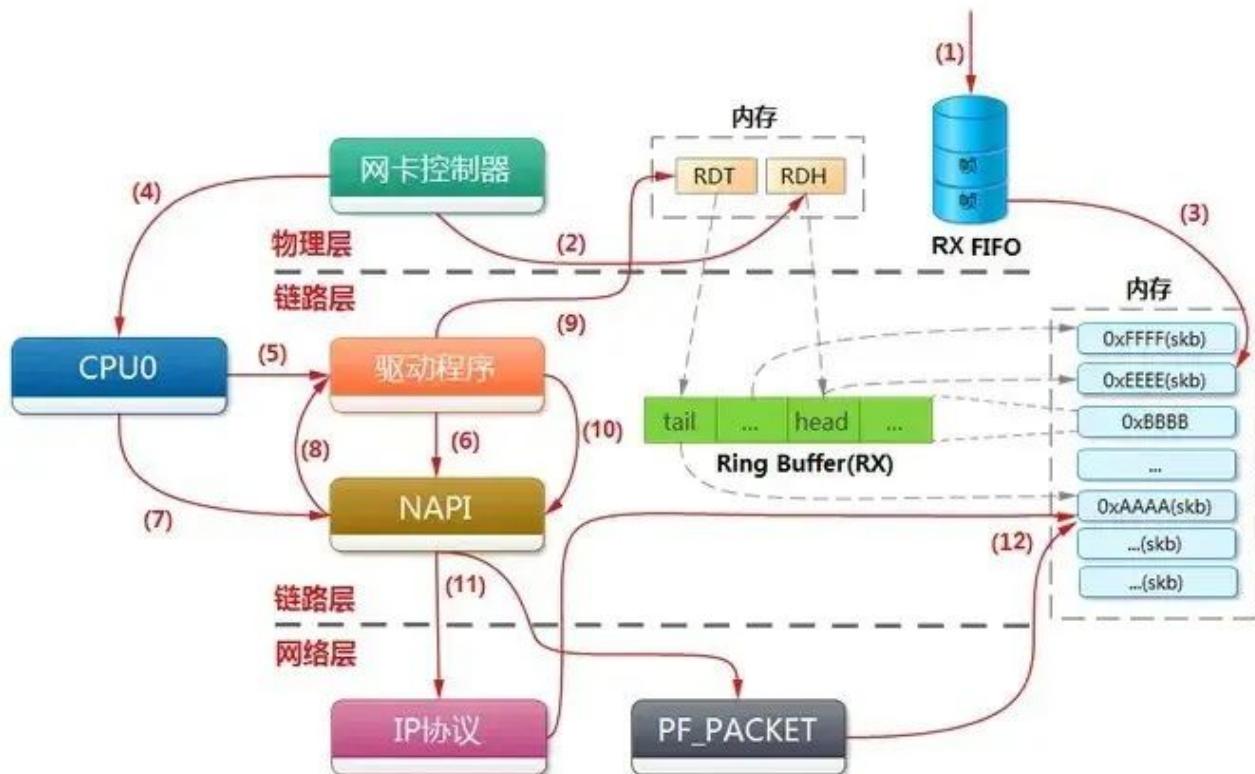
```

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）。

到此准备工作完成。

## Linux网络包：中断到网络层接收

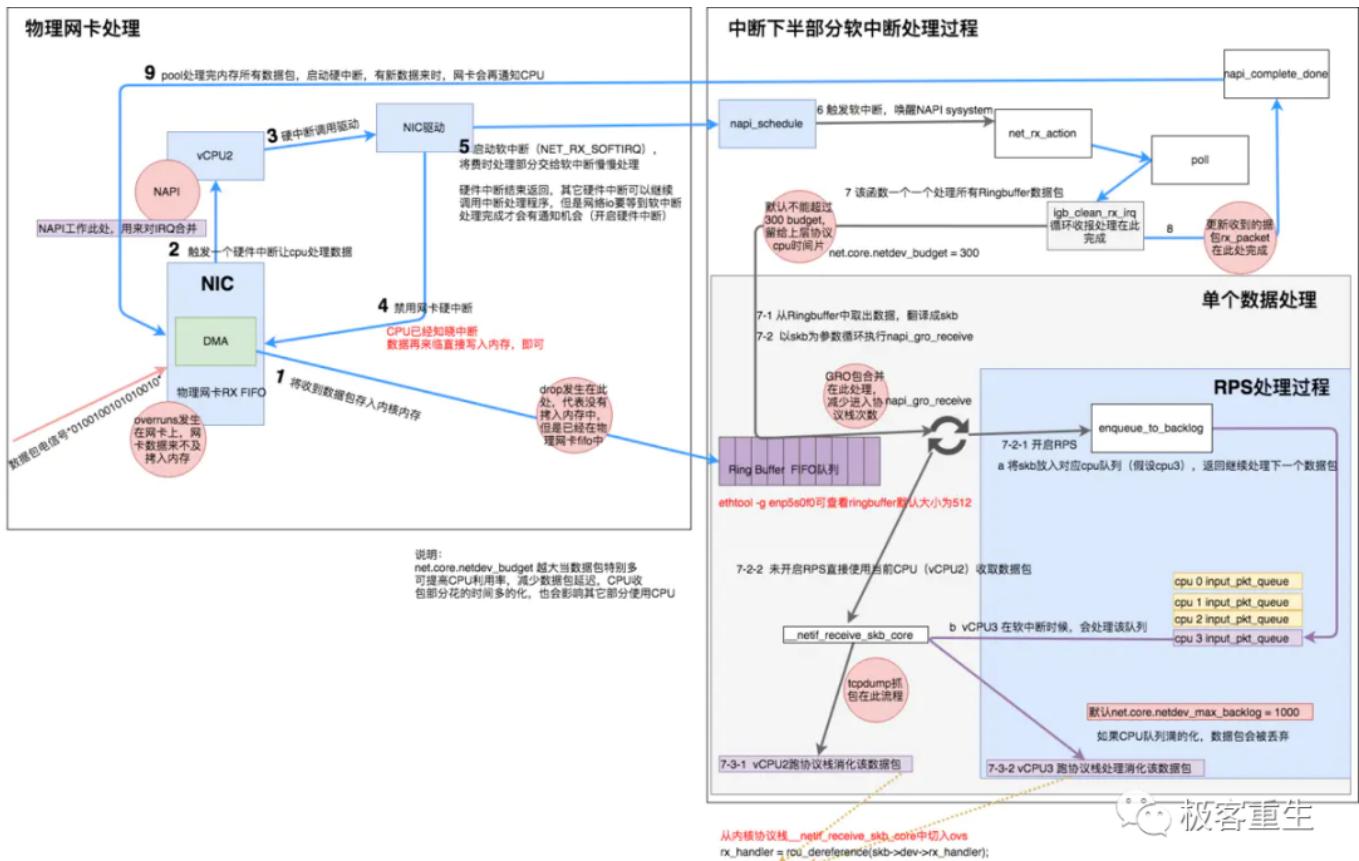
网卡收包从整体上是网线中的高低电平转换到网卡FIFO存储再拷贝到系统主内存（DDR3）的过程，其中涉及到网卡控制器，CPU，DMA，驱动程序，在OSI模型中属于物理层和链路层，如下图所示。



注：物理层指网卡，物理层的所有部件都是网卡的组成部分。

极客重生

## 中断处理



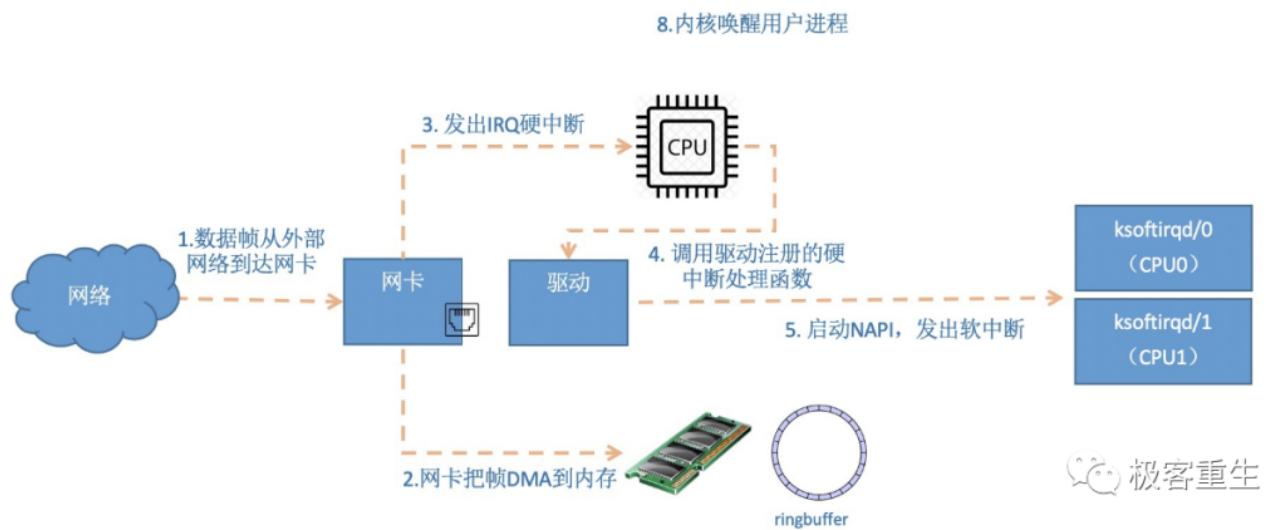
物理网卡收到数据包的处理流程如上图左半部分所示，详细步骤如下：

- 网卡收到数据包，先将高低电平转换到网卡fifo存储，网卡申请ring buffer的描述，根据描述找到具体的物理地址，从fifo队列物理网卡会使用DMA将数据包写到了该物理地址，其实就是在skb\_buffer中。
- 这个时候数据包已经被转移到skb\_buffer中，因为是DMA写入，内核并没有监控数据包写入情况，这时候NIC触发一个硬中断，每一个硬件中断会对应一个中断号，且指定一个vCPU来处理，如上图vcpu2收到了该硬件中断。
- 硬件中断的中断处理程序，调用驱动程序完成，a.启动软中断
- 硬中断触发的驱动程序会禁用网卡硬中断，其实这时候意思是告诉NIC，再来数据不用触发硬中断了，把数据DMA拷入系统内存即可
- 硬中断触发的驱动程序会启动软中断，启用软中断目的是将数据包后续处理流程交给软中断慢慢处理，这个时候退出硬件中断了，但是注意和网络有关的硬中断，要等到后续开启硬中断后，才有机会再次被触发
- NAPI触发软中断，触发napi系统
- 消耗ringbuffer指向的skb\_buffer
- NAPI循环处理ringbuffer数据，处理完成
- 启动网络硬件中断，有数据来时候就可以继续触发硬件中断，继续通知CPU来消耗数据包。

其实上述过程简单描述为：网卡收到数据包，DMA到内核内存，中断通知内核数据有了，内核按轮次处理消耗数据包，一轮处理完成后，开启硬中断。其核心就是网卡和内核其实是生

生产和消费模型，网卡生产，内核负责消费，生产者需要通知消费者消费；如果生产过快会产生丢包，如果消费过慢也会产生问题。也就说在高流量压力情况下，只有生产消费优化后，消费能力够快，此生产消费关系才可以正常维持，所以如果物理接口有丢包计数时候，未必是网卡存在问题，也可能是内核消费的太慢。

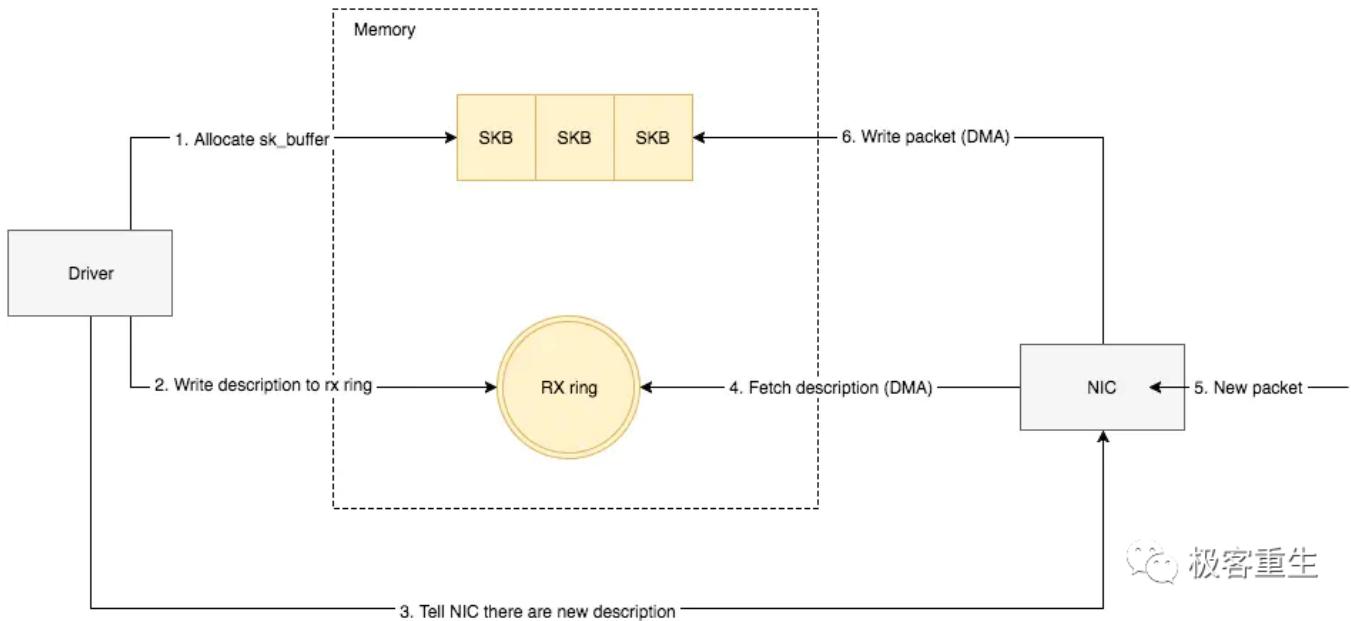
关于CPU与ksoftirqd的关系可以描述如下：



### 网卡收到的数据写入到内核内存

NIC在接收到数据包之后，首先需要将数据同步到内核中，这中间的桥梁是rx ring buffer。它是由NIC和驱动程序共享的一片区域，事实上，rx ring buffer存储的并不是实际的packet数据，而是一个描述符，这个描述符指向了它真正的存储地址，具体流程如下：

1. 驱动在内存中分配一片缓冲区用来接收数据包，叫做sk\_buffer;
2. 将上述缓冲区的地址和大小（即接收描述符），加入到rx ring buffer。描述符中的缓冲区地址是DMA使用的物理地址;
3. 驱动通知网卡有一个新的描述符;
4. 网卡从rx ring buffer中取出描述符，从而获知缓冲区的地址和大小;
5. 网卡收到新的数据包;
6. 网卡将新数据包通过DMA直接写到sk\_buffer中。



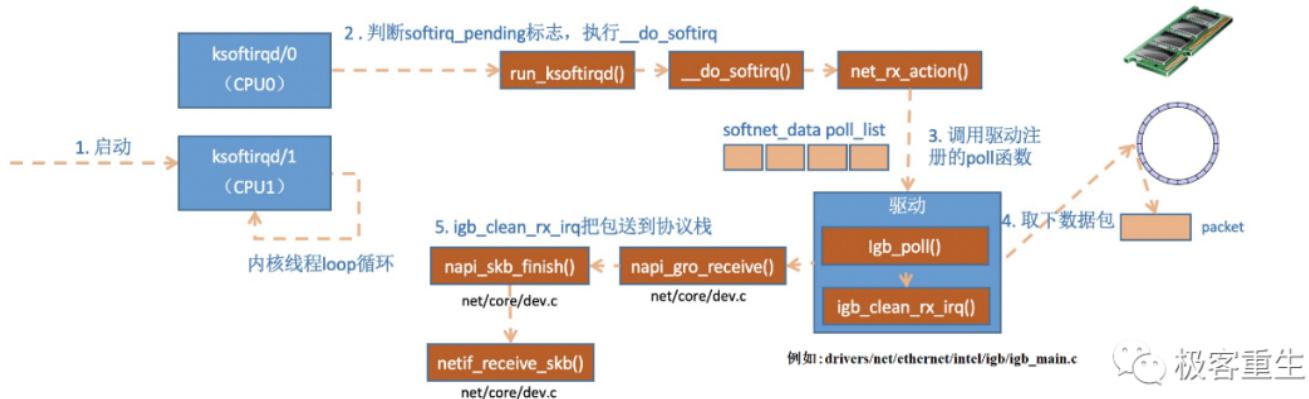
当驱动处理速度跟不上网卡收包速度时，驱动来不及分配缓冲区，NIC接收到的数据包无法及时写到sk\_buffer，就会产生堆积，当NIC内部缓冲区写满后，就会丢弃部分数据，引起丢包。这部分丢包为rx\_fifo\_errors，在/proc/net/dev中体现为fifo字段增长，在ifconfig中体现为overruns指标增长。

## 中断下半部分

ksoftirqd内核线程处理软中断，即中断下半部分软中断处理过程：

- 1.NAPI（以e1000网卡为例）：net\_rx\_action() -> e1000\_clean() -> e1000\_clean\_rx\_irq() -> e1000\_receive\_skb() -> netif\_receive\_skb()
2. 非 NAPI（以 dm9000 网卡为例）：net\_rx\_action() -> process\_backlog() -> netif\_receive\_skb()

最后网卡驱动通过netif\_receive\_skb()将sk\_buff上送协议栈。

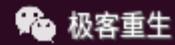


内核线程初始化的时候，我们介绍了ksoftirqd中两个线程函数ksoftirqd\_should\_run和run\_ksoftirqd。其中ksoftirqd\_should\_run代码如下：

```

Breakpoint 6, ksoftirqd_should_run (cpu=0) at kernel/softirq.c:592
592         return local_softirq_pending();
(gdb) li
587         open_softirq(HI_SOFTIRQ, tasklet_hi_action);
588     }
589
590     static int ksoftirqd_should_run(unsigned int cpu)
591     {
592         return local_softirq_pending();
593     }
594
595     static void run_ksoftirqd(unsigned int cpu)
596     {
(gdb)
597         local_irq_disable();
598         if (local_softirq_pending()) {
599             /*
600             * We can safely run softirq on inline stack, as we are not deep
601             * in the task stack here.
602             */
603             __do_softirq();
604             local_irq_enable();
605             cond_resched();
606         }
607     }
608     local_irq_enable();
609 }

```



```

#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：

```

1 static void run_ksoftirqd(unsigned int cpu)
2 {
3     local_irq_disable();
4     if (local_softirq_pending()) {
5         __do_softirq();
6         rcu_note_context_switch(cpu);
7         local_irq_enable();
8         cond_resched();
9     }
10 }
11 local_irq_enable();
12 }

```

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)
```

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通的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。

```
1 static void net_rx_action(struct softirq_action *h)
2 {
3     struct softnet_data *sd = &__get_cpu_var(softnet_data);
4     unsigned long time_limit = jiffies + 2;
5     int budget = netdev_budget;
6     void *have;
7     local_irq_disable();
8     while (!list_empty(&sd->poll_list)) {
9         .....
10        n = list_first_entry(&sd->poll_list, struct napi_struct, poll_li
11        work = 0;
12        if (test_bit(NAPI_STATE_SCHED, &n->state)) {
13            work = n->poll(n, weight);
14            trace_napi_poll(n);
15        }
16        budget -= work;
17    }
```

18 }

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。

```

1 /**
2  * igb_poll - NAPI Rx polling callback
3  * @napi: napi polling structure
4  * @budget: count of how many packets we should handle
5 */
6 static int igb_poll(struct napi_struct *napi, int budget)
7 {
8     ...
9     if (q_vector->tx.ring)
10         clean_complete = igb_clean_tx_irq(q_vector);
11     if (q_vector->rx.ring)
12         clean_complete &= igb_clean_rx_irq(q_vector, budget);
13     ...
14 }
```

在读取操作中，igb\_poll的重点工作是对igb\_clean\_rx\_irq的调用。

```

1 static bool igb_clean_rx_irq(struct igb_q_vector *q_vector, const int bu
2 {
3     ...
4     do {
5         /* retrieve a buffer from the ring */
6         skb = igb_fetch_rx_buffer(rx_ring, rx_desc, skb);
7         /* fetch next buffer in frame if non-eop */
8         if (igb_is_non_eop(rx_ring, rx_desc))
9             continue;
10        }
11        /* verify the packet layout is correct */
12        if (igb_cleanup_headers(rx_ring, rx_desc, skb)) {
13            skb = NULL;
```

```

14 continue;
15 }
16 /* populate checksum, timestamp, VLAN, and protocol */
17 igb_process_skb_fields(rx_ring, rx_desc, skb);
18 napi_gro_receive(&q_vector->napi, skb);
19 }
```

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skb变量的timestamp, VLAN id, protocol等字段。接下来进入到napi\_gro\_receive中：

```

1 //file: net/core/dev.c
2 gro_result_t napi_gro_receive(struct napi_struct *napi, struct sk_buff *sk
3 {
4     skb_gro_reset_offset(skb);
5     return napi_skb_finish(dev_gro_receive(napi, skb), skb);
6 }
```

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。

```

1 //file: net/core/dev.c
2 static gro_result_t napi_skb_finish(gro_result_t ret, struct sk_buff *sk
3 {
4     switch (ret) {
5         case GRO_NORMAL:
6             if (netif_receive_skb(skb))
7                 ret = GRO_DROP;
8             break;
9             .....
10 }
```

在netif\_receive\_skb中，数据包将被送到协议栈中，接下来在网络层协议层的处理流程便不再赘述。

## 总结

### send发包过程

- 1、网卡驱动创建tx descriptor ring（一致性DMA内存），将tx descriptor ring的总线地址写入网卡寄存器TDBA
- 2、协议栈通过dev\_queue\_xmit()将sk\_buff下送网卡驱动
- 3、网卡驱动将sk\_buff放入tx descriptor ring，更新TDT
- 4、DMA感知到TDT的改变后，找到tx descriptor ring中下一个将要使用的descriptor
- 5、DMA通过PCI总线将descriptor的数据缓存区复制到Tx FIFO
- 6、复制完后，通过MAC芯片将数据包发送出去
- 7、发送完后，网卡更新TDH，启动硬中断通知CPU释放数据缓存区中的数据包

### recv收包过程

- 1、网卡驱动创建rx descriptor ring（一致性DMA内存），将rx descriptor ring的总线地址写入网卡寄存器RDBA
- 2、网卡驱动为每个descriptor分配sk\_buff和数据缓存区，流式DMA映射数据缓存区，将数据缓存区的总线地址保存到descriptor
- 3、网卡接收数据包，将数据包写入Rx FIFO
- 4、DMA找到rx descriptor ring中下一个将要使用的descriptor
- 5、整个数据包写入Rx FIFO后，DMA通过PCI总线将Rx FIFO中的数据包复制到descriptor的数据缓存区
- 6、复制完后，网卡启动硬中断通知CPU数据缓存区中已经有新的数据包了，CPU执行硬中断函数：

NAPI（以e1000网卡为例）：e1000\_intr() -> \_\_napi\_schedule() -> \_\_raise\_softirq\_irqoff(NET\_RX\_SOFTIRQ)

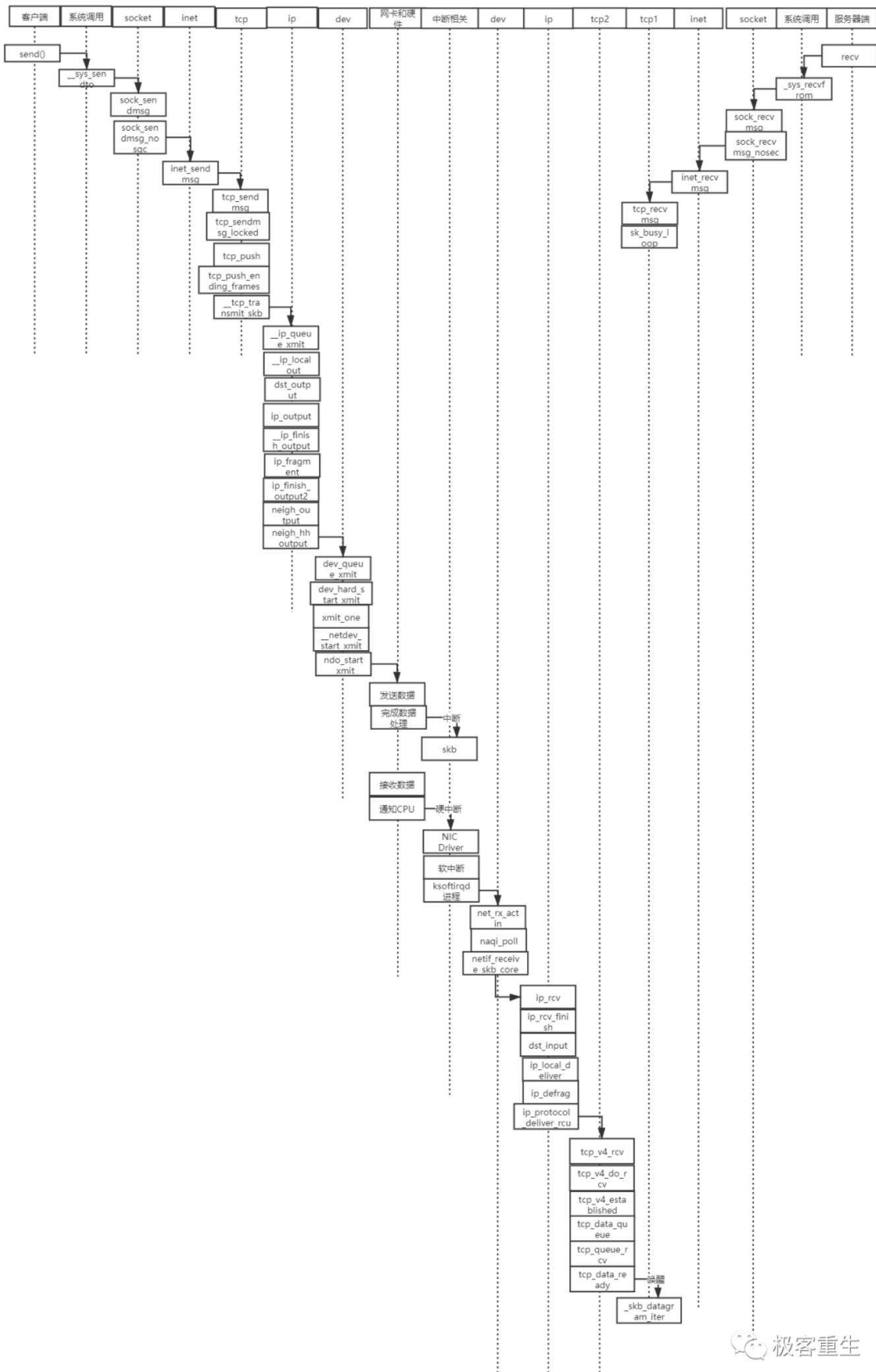
非NAPI（以dm9000网卡为例）：dm9000\_interrupt() -> dm9000\_rx() -> netif\_rx() -> napi\_schedule() -> \_\_napi\_schedule() -> \_\_raise\_softirq\_irqoff(NET\_RX\_SOFTIRQ)

7、ksoftirqd执行软中断函数net\_rx\_action()：

NAPI（以e1000网卡为例）：net\_rx\_action() -> e1000\_clean() -> e1000\_clean\_rx\_irq() -> e1000\_receive\_skb() -> netif\_receive\_skb()

非NAPI（以dm9000网卡为例）：net\_rx\_action() -> process\_backlog() -> netif\_receive\_skb()

## 8、网卡驱动通过netif\_receive\_skb()将sk\_buff上送协议栈



## Linux网络子系统的分层

Linux网络子系统实现需要：

- 支持不同的协议族 (INET, INET6, UNIX, NETLINK...)
- 支持不同的网络设备
- 支持统一的BSD socket API
- 需要屏蔽协议、硬件、平台(API)的差异，因而采用分层结构

### 系统调用

系统调用提供用户的应用程序访问内核的唯一途径。协议无关接口由socket layer来实现的，其提供一组通用功能，以支持各种不同的协议。网络协议层为socket层提供具体协议接口——proto{}, 实现具体的协议细节。设备无关接口，提供一组通用函数供底层网络设备驱动程序使用。设备驱动与特定网卡设备相关，定义了具体的协议细节，会分配一个net\_device结构，然后用其必需的例程进行初始化。

来源：<https://www.cnblogs.com/ypholic/p/14337328.html>

- END -

看完一键三连**在看， 转发， 点赞**

是对文章最大的赞赏，极客重生感谢你❤

### 推荐阅读

[图解Linux 内核TCP/IP 协议栈实现|Linux网络硬核系列](#)

[网络排障全景指南手册v1.0精简版pdf](#)

[一个奇葩的网络问题](#)



收录于话题 #深入理解网络 31

< 上一篇

TCP/IP协议栈到底是内核态好还是用户态好?

下一篇 >

Google的TCP BBR拥塞控制算法深度解析

喜欢此内容的人还喜欢

算法面试 | 论如何4个月高效刷满 500 题并形成长期记忆

极客重生

