E1000E 驱动源码分析

mnstory.net

驱动初始化与删除

我们从入口函数 module_init(e1000_init_module) 开始看，不过这个 e1000_init_module 简洁得只剩一句话 pci_register_driver(&e1000_driver)，看看 e1000_driver 如何定义：

```
/* PCI Device API Driver */
static struct pci_driver e1000_driver = {
    //就像你一样，是个东西都得有名字，它的名字是
    //char e1000e_driver_name[] = "e1000e"
    .name     = e1000e_driver_name,
    .id_table = e1000_pci_tbl,
    .probe    = e1000_probe,
    .remove   = e1000_remove,
#ifdef CONFIG_PM
    //支持电源管理
    .driver   = {
        .pm = &e1000_pm_ops,
    },
#endif
    .shutdown = e1000_shutdown,
    .err_handler = &e1000_err_handler
};
```

定义一个 PCI 驱动，哪些 PCI 设备可以使用此驱动？
系统很热心，帮你把判断的代码写了，你只需提供匹配表即可，而.id_table 就是匹配表。
你造吗？PCI 有 3 种地址空间：I/O 空间，内存地址空间，配置空间。而配置空间一个 PCI 的配置空间至少有 256 字节，布局如：

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x00	Vendor ID		Device ID		Command Register		Status Register		Rev ID	Class Code			Cache Line	Latency Timer	Header Type	BIST
0x10	Base Address 0				Base Address 1				Base Address 2				Base Address 3			
0x20	Base Address 4				Base Address 5				CardBus CIS Pointer				Subsystem Vendor ID		Subsystem Device ID	
0x30	Expansion ROM Base Address				Reserved								IRQ	IRQ Pin	Min_Gnt	Max_Lat
0x40 … 0xf0	Device specific area															

图片来源：http://blog.csdn.net/lamdoc/article/details/7698709

id_table 是 struct pci_device_id 类型的一个数组，每一条 pci_device_id，就是一条使用的 PCI 硬件信息，match 上了，就可以使用这个驱动，例如我的网卡，其类型为：

```
#lspci -nn | grep -E 'Ethernet controller.*Intel'
04:00.0 Ethernet controller [0200]: Intel Corporation 82574L Gigabit Network Connection [8086:10d3]
```

那就匹配上这条 { PCI_VDEVICE(INTEL, E1000_DEV_ID_82574L), board_82574 }

e1000_driver 里，我们关注两个函数，一个初始化(.probe=e1000_probe)，一个析构(.remove=e1000_remove)。初始化说完了，析构也简单，一句话，反注册 pci：

```
static void __exit e1000_exit_module(void)
{
    pci_unregister_driver(&e1000_driver);
}
```

网口初始化与删除

```
static int e1000_probe(struct pci_dev *pdev, const struct
pci_device_id *ent)
{
    struct net_device *netdev;
    struct e1000_adapter *adapter;
    struct e1000_hw *hw;
    //ent->driver_data相当于支持e1000e驱动的硬件版本号
    //通过lspci可以看出网卡型号，例如我的Intel Corporation 82574L
Gigabit Network Connection, 所以ei为e1000_82574_info
```

```
    const struct e1000_info *ei = e1000_info_tbl[ent->driver_data];
    resource_size_t mmio_start, mmio_len;
    resource_size_t flash_start, flash_len;
    static int cards_found;
    u16 aspm_disable_flag = 0;
    int bars, i, err, pci_using_dac;
    //EEPROM, Electrically Erasable Programmable Read-Only Memory
    //APME, ACPI Magic Packet filter
    //这么看来，只可以对ACPI包过滤进行设置
    u16 eeprom_data = 0;
    u16 eeprom_apme_mask = E1000_EEPROM_APME;

    //ASPM, Active State Power Management
    //通过lspci -vvvv -s 04:00.0 | grep -E --color 'ASPM|^'
    //查看我的网卡的ASPM属性
    // LnkCap: Port #0, Speed 2.5GT/s, Width x1, ASPM L0s L1, Latency
L0 <128ns, L1 <64us ClockPM- Surprise- LLActRep- BwNot-
    // LnkCtl: ASPM Disabled; RCB 64 bytes Disabled- Retrain-
CommClk+ ExtSynch- ClockPM- AutWidDis- BWInt- AutBWInt-
    if (ei->flags2 & FLAG2_DISABLE_ASPM_L0S)
        aspm_disable_flag = PCIE_LINK_STATE_L0S;
    if (ei->flags2 & FLAG2_DISABLE_ASPM_L1)
        aspm_disable_flag |= PCIE_LINK_STATE_L1;
    if (aspm_disable_flag)
        e1000e_disable_aspm(pdev, aspm_disable_flag);

    //初始化设备内存并通知PCI BIOS使能此设备
    err = pci_enable_device_mem(pdev);
    if (err)
        return err;

    pci_using_dac = 0;
    //通知内核，此PCI设备是DMA 64位寻址
    err = dma_set_mask(&pdev->dev, DMA_BIT_MASK(64));
    if (!err) {
        err = dma_set_coherent_mask(&pdev->dev, DMA_BIT_MASK(64));
        if (!err)
            pci_using_dac = 1;
    } else {
       //干不了，只能退而求其次，用位寻址
        err = dma_set_mask(&pdev->dev, DMA_BIT_MASK(32));
        if (err) {
            err = dma_set_coherent_mask(&pdev->dev,
                            DMA_BIT_MASK(32));
```

```c
            if (err) {
                //oh shit, 都不行，死去吧
                dev_err(&pdev->dev,
                    "No usable DMA configuration, aborting\n");
                goto err_dma;
            }
        }
    }
    //获取IO资源
    bars = pci_select_bars(pdev, IORESOURCE_MEM);
    err = pci_request_selected_regions_exclusive(pdev, bars,
                          e1000e_driver_name);
    if (err)
        goto err_pci_reg;

    /* AER (Advanced Error Reporting) hooks */
    pci_enable_pcie_error_reporting(pdev);

    //让PCI获得使用总线的权限
    pci_set_master(pdev);
    /* PCI config space info */
    err = pci_save_state(pdev);
    if (err)
        goto err_alloc_etherdev;

    err = -ENOMEM;
    //分配net_device结构，里面会预留空间来放e1000_adapter
    netdev = alloc_etherdev(sizeof(struct e1000_adapter));
    if (!netdev)
        goto err_alloc_etherdev;

    SET_NETDEV_DEV(netdev, &pdev->dev);

    netdev->irq = pdev->irq;

    pci_set_drvdata(pdev, netdev);//将net_device和pci关联
    adapter = netdev_priv(netdev);//netdev里面预留的内存，就是
e1000_adapter,用来存储和此PCI设备相关的数据
    //接下来就是对adapter的初始化
    hw = &adapter->hw;
    adapter->netdev = netdev;
    adapter->pdev = pdev;
    adapter->ei = ei;
    adapter->pba = ei->pba;
```

```
    adapter->flags = ei->flags;
    adapter->flags2 = ei->flags2;
    adapter->hw.adapter = adapter;
    adapter->hw.mac.type = ei->mac;
    adapter->max_hw_frame_size = ei->max_hw_frame_size;
    //开启消息类型
    adapter->msg_enable = netif_msg_init(debug, DEFAULT_MSG_ENABLE);

    //这里获取的是PCI bar 0对应的物理地址范围，这个物理地址是由硬件决定
的，BIOS是否参与？
    mmio_start = pci_resource_start(pdev, 0);
    mmio_len = pci_resource_len(pdev, 0);

    err = -EIO;
    //物理地址必须通过ioremap映射为虚拟地址，内核才能访问，如果应用层想访
问，需要用remap_page_range
    adapter->hw.hw_addr = ioremap(mmio_start, mmio_len);
    if (!adapter->hw.hw_addr)
        goto err_ioremap;

    //从流程看，对应我的硬件，这个flags就是e1000_82574_info.flags，流程
走不进来
    if ((adapter->flags & FLAG_HAS_FLASH) &&
        (pci_resource_flags(pdev, 1) & IORESOURCE_MEM)) {
        //如果有flash标记，取bar 1的物理地址映射为flash_address
        flash_start = pci_resource_start(pdev, 1);
        flash_len = pci_resource_len(pdev, 1);
        adapter->hw.flash_address = ioremap(flash_start, flash_len);
        if (!adapter->hw.flash_address)
            goto err_flashmap;
    }

    /* Set default EEE advertisement */
    if (adapter->flags2 & FLAG2_HAS_EEE)
        adapter->eee_advert = MDIO_EEE_100TX | MDIO_EEE_1000T;

    /* construct the net_device struct */
    //设置net_device的操作回调和ethtool回调
    //这步是关键，C语言就喜欢回调，回调，回调
    netdev->netdev_ops = &e1000e_netdev_ops;
    e1000e_set_ethtool_ops(netdev);
    netdev->watchdog_timeo = 5 * HZ;
    //关联net_device于napi，napi结合了中断与轮询，可以提高网络收包效益，
现在默认使用NAPI了
```

```
netif_napi_add(netdev, &adapter->napi, e1000e_poll, 64);
strlcpy(netdev->name, pci_name(pdev), sizeof(netdev->name));
//记录一下之前获取的PCI bar 0的物理地址范围
netdev->mem_start = mmio_start;
netdev->mem_end = mmio_start + mmio_len;

adapter->bd_number = cards_found++;

e1000e_check_options(adapter);

/* setup adapter struct */
//adapter还有东西需要初始化
err = e1000_sw_init(adapter);
if (err)
    goto err_sw_init;
//对类型网卡，其操作回调为
//e1000_82574_info.mac_ops
//.mac_ops = &e82571_mac_ops,
//.phy_ops = &e82_phy_ops_bm,
//.nvm_ops = &e82571_nvm_ops,
memcpy(&hw->mac.ops, ei->mac_ops, sizeof(hw->mac.ops));
memcpy(&hw->nvm.ops, ei->nvm_ops, sizeof(hw->nvm.ops));
memcpy(&hw->phy.ops, ei->phy_ops, sizeof(hw->phy.ops));

err = ei->get_variants(adapter);
if (err)
    goto err_hw_init;

if ((adapter->flags & FLAG_IS_ICH) &&
    (adapter->flags & FLAG_READ_ONLY_NVM))
    e1000e_write_protect_nvm_ich8lan(&adapter->hw);

hw->mac.ops.get_bus_info(&adapter->hw);

adapter->hw.phy.autoneg_wait_to_complete = 0;

/* Copper options */
if (adapter->hw.phy.media_type == e1000_media_type_copper) {
    adapter->hw.phy.mdix = AUTO_ALL_MODES;
    adapter->hw.phy.disable_polarity_correction = 0;
    adapter->hw.phy.ms_type = e1000_ms_hw_default;
}

if (hw->phy.ops.check_reset_block &&
```

```
hw->phy.ops.check_reset_block(hw))
        dev_info(&pdev->dev,
             "PHY reset is blocked due to SOL/IDER session.\n");

    /* Set initial default active device features */
    netdev->features = (NETIF_F_SG |
                NETIF_F_HW_VLAN_CTAG_RX |
                NETIF_F_HW_VLAN_CTAG_TX |
                NETIF_F_TSO |
                NETIF_F_TSO6 |
                NETIF_F_RXHASH |
                NETIF_F_RXCSUM |
                NETIF_F_HW_CSUM);

    /* Set user-changeable features (subset of all device features)
*/
    netdev->hw_features = netdev->features;
    netdev->hw_features |= NETIF_F_RXFCS;
    netdev->priv_flags |= IFF_SUPP_NOFCS;
    netdev->hw_features |= NETIF_F_RXALL;

    if (adapter->flags & FLAG_HAS_HW_VLAN_FILTER)
        netdev->features |= NETIF_F_HW_VLAN_CTAG_FILTER;

    netdev->vlan_features |= (NETIF_F_SG |
                 NETIF_F_TSO |
                 NETIF_F_TSO6 |
                 NETIF_F_HW_CSUM);

    netdev->priv_flags |= IFF_UNICAST_FLT;

    if (pci_using_dac) {
        netdev->features |= NETIF_F_HIGHDMA;
        netdev->vlan_features |= NETIF_F_HIGHDMA;
    }

    if (e1000e_enable_mng_pass_thru(&adapter->hw))
        adapter->flags |= FLAG_MNG_PT_ENABLED;

    /* before reading the NVM, reset the controller to
     * put the device in a known good starting state
     */
    adapter->hw.mac.ops.reset_hw(&adapter->hw);
```

```
	/* systems with ASPM and others may see the checksum fail on the
first
	 * attempt. Let's give it a few tries
	 */
	for (i = 0;; i++) {
		if (e1000_validate_nvm_checksum(&adapter->hw) >= 0)
			break;
		if (i == 2) {
			dev_err(&pdev->dev, "The NVM Checksum Is Not Valid\n");
			err = -EIO;
			goto err_eeprom;
		}
	}

	e1000_eeprom_checks(adapter);

	/* copy the MAC address */
	if (e1000e_read_mac_addr(&adapter->hw))
		dev_err(&pdev->dev,
			"NVM Read Error while reading MAC address\n");

	memcpy(netdev->dev_addr, adapter->hw.mac.addr, netdev->addr_len);

	if (!is_valid_ether_addr(netdev->dev_addr)) {
		dev_err(&pdev->dev, "Invalid MAC Address: %pM\n",
			netdev->dev_addr);
		err = -EIO;
		goto err_eeprom;
	}

	init_timer(&adapter->watchdog_timer);
	//触发调用e1000_watchdog_task 这个task主要负责更新网口状态变化
	adapter->watchdog_timer.function = e1000_watchdog;
	adapter->watchdog_timer.data = (unsigned long)adapter;

	init_timer(&adapter->phy_info_timer);
	//触发调用e1000e_update_phy_task -> get_info ->
e82_phy_ops_bm.get_info = e1000e_get_phy_info_m88 ->
	adapter->phy_info_timer.function = e1000_update_phy_info;
	adapter->phy_info_timer.data = (unsigned long)adapter;

	INIT_WORK(&adapter->reset_task, e1000_reset_task);
	INIT_WORK(&adapter->watchdog_task, e1000_watchdog_task);
	INIT_WORK(&adapter->downshift_task, e1000e_downshift_workaround);
```

```
    INIT_WORK(&adapter->update_phy_task, e1000e_update_phy_task);
    INIT_WORK(&adapter->print_hang_task, e1000_print_hw_hang);
    . . .
    //从下面这个strlcpy可以看出，每个支持e1000e的网口初始化，都会调用这个
函数一次
    strlcpy(netdev->name, "eth%d", sizeof(netdev->name));
    //初始化netdev这么多参数，终于初始化完成，可以register netdev了
    err = register_netdev(netdev);
    if (err)
        goto err_register;
    //netmap attach的位置，在netdev初始化完成后
#ifdef DEV_NETMAP
    e1000_netmap_attach(adapter);
#endif /* DEV_NETMAP */
    /* carrier off reporting is important to ethtool even BEFORE open
*/
    netif_carrier_off(netdev);//驱动prob后，还不能工作

    /* init PTP hardware clock */
    e1000e_ptp_init(adapter);
    /*
     这里打印网卡设备的基本信息
     例如：
     irq 54 for MSI/MSI-X
     eth4: (PCI Express:2.5GT/s:Width x4) 00:1b:21:98:e5:9e
     eth4: Intel(R) PRO/1000 Network Connection
     eth4: MAC: 0, PHY: 4, PBA No: D72468-005
     */
    e1000_print_device_info(adapter);

    if (pci_dev_run_wake(pdev))
        pm_runtime_put_noidle(&pdev->dev);

    return 0;
    . . .
}
```

初始化网卡是 probe，其对应位删除网卡 remove

```
static void e1000_remove(struct pci_dev *pdev)
{//e1000_remove 对应e1000_probe
    struct net_device *netdev = pci_get_drvdata(pdev);
    struct e1000_adapter *adapter = netdev_priv(netdev);
    bool down = test_bit(__E1000_DOWN, &adapter->state);
```

```
e1000e_ptp_remove(adapter);

/* The timers may be rescheduled, so explicitly disable them
 * from being rescheduled.
 */
if (!down)
    set_bit(__E1000_DOWN, &adapter->state);
del_timer_sync(&adapter->watchdog_timer);
del_timer_sync(&adapter->phy_info_timer);

cancel_work_sync(&adapter->reset_task);
cancel_work_sync(&adapter->watchdog_task);
cancel_work_sync(&adapter->downshift_task);
cancel_work_sync(&adapter->update_phy_task);
cancel_work_sync(&adapter->print_hang_task);

if (adapter->flags & FLAG_HAS_HW_TIMESTAMP) {
    cancel_work_sync(&adapter->tx_hwtstamp_work);
    if (adapter->tx_hwtstamp_skb) {
        dev_kfree_skb_any(adapter->tx_hwtstamp_skb);
        adapter->tx_hwtstamp_skb = NULL;
    }
}

//难道UP的时候就不需要power down？应该是流程保证up的时候无法remove
if (!(netdev->flags & IFF_UP))
    e1000_power_down_phy(adapter);

/* Don't lie to e1000_close() down the road. */
if (!down)
    clear_bit(__E1000_DOWN, &adapter->state);
unregister_netdev(netdev);

if (pci_dev_run_wake(pdev))
    pm_runtime_get_noresume(&pdev->dev);

/* Release control of h/w to f/w.  If f/w is AMT enabled, this
 * would have already happened in close and is redundant.
 */
e1000e_release_hw_control(adapter);

e1000e_reset_interrupt_capability(adapter);
kfree(adapter->tx_ring);
```

```c
    kfree(adapter->rx_ring);

#ifdef DEV_NETMAP
    //e1000_probe的时候，调用了e1000_netmap_attach，在remove的时候，调用netmap_detach，用于释放attach分配的资源
    //释放ring，释放adapter，netmap的adapter和e1000e的adapter不是存放到同一个地方的
    //struct netmap_adapter *na = (struct netmap_adapter *)netdev->ax25_ptr;
    //struct e1000_adapter *adapter = netdev_priv(netdev);
    //所以free了netmap_adapter不影响e1000_adapter继续使用
    netmap_detach(netdev);
#endif /* DEV_NETMAP */

    iounmap(adapter->hw.hw_addr);
    if (adapter->hw.flash_address)
        iounmap(adapter->hw.flash_address);
    pci_release_selected_regions(pdev,
                 pci_select_bars(pdev, IORESOURCE_MEM));

    free_netdev(netdev);

    /* AER disable */
    pci_disable_pcie_error_reporting(pdev);

    pci_disable_device(pdev);
}
```

网口的 UP 与 DOWN

在 probe 函数里，定义了 ops 回调，里面有设备的 open，stop，发包 start_xmit 等：

```c
static const struct net_device_ops e1000e_netdev_ops = {
    .ndo_open       = e1000_open,
    .ndo_stop       = e1000_close,
    .ndo_start_xmit     = e1000_xmit_frame,
    .ndo_get_stats64  = e1000e_get_stats64,
    .ndo_set_rx_mode  = e1000e_set_rx_mode,
    .ndo_set_mac_address = e1000_set_mac,
    .ndo_change_mtu     = e1000_change_mtu,
    .ndo_do_ioctl     = e1000_ioctl,
    .ndo_tx_timeout     = e1000_tx_timeout,
    .ndo_validate_addr    = eth_validate_addr,
```

```
    .ndo_vlan_rx_add_vid = e1000_vlan_rx_add_vid,
    .ndo_vlan_rx_kill_vid    = e1000_vlan_rx_kill_vid,
#ifdef CONFIG_NET_POLL_CONTROLLER
    //在关中断的情况下发送数据包，例如使用netconsole发送printk到其他host
    .ndo_poll_controller = e1000_netpoll,
#endif
    .ndo_set_features = e1000_set_features,
};
```

不要忘记，调用e1000e_probe的时候，网口处于down状态，你需要ifconfig eth up
此时，会回调 dev_change_flags->__dev_open->ndo_open->e1000_open

对应地，当ifconfig eth down的时候，会回调
dev_change_flags->__dev_close->__dev_close_many->ndo_stop->e1000_close

```
static int e1000_open(struct net_device *netdev)
{
    struct e1000_adapter *adapter = netdev_priv(netdev);
    struct e1000_hw *hw = &adapter->hw;
    struct pci_dev *pdev = adapter->pdev;
    int err;

    /* disallow open during test */
    if (test_bit(__E1000_TESTING, &adapter->state))
        return -EBUSY;

    pm_runtime_get_sync(&pdev->dev);
    //确保处于no carrier状态
    netif_carrier_off(netdev);

    /* allocate transmit descriptors */
    //ring的slot是在这里分配的，每次up网口的时候
    err = e1000e_setup_tx_resources(adapter->tx_ring);
    if (err)
        goto err_setup_tx;

    /* allocate receive descriptors */
    err = e1000e_setup_rx_resources(adapter->rx_ring);
    if (err)
        goto err_setup_rx;

    /* If AMT is enabled, let the firmware know that the network
     * interface is now open and reset the part to a known state.
     */
```

```
    if (adapter->flags & FLAG_HAS_AMT) {
        e1000e_get_hw_control(adapter);
        e1000e_reset(adapter);
    }
    //确保通电
    e1000e_power_up_phy(adapter);

    //配置vlan id
    adapter->mng_vlan_id = E1000_MNG_VLAN_NONE;
    if ((adapter->hw.mng_cookie.status &
E1000_MNG_DHCP_COOKIE_STATUS_VLAN))
        e1000_update_mng_vlan(adapter);

    /* DMA latency requirement to workaround jumbo issue */
    pm_qos_add_request(&adapter->netdev->pm_qos_req,
PM_QOS_CPU_DMA_LATENCY,
               PM_QOS_DEFAULT_VALUE);

    /* before we allocate an interrupt, we must be ready to handle
it.
     * Setting DEBUG_SHIRQ in the kernel makes it fire an interrupt
     * as soon as we call pci_request_irq, so we have to setup our
     * clean_rx handler before we do so.
     */
    //执行了alloc_rx_buf回调，用于分配rx skb的内存
    e1000_configure(adapter);

    //初始化IRQ
    err = e1000_request_irq(adapter);
    if (err)
        goto err_req_irq;

    /* Work around PCIe errata with MSI interrupts causing some
chipsets to
     * ignore e1000e MSI messages, which means we need to test our
MSI
     * interrupt now
     */
    if (adapter->int_mode != E1000E_INT_MODE_LEGACY) {
        err = e1000_test_msi(adapter);
        if (err) {
            e_err("Interrupt allocation failed\n");
            goto err_req_irq;
        }
```

```c
    }

    //开始up网口
    /* From here on the code is the same as e1000e_up() */
    clear_bit(__E1000_DOWN, &adapter->state);

    //允许NAPI调度
    napi_enable(&adapter->napi);

    e1000_irq_enable(adapter);

    adapter->tx_hang_recheck = false;
    //允许上层调用设备的hard_start_xmit routine函数，开始数据传送
    netif_start_queue(netdev);

#ifdef DEV_NETMAP
    //此netdev上的所有ring开始工作，类似netif_start_queue
    netmap_enable_all_rings(netdev);
#endif /* DEV_NETMAP */

    adapter->idle_check = true;
    hw->mac.get_link_status = true;
    pm_runtime_put(&pdev->dev);

    /* fire a link status change interrupt to start the watchdog */
    if (adapter->msix_entries)
        ew32(ICS, E1000_ICS_LSC | E1000_ICR_OTHER);
    else
        ew32(ICS, E1000_ICS_LSC);

    return 0;
    . . .
}

int e1000e_setup_rx_resources(struct e1000_ring *rx_ring)
{
    struct e1000_adapter *adapter = rx_ring->adapter;
    struct e1000_buffer *buffer_info;
    int i, size, desc_len, err = -ENOMEM;


    size = sizeof(struct e1000_buffer) * rx_ring->count;
    //分配buffer info，用于关联desc，分配的个数rx_ring->count默认为，使
用的是vzalloc，allocate virtually continguos memory with zero fill
```

```
    rx_ring->buffer_info = vzalloc(size);
    if (!rx_ring->buffer_info)
        goto err;
    //为支持packet split，需要初始化e1000_ps_page结构，用于收包
    for (i = 0; i < rx_ring->count; i++) {
        buffer_info = &rx_ring->buffer_info[i];
        buffer_info->ps_pages = kcalloc(PS_PAGE_BUFFERS,
                        sizeof(struct e1000_ps_page),
                        GFP_KERNEL);
        if (!buffer_info->ps_pages)
            goto err_pages;
    }
    //可能装两个结构体，一是e1000_tx_desc，一是
e1000_rx_desc_packet_split，但前者要小，所以分配大的，可以兼容小的
    desc_len = sizeof(union e1000_rx_desc_packet_split);

    /* Round up to nearest 4K */
    rx_ring->size = rx_ring->count * desc_len;
    rx_ring->size = ALIGN(rx_ring->size, 4096);
    //分配desc
    err = e1000_alloc_ring_dma(adapter, rx_ring);
    if (err)
        goto err_pages;

    rx_ring->next_to_clean = 0;
    rx_ring->next_to_use = 0;
    rx_ring->rx_skb_top = NULL;

    return 0;
    . . .
}

static int e1000_request_irq(struct e1000_adapter *adapter)
{
    struct net_device *netdev = adapter->netdev;
    int err;
    /*
    #define E1000E_INT_MODE_LEGACY     0
    #define E1000E_INT_MODE_MSI        1
    #define E1000E_INT_MODE_MSIX       2
     eth0为E1000E_INT_MODE_MSI模式
    # cat /proc/interrupts | grep eth0
    46:      91341          0          0          0  IR-PCI-MSI-edge       eth0
```

```
 eth1为E1000E_INT_MODE_MSIX模式
# cat /proc/interrupts | grep eth1
47:     103786          0          0          0  IR-PCI-MSI-edge      eth1-rx-0
48:          0          0          0          0  IR-PCI-MSI-edge      eth1-tx-0
49:         50          0          0          0  IR-PCI-MSI-edge      eth1
 */
    if (adapter->msix_entries) {
        //如果是MSIX方式，有rx, tx, other 3种中断
        err = e1000_request_msix(adapter);
        if (!err)
            return err;
        //我x，出错了，回滚到MSI
        /* fall back to MSI */
        e1000e_reset_interrupt_capability(adapter);
        adapter->int_mode = E1000E_INT_MODE_MSI;
        e1000e_set_interrupt_capability(adapter);
    }
    if (adapter->flags & FLAG_MSI_ENABLED) {
        //如果是MSI
        err = request_irq(adapter->pdev->irq, e1000_intr_msi, 0,
                  netdev->name, netdev);
        if (!err)
            return err;

        /* fall back to legacy interrupt */
        e1000e_reset_interrupt_capability(adapter);
        adapter->int_mode = E1000E_INT_MODE_LEGACY;
    }

    err = request_irq(adapter->pdev->irq, e1000_intr, IRQF_SHARED,
              netdev->name, netdev);
    if (err)
        e_err("Unable to allocate interrupt, Error: %d\n", err);

    return err;
}

void e1000e_down(struct e1000_adapter *adapter)
{
    struct net_device *netdev = adapter->netdev;
    struct e1000_hw *hw = &adapter->hw;
    u32 tctl, rctl;

    /* signal that we're down so the interrupt handler does not
```

```
	 * reschedule our watchdog timer
	 */
	set_bit(__E1000_DOWN, &adapter->state);

	/* disable receives in the hardware */
	rctl = er32(RCTL);
	if (!(adapter->flags2 & FLAG2_NO_DISABLE_RX))
		ew32(RCTL, rctl & ~E1000_RCTL_EN);
	/* flush and sleep below */
	//不能发送数据了
	netif_stop_queue(netdev);

#ifdef DEV_NETMAP
	//down的时候，并不需要free啥，只是标记ring不能使用了
	netmap_disable_all_rings(netdev);
#endif

	/* disable transmits in the hardware */
	tctl = er32(TCTL);
	tctl &= ~E1000_TCTL_EN;
	ew32(TCTL, tctl);

	/* flush both disables and wait for them to finish */
	e1e_flush();
	usleep_range(10000, 20000);
	//关中断
	e1000_irq_disable(adapter);

	napi_synchronize(&adapter->napi);

	del_timer_sync(&adapter->watchdog_timer);
	del_timer_sync(&adapter->phy_info_timer);

	netif_carrier_off(netdev);

	spin_lock(&adapter->stats64_lock);
	e1000e_update_stats(adapter);
	spin_unlock(&adapter->stats64_lock);

	e1000e_flush_descriptors(adapter);
	//这里删除的是tx_ring, rx_ring内部的结构，本身并没有删除
	e1000_clean_tx_ring(adapter->tx_ring);
	e1000_clean_rx_ring(adapter->rx_ring);
```

```
    adapter->link_speed = 0;
    adapter->link_duplex = 0;

    /* Disable Si errata workaround on PCHx for jumbo frame flow */
    if ((hw->mac.type >= e1000_pch2lan) &&
        (adapter->netdev->mtu > ETH_DATA_LEN) &&
        e1000_lv_jumbo_workaround_ich8lan(hw, false))
        e_dbg("failed to disable jumbo frame workaround mode\n");

    if (!pci_channel_offline(adapter->pdev))
        e1000e_reset(adapter);

    /* TODO: for power management, we could drop the link and
     * pci_disable_device here.
     */
}
```

收发包队列

每一个adapter,都有一个tx_ring指针和rx_ring指针,其类型为struct e1000_ring。

```
struct e1000_adapter {
    /* Tx - one ring per active queue */
    struct e1000_ring *tx_ring ____cacheline_aligned_in_smp;
    /* Rx */
    struct e1000_ring *rx_ring;

    bool (*clean_rx) (struct e1000_ring *ring, int *work_done,
         int work_to_do) ____cacheline_aligned_in_smp;
    void (*alloc_rx_buf) (struct e1000_ring *ring, int cleaned_count,
         gfp_t gfp);

    //for ethtool
    struct e1000_ring test_tx_ring;
    struct e1000_ring test_rx_ring;
    //descriptor count, 描述符个数
    u16 tx_ring_count;
    u16 rx_ring_count;
};

struct e1000_ring {
    struct e1000_adapter *adapter;  /* back pointer to adapter */
    void *desc;           /* pointer to ring memory, 驱动能访问的描述符
队列首地址 */
```

```
    dma_addr_t dma;           /* phys address of ring, 对应desc, 可以用
来寻址硬件设备里的内存*/
    unsigned int size;        /* length of ring in bytes, desc的字节数
*/
    unsigned int count;       /* number of desc. in ring, 收发包队列的有
多少个描述符, 也就是槽, 初始为adapter->tx_ring_count/rx_ring_count*/

    u16 next_to_use;
    u16 next_to_clean;

    //读写head和tail位置的物理地址
    void __iomem *head;
    void __iomem *tail;

    /* array of buffer information structs */
    struct e1000_buffer *buffer_info;

    char name[IFNAMSIZ + 5];
    u32 ims_val;
    u32 itr_val;
    void __iomem *itr_register;
    int set_itr;
    //这个是收包时, 如果遇到分片包, 需要保存第一个包, 后续的包要连接到第一
个包, 所以叫top
    struct sk_buff *rx_skb_top;
};
```

```
int e1000_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
    ├── netdev->netdev_ops = &e1000e_netdev_ops 注册回调
    |       包含.ndo_open = e1000_open
    └── int e1000_sw_init(struct e1000_adapter *adapter)
            └── int e1000_alloc_queues(struct e1000_adapter *adapter)
                    ├── adapter->tx_ring = kzalloc(sizeof(struct e1000_ring), GFP_KERNEL)
                    |       从这个大小可以看出e1000e每一个adapter下面, 有且只有一个tx_ring和rx_ring
                    ├── tx_ring->count = adapter->tx_ring_count = E1000_DEFAULT_TXD = 256
                    └── tx_ring->adapter = adapter rx_ring也在这里初始化, 流程一样

```

```
int e1000_open(struct net_device *netdev)
    ├── int e1000e_setup_tx_resources(struct e1000_ring *tx_ring = adapter->tx_ring)
    |       ├── tx_ring->buffer_info = vzalloc(sizeof(struct e1000_buffer) * tx_ring->count) 分配buffer_info
    |       ├── tx_ring->size = ALIGN(tx_ring->count * sizeof(struct e1000_tx_desc), 4096)   设置desc的大小
    |       ├── tx_ring->desc = dma_alloc_coherent(&pdev->dev, tx_ring->size, &tx_ring->dma, GFP_KERNEL)
    |       |     刚才设置了desc的大小，现在建立一致性映射，分配tx_ring->desc给驱动使用
    |       |     总线地址tx_ring->dma为设备能访问的地址
    |       ├── tx_ring->next_to_use = 0;
    |       └── tx_ring->next_to_clean = 0;
    ├── int e1000e_setup_rx_resources(struct e1000_ring *tx_ring = adapter->rx_ring)
    |       └── rx_ring->rx_skb_top = NULL 收包队列和发包队列是离不开的好基友，
    |            初始化都差不多，只是，多初始化了一些特有变量，例如rx_skb_top，buffer_info[i]->ps_pages
    └── void e1000_configure(struct e1000_adapter *adapter)
            ├── void e1000_configure_tx(struct e1000_adapter *adapter)
            |       ├── tx_ring->head = adapter->hw.hw_addr + E1000_TDH(0)
            |       └── tx_ring->tail = adapter->hw.hw_addr + E1000_TDT(0)
            |             head和tail用于同步设备发包其实和结束位置
            ├── void e1000_configure_rx(struct e1000_adapter *adapter)
            |     除了和tx_ring类似设置head和tail外，还要根据不同的条件设置clean_rx和alloc_rx_buf，例如
            |       ├── adapter->clean_rx = e1000_clean_jumbo_rx_irq，从网卡里取出接收成功的包，
            |       |    递交给上层协议栈，清除对应slot的desc
            |       └── adapter->alloc_rx_buf = e1000_alloc_jumbo_rx_buffers，为网卡的可用slot分配内存
            └── adapter->alloc_rx_buf(rx_ring, e1000_desc_unused(rx_ring), GFP_KERNEL) 收包需要提前为网卡分配内存
                    这里e1000_desc_unused(rx_ring) = ring->count+ring->next_to_clean-ring->next_to_use-1
                    因为count=256, next_to_clean=next_to_use=0, 计算出来应该是255而非256
```

除了 e1000_configure 第一次会调用 alloc_rx_buf 来初始化内存外，clean_rx 里也会调用 alloc_rx_buf 里补充内存。

clean_rx 又在哪里调用的呢？假设收 64 个包，weight=64

```
int e1000e_poll(struct napi_struct *napi, int weight)
{
    int work_done = 0;
    adapter->clean_rx(adapter->rx_ring, &work_done, weight);
}
```

收包

收包触发

一种是收到中断后调用 e1000e_poll，一种是通过 napi 直接轮询，事实上，e1000e 在收到中断后，也是调用的 napi 轮询，以 MSI 中断为例：

```
static irqreturn_t e1000_intr_msi(int __always_unused irq, void
*data)
{
    . . .

    //这里，启用napi sched
    if (napi_schedule_prep(&adapter->napi)) {
        adapter->total_tx_bytes = 0;
        adapter->total_tx_packets = 0;
        adapter->total_rx_bytes = 0;
        adapter->total_rx_packets = 0;
        __napi_schedule(&adapter->napi);
```

```
    }

    return IRQ_HANDLED;
}
```

如果没有打上 NAPI_STATE_DISABLE 标记，也没有打上 NAPI_STATE_SCHED 标记，才 OK。

```
static inline bool napi_schedule_prep(struct napi_struct *n)
{
    return !napi_disable_pending(n) &&
        !test_and_set_bit(NAPI_STATE_SCHED, &n->state);
}

void __napi_schedule(struct napi_struct *n)
{
    unsigned long flags;

    local_irq_save(flags);//禁止中断，并记住之前的状态
    ____napi_schedule(&__get_cpu_var(softnet_data), n);
    local_irq_restore(flags);//恢复中断
}

static inline void ____napi_schedule(struct softnet_data *sd,
struct napi_struct *napi)
{
    //把自己挂到per cpu的softnet_data上，触发NET_RX_SOFTIRQ软中断
    list_add_tail(&napi->poll_list, &sd->poll_list);
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

在 net_dev_init 的时候，你会看到

```
open_softirq(NET_TX_SOFTIRQ, net_tx_action);
open_softirq(NET_RX_SOFTIRQ, net_rx_action);
```

那么，软中断触发函数为 net_rx_action

```
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;
    zcm_wakeup_fn_t zcm_wakeup_ptr = NULL;

    local_irq_disable();
```

```c
    //遍历之前挂在在这上面的napi_struct
    while (!list_empty(&sd->poll_list)) {
        struct napi_struct *n;
        int work, weight;

        /* If softirq window is exhuasted then punt.
         * Allow this to run for 2 jiffies since which will allow
         * an average latency of 1.5/HZ.
         */
        if (unlikely(budget <= 0 || time_after_eq(jiffies,
time_limit)))
            goto softnet_break;

        local_irq_enable();

        /* Even though interrupts have been re-enabled, this
         * access is safe because interrupts can only add new
         * entries to the tail of this list, and only ->poll()
         * calls can remove this head entry from the list.
         */
        n = list_first_entry(&sd->poll_list, struct napi_struct,
poll_list);

        have = netpoll_poll_lock(n);

        //这个权重，表示一次最多可以处理多少个
        weight = n->weight;

        /* This NAPI_STATE_SCHED test is for avoiding a race
         * with netpoll's poll_napi().  Only the entity which
         * obtains the lock and sees NAPI_STATE_SCHED set will
         * actually make the ->poll() call.  Therefore we avoid
         * accidentally calling ->poll() when NAPI is not scheduled.
         */
        work = 0;
        if (test_bit(NAPI_STATE_SCHED, &n->state)) {
            //还检查一下这个标记，没有打NAPI_STATE_SCHED就不调用poll
            rcu_read_lock();
            //通用网卡处理方案
            zcm_wakeup_ptr = rcu_dereference(zcm_wakeup_fn);
            if (zcm_poll_enabled && zcm_wakeup_ptr
                && n->dev && n->dev->vt_zcm_ptr) {
                work = zcm_wakeup_ptr(n->dev->vt_zcm_ptr, n);
```

```
            } else {
                //在e1000_probe的时候，已经注册了poll=e1000e_poll,
weight=64
                //netif_napi_add(netdev, &adapter->napi, e1000e_poll,
64);
                work = n->poll(n, weight);//回调poll
                trace_napi_poll(n);
            }
            rcu_read_unlock();
        }

        WARN_ON_ONCE(work > weight);
        budget -= work;
        local_irq_disable();

        . . .

        netpoll_poll_unlock(have);
    }
    . . .
}
```

再回到 e1000e 驱动：

```
static int e1000e_poll(struct napi_struct *napi, int weight)
{
    struct e1000_adapter *adapter = container_of(napi, struct
e1000_adapter,
                                napi);
    struct e1000_hw *hw = &adapter->hw;
    struct net_device *poll_dev = adapter->netdev;
    int tx_cleaned = 1, work_done = 0;//这里work_done要置，clean_rx回
调里面没置

    adapter = netdev_priv(poll_dev);

    if (test_bit(__E1000_DOWN, &adapter->state))
    {
        zcm_nic_t *nic = nmd_nic(poll_dev);
        return 0;
    }

    if (!adapter->msix_entries ||
        (adapter->rx_ring->ims_val & adapter->tx_ring->ims_val))
```

```
        tx_cleaned = e1000_clean_tx_irq(adapter->tx_ring);//这里，查看
并回收tx slot

    //我们在e1000_configure_rx，注册了，将已经done的data，从网卡弄到内核
去
    //adapter->clean_rx = e1000_clean_rx_irq;
    //adapter->alloc_rx_buf = e1000_alloc_rx_buffers;
    adapter->clean_rx(adapter->rx_ring, &work_done, weight);

    if (!tx_cleaned)
        work_done = weight;

    /* If weight not fully consumed, exit the polling mode */
    if (work_done < weight) {
        if (adapter->itr_setting & 3)
            e1000_set_itr(adapter);//根据当前的数据量，动态调节iterrupt
的间隙
        napi_complete(napi);
        if (!test_bit(__E1000_DOWN, &adapter->state)) {
            if (adapter->msix_entries)
                ew32(IMS, adapter->rx_ring->ims_val);
            else
                e1000_irq_enable(adapter);//恢复IRQ
        }
    }

    return work_done;
}
```

我们分析一下巨帧的分配，类似 e1000_alloc_rx_buffers：

```
static void e1000_alloc_jumbo_rx_buffers(struct e1000_ring *rx_ring,
                    int cleaned_count, gfp_t gfp)
{
    struct e1000_adapter *adapter = rx_ring->adapter;
    struct net_device *netdev = adapter->netdev;
    struct pci_dev *pdev = adapter->pdev;
    union e1000_rx_desc_extended *rx_desc;
    struct e1000_buffer *buffer_info;
    struct sk_buff *skb;
    unsigned int i;
    unsigned int bufsz = 256 - 16;  /* for skb_reserve */
    //最开始的时候，next_to_use为，cleaned_count = 255且条件为先取值再--
的(cleaned_count--)，所以，要循环次
    i = rx_ring->next_to_use;//recv分配的时候，是为next to use分配，而
```

```
next to clean为已经有数据包了的
    buffer_info = &rx_ring->buffer_info[i];

    while (cleaned_count--) {//一次性分配的最大个数

        skb = buffer_info->skb;//尽可能回收利用，例如drop的包，可回收利
用
        if (skb) {
            skb_trim(skb, 0);
            goto check_page;
        }

        skb = __netdev_alloc_skb_ip_align(netdev, bufsz, gfp);
        if (unlikely(!skb)) {
            /* Better luck next round */
            adapter->alloc_rx_buff_failed++;
            break;
        }

        buffer_info->skb = skb;//其实SKB在网卡收包的时候用不到
check_page:
        /* allocate a new page if necessary */
        //e1000e驱动访问的是buffer_info->page，设备访问的是
buffer_info->dma
        if (!buffer_info->page) {
            buffer_info->page = alloc_page(gfp);
            if (unlikely(!buffer_info->page)) {//分配失败
                adapter->alloc_rx_buff_failed++;
                break;
            }
        }

        if (!buffer_info->dma) {
            buffer_info->dma = dma_map_page(&pdev->dev,
                            buffer_info->page, 0,
                            PAGE_SIZE,
                            DMA_FROM_DEVICE);
            if (dma_mapping_error(&pdev->dev, buffer_info->dma)) {
                adapter->alloc_rx_buff_failed++;
                break;
            }
        }

        rx_desc = E1000_RX_DESC_EXT(*rx_ring, i);
```

```
        //最终地址赋值给描述符的read.buffer_addr, 才表示此个slot可以收包
        rx_desc->read.buffer_addr = cpu_to_le64(buffer_info->dma);
        //最后一次, i=254, clean_count为, 这时候, 已经循环了次, 接下来
++i, i=255, 但是, 还没到, 所以, break出去后i=255, clean_count=0,
buffer_info[i=255]
        if (unlikely(++i == rx_ring->count))
            i = 0;
        buffer_info = &rx_ring->buffer_info[i];
    }
    //第一次进来, 索引为的描述符, 没有初始化
    if (likely(rx_ring->next_to_use != i)) {
        //表示又准备好内存, next_to_use指向设备不能用的内存的开始位置
        rx_ring->next_to_use = i;
        //这里i--获取最后一个索引, 因为上面的i是多了一位的, 本来个槽, 但初
始化只初始化了[0,254]=255个, 而i为并不表示最后一个, i-1才表示最后一个
        if (unlikely(i-- == 0))
            i = (rx_ring->count - 1);
        //对rx_ring来说, rx_ring->next_to_use-1表示tail
        //此时, 收包队列里的tail对网卡来说, 表示可以将数据包写入到<=tail指
示描述符位置
        /* Force memory writes to complete before letting h/w
         * know there are new descriptors to fetch.  (Only
         * applicable for weak-ordered memory model archs,
         * such as IA-64).
         */
        wmb();
        if (adapter->flags2 & FLAG2_PCIM2PCI_ARBITER_WA)
            e1000e_update_rdt_wa(rx_ring, i);
        else
            writel(i, rx_ring->tail);
    }
}
```

从网卡到内核

```
static bool e1000_clean_rx_irq(struct e1000_ring *rx_ring, int
*work_done,
                int work_to_do)
{
    struct e1000_adapter *adapter = rx_ring->adapter;
    struct net_device *netdev = adapter->netdev;
```

```c
    struct pci_dev *pdev = adapter->pdev;
    struct e1000_hw *hw = &adapter->hw;
    union e1000_rx_desc_extended *rx_desc, *next_rxd;
    struct e1000_buffer *buffer_info, *next_buffer;
    u32 length, staterr;
    unsigned int i;
    int cleaned_count = 0;
    bool cleaned = false;
    unsigned int total_rx_bytes = 0, total_rx_packets = 0;

#ifdef DEV_NETMAP
    //这里如果返回非，就表示netmap已经处理了，如果返回，表示走以前e1000e的逻辑
    if (netmap_rx_irq(netdev, 0, work_done))
        return 1; /* seems to be ignored */
#endif /* DEV_NETMAP */
    i = rx_ring->next_to_clean;
    rx_desc = E1000_RX_DESC_EXT(*rx_ring, i);
    staterr = le32_to_cpu(rx_desc->wb.upper.status_error);
    buffer_info = &rx_ring->buffer_info[i];
    //这里一个循环，从next_to_clean开始，将每个Rx slot上的包状态信息取出来，如果打上了E1000_RXD_STAT_DD(Descriptor Done标记)，就表示可以传送到上层协议栈
    while (staterr & E1000_RXD_STAT_DD) {
        struct sk_buff *skb;

        if (*work_done >= work_to_do)
            break;
        (*work_done)++;
        rmb(); /* read descriptor and rx_buffer_info after status DD */

        skb = buffer_info->skb;
        buffer_info->skb = NULL;//上层托管skb并清除原slot上的skb指针

        i++;
        if (i == rx_ring->count)
            i = 0;
        //取下一个slot
        next_rxd = E1000_RX_DESC_EXT(*rx_ring, i);
        prefetch(next_rxd);

        next_buffer = &rx_ring->buffer_info[i];
```

```
        cleaned = true;
        cleaned_count++;
        //给设备的DMA地址也要解开
        //这个地址在哪儿映射的？参考e1000_alloc_rx_buffers
        dma_unmap_single(&pdev->dev, buffer_info->dma,
                adapter->rx_buffer_len, DMA_FROM_DEVICE);
        buffer_info->dma = 0;
        prefetch(skb->data - NET_IP_ALIGN);

        length = le16_to_cpu(rx_desc->wb.upper.length);

        /* !EOP means multiple descriptors were used to store a
single
         * packet, if that's the case we need to toss it.  In fact,
we
         * need to toss every packet with the EOP bit clear and the
         * next frame that _does_ have the EOP bit set, as it is by
         * definition only a frame fragment
         */
        if (unlikely(!(staterr & E1000_RXD_STAT_EOP)))//如果没有打上
End of Packet标记，这个包要丢弃
            adapter->flags2 |= FLAG2_IS_DISCARDING;
        //不支持多个Fragement
        if (adapter->flags2 & FLAG2_IS_DISCARDING) {
            /* All receives must fit into a single buffer */
            e_dbg("Receive packet consumed multiple buffers\n");
            /* recycle */
            buffer_info->skb = skb;//skb保留，分配的时候就不用重新分配内
存了
            if (staterr & E1000_RXD_STAT_EOP)
                adapter->flags2 &= ~FLAG2_IS_DISCARDING;
            goto next_desc;
        }

        //包有错误，且net_device不接收错误包
        if (unlikely((staterr & E1000_RXDEXT_ERR_FRAME_ERR_MASK) &&
                 !(netdev->features & NETIF_F_RXALL))) {
            /* recycle */
            buffer_info->skb = skb;
            goto next_desc;
        }

        /* adjust length to remove Ethernet CRC */
        if (!(adapter->flags2 & FLAG2_CRC_STRIPPING)) {
```

```
            /* If configured to store CRC, don't subtract FCS,
             * but keep the FCS bytes out of the total_rx_bytes
             * counter
             */
            if (netdev->features & NETIF_F_RXFCS)
                total_rx_bytes -= 4;//你们也用魔数啊？？？要不要过
CHECKLIST，要不要被罚款？？？
            else
                length -= 4;
        }

        total_rx_bytes += length;
        total_rx_packets++;

        /* code added for copybreak, this should improve
         * performance for small packets with large amounts
         * of reassembly being done in the stack
         */
        if (length < copybreak) {
            //包长小于copybreak(当前宏为)的，COPY到小内存给上层，不用浪费
内存
            struct sk_buff *new_skb =
                netdev_alloc_skb_ip_align(netdev, length);
            if (new_skb) {
                skb_copy_to_linear_data_offset(new_skb,
                                   -NET_IP_ALIGN,
                                   (skb->data -
                                NET_IP_ALIGN),
                                   (length +
                                NET_IP_ALIGN));
                /* save the skb in buffer_info as good */
                buffer_info->skb = skb;
                skb = new_skb;
            }
            /* else just continue with the old one */
        }
        /* end copybreak code */
        skb_put(skb, length);//设置skb数据长度，增加length

        //检查skb check sum是否正确
        /* Receive Checksum Offload */
        e1000_rx_checksum(adapter, staterr, skb);
        //rxhash，涉及到将包分发到哪个cpu
        e1000_rx_hash(netdev, rx_desc->wb.lower.hi_dword.rss, skb);
```

```
        e1000_receive_skb(adapter, netdev, skb, staterr,
                  rx_desc->wb.upper.vlan);

next_desc:
        rx_desc->wb.upper.status_error &= cpu_to_le32(~0xFF);

        /* return some buffers to hardware, one at a time is too slow
*/
        if (cleaned_count >= E1000_RX_BUFFER_WRITE) {
            adapter->alloc_rx_buf(rx_ring, cleaned_count,
                          GFP_ATOMIC);
            cleaned_count = 0;
        }

        /* use prefetched values */
        rx_desc = next_rxd;
        buffer_info = next_buffer;

        staterr = le32_to_cpu(rx_desc->wb.upper.status_error);
    }
    rx_ring->next_to_clean = i;//下次继续

    cleaned_count = e1000_desc_unused(rx_ring);
    if (cleaned_count)
        adapter->alloc_rx_buf(rx_ring, cleaned_count, GFP_ATOMIC);
    //累计计算
    adapter->total_rx_bytes += total_rx_bytes;
    adapter->total_rx_packets += total_rx_packets;
    return cleaned;
}

static void e1000_receive_skb(struct e1000_adapter *adapter,
                  struct net_device *netdev, struct sk_buff *skb,
                  u32 staterr, __le16 vlan)
{
    u16 tag = le16_to_cpu(vlan);
    //打上timestamp
    e1000e_rx_hwtstamp(adapter, staterr, skb);

    skb->protocol = eth_type_trans(skb, netdev);

    if (staterr & E1000_RXD_STAT_VP)//如果是VLANE还要打上vlan标记
        __vlan_hwaccel_put_tag(skb, htons(ETH_P_8021Q), tag);
```

```
    //napi收包
    napi_gro_receive(&adapter->napi, skb);
}
```

gro receive 会根据情况重组数据包，然后如果正常，调用 netif_receive_skb，到此数据包已经收到内核了，内核的传递为：

```
int netif_receive_skb(struct sk_buff *skb)
{
    int rc = NET_RX_SUCCESS;
    dp_rcv_skb_fn_t dp_rcv_skb_in_fn = NULL;

    //检查timestamp
    net_timestamp_check(netdev_tstamp_prequeue, skb);

    if (skb_defer_rx_timestamp(skb))
        return NET_RX_SUCCESS;
    . . .
    return __netif_receive_skb(skb);
}

static int __netif_receive_skb_core(struct sk_buff *skb, bool
pfmemalloc)
{
    struct packet_type *ptype, *pt_prev;
    rx_handler_func_t *rx_handler;
    struct net_device *orig_dev;
    struct net_device *null_or_dev;
    bool deliver_exact = false;
    int ret = NET_RX_DROP;
    __be16 type;

    net_timestamp_check(!netdev_tstamp_prequeue, skb);

    trace_netif_receive_skb(skb);

    /* if we've gotten here through NAPI, check netpoll */
    if (netpoll_receive_skb(skb))
        goto out;

    orig_dev = skb->dev;

    skb_reset_network_header(skb);
    if (!skb_transport_header_was_set(skb))
        skb_reset_transport_header(skb);
```

```
    skb_reset_mac_len(skb);

    pt_prev = NULL;

    rcu_read_lock();

another_round:
    skb->skb_iif = skb->dev->ifindex;

    __this_cpu_inc(softnet_data.processed);

    if (skb->protocol == cpu_to_be16(ETH_P_8021Q) ||
        skb->protocol == cpu_to_be16(ETH_P_8021AD)) {
        skb = vlan_untag(skb);
        if (unlikely(!skb))
            goto unlock;
    }
    if (pfmemalloc)
        goto skip_taps;

    //依次调用回调函数处理数据
    //先调用ptype_all的，使用与数据包是任意协议的情况
    list_for_each_entry_rcu(ptype, &ptype_all, list) {
        if (!ptype->dev || ptype->dev == skb->dev) {
            //所有pt_prev的写法都是后赋值型，所以他叫prev
            if (pt_prev)
                ret = deliver_skb(skb, pt_prev, orig_dev);
            pt_prev = ptype;
        }
    }

skip_taps:

    if (pfmemalloc && !skb_pfmemalloc_protocol(skb))
        goto drop;

    //如果有vlan
    if (vlan_tx_tag_present(skb)) {
        if (pt_prev) {
            //这一次pt_prev调用，是因为，前面留了最后一个ptype_all没有调用
            ret = deliver_skb(skb, pt_prev, orig_dev);
            pt_prev = NULL;
        }
        if (vlan_do_receive(&skb))
```

```
            goto another_round;//如果有需要修改skb，添加vlan头部，然后再
走一遍
        else if (unlikely(!skb))
            goto unlock;
    }

    /*这里开始处理设备的rx_handler回调，例如openvswitch，在netdev_create
时注册了netdev_frame_hook
        err = netdev_rx_handler_register(netdev_vport->dev,
netdev_frame_hook,vport);
        这种回调可以让流程不再往下继续，例如当netdev_frame_hook返回的
RX_HANDLER_CONSUMED的时候，就终止执行。
    */
    rx_handler = rcu_dereference(skb->dev->rx_handler);
    if (rx_handler) {
        if (pt_prev) {
            //这一次pt_prev调用，是因为，前面留了最后一个ptype_all没有调用
            ret = deliver_skb(skb, pt_prev, orig_dev);
            pt_prev = NULL;
        }
        switch (rx_handler(&skb)) {
        case RX_HANDLER_CONSUMED:
            ret = NET_RX_SUCCESS;
            goto unlock;
        case RX_HANDLER_ANOTHER:
            goto another_round;
        case RX_HANDLER_EXACT:
            deliver_exact = true;
        case RX_HANDLER_PASS:
            break;
        default:
            BUG();
        }
    }

    if (unlikely(vlan_tx_tag_present(skb))) {
        if (vlan_tx_tag_get_id(skb))
            skb->pkt_type = PACKET_OTHERHOST;
        /* Note: we might in the future use prio bits
         * and set skb->priority like in vlan_do_receive()
         * For the time being, just ignore Priority Code Point
         */
        skb->vlan_tci = 0;
    }
```

```
    /* deliver only exact match when indicated */
    null_or_dev = deliver_exact ? skb->dev : NULL;

    /*
        这里，根据skb协议类型，调用不同的回调，以ipv4为例，在inet_init里调用dev_add_pack(&ip_packet_type)注册了
        static struct packet_type ip_packet_type __read_mostly = {
            .type = cpu_to_be16(ETH_P_IP),
            .func = ip_rcv,
        };
        其func为ip_rcv，于是，遍历到之后，直接调用deliver_skb->.func->ip_rcv
    */
    type = skb->protocol;
    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;
        }
    }

    //之前为什么要留下一次pt_prev不调用呢
    if (pt_prev) {
        if (unlikely(skb_orphan_frags(skb, GFP_ATOMIC)))
            goto drop;
        else
            ret = pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
    } else {
drop:
        atomic_long_inc(&skb->dev->rx_dropped);
        kfree_skb(skb);
        /* Jamal, now you will not able to escape explaining
         * me how you were going to use this. :-)
         */
        ret = NET_RX_DROP;
    }

unlock:
```

```
    rcu_read_unlock();
out:
    return ret;
}
```

rx_ring 索引

head, tail 为闭区间索引，假设 slot 个数为 16。
next_to_use 是内存分配完毕，网卡可以将数据写入其描述符的最后下标。
next_to_clean 是网卡已经讲数据写入后，驱动从描述符中将数据取出的开始下标。

所以，next_to_clean 指示的位置，一定要 next_to_use 先初始化过。

空 代表没有分配内存
0 代表分配了内存但是没有数据
1 代表数据准备好

1. e1000e_setup_rx_resources - 初始化阶段

```
head=tail=0
|
v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
^
|
clean=use=0
```

2. e1000_configure->adapter.alloc_rx_buf - 第一次为每个描述符分配内存
参考 e1000_alloc_jumbo_rx_buffers

```
head=0                                    tail=14
|                                         |
v                                         v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |  |
|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
^                                            ^
|                                            |
clean=0                                      use=15
```

3. e1000e_poll->adapter.clean_rx - 开始检查数据包
参考 e1000_clean_jumbo_rx_irq
假设有 11 个 slot 已经 Descriptor Done 了。

```
head=0                                      tail=14
|                                           |
v                                           v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |0 |0 |0 |0 |  |
|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
^                                              ^
|                                              |
clean=0                                        use=15
```

假如循环了 8 次

```
head=0                                      tail=14
|                                           |
v                                           v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |1 |1 |1 |0 |0 |0 |0 |  |
|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
^                                              ^
|                                              |
clean=0                                        use=15
```

然后调用 alloc_rx_buf(8)来填充内存，参考 e1000_alloc_jumbo_rx_buffers

```
head=0            tail=6
|                 |
v                 v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|0 |0 |0 |0 |0 |0 |0 |  |1 |1 |1 |0 |0 |0 |0 |0 |
|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
^                    ^
|                    |
clean=0              use=7
```

由于 alloc_rx_buf 是由 clean_rx 调用的，alloc 的 slot 个数由 clean_rx 精确控制

use 指针不会超过还未取数据的 slot，中间留了一空隙，保证读写不会追尾

11 个 slot 都遍历完之后，会将 next_to_clean 赋值为 i，i 为第一个没有 Descriptor Done 的下标

例如，现在是第 12 位为未 done 的，那么 next_to_clean = i = 11

```
head=0            tail=6
|                 |
v                 v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|0 |0 |0 |0 |0 |0 |0 |  |  |  |  |0 |0 |0 |0 |0 |
|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                    ^           ^
                    |           |
                    use=7       clean=11
```

clean_rx 结尾的地方，还会 adapter->alloc_rx_buf(rx_ring, e1000_desc_unused(rx_ring), GFP_ATOMIC)

此时，e1000_desc_unused 计算出来的 unused = ring->next_to_clean - ring->next_to_use - 1 = 11 - 7 - 1 = 3

于是，调用完 alloc_rx_buf 后，会变为

```
head=0                        tail=10
|                             |
v                             v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |  |0 |0 |0 |0 |0 |
|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                ^
                                |
                                clean=use=11
```

发包

从内核到网卡

系统发包一般调用流程：

```
int dev_queue_xmit(struct sk_buff *skb)
{
    struct net_device *dev = skb->dev;
    struct netdev_queue *txq;
```

```
    struct Qdisc *q;
    int rc = -ENOMEM;
    dp_rcv_skb_fn_t dp_rcv_skb_out_fn = NULL;

    skb_reset_mac_header(skb);

    //by cys  通用网卡截获
    rcu_read_lock();
    dp_rcv_skb_out_fn = rcu_dereference(dp_rcv_skb_out);
    if (zcm_poll_enabled && dp_rcv_skb_out_fn && skb->dev &&
skb->dev->vt_zcm_ptr) {
        if (!skb_is_from_dp(skb))
        {
            rc = dp_rcv_skb_out_fn(skb);
            rcu_read_unlock();
            return rc;
        }
    }
    rcu_read_unlock();

    /* Disable soft irqs for various locks below. Also
     * stops preemption for RCU.
     */
    rcu_read_lock_bh();
    //根据设备的prority map获取skb优先级
    skb_update_prio(skb);
    //skb入dev的队列，如果有回调ndo_select_queue，就用回调来判断
    txq = netdev_pick_tx(dev, skb);
    q = rcu_dereference_bh(txq->qdisc);

#ifdef CONFIG_NET_CLS_ACT
    skb->tc_verd = SET_TC_AT(skb->tc_verd, AT_EGRESS);
#endif
    trace_net_dev_queue(skb);
    if (q->enqueue) {
        //如果调度器qdisc有自己的入队函数，有队列，一般是物理网□
        rc = __dev_xmit_skb(skb, q, dev, txq);
        goto out;
    }

    //下面是没有自己的队列的，多少loop和tun□
    /* The device has no queue. Common case for software devices:
       loopback, all the sorts of tunnels...
```

```
	Really, it is unlikely that netif_tx_lock protection is
necessary
	here.  (f.e. loopback and IP tunnels are clean ignoring
statistics
	counters.)
	However, it is possible, that they rely on protection
	made by us here.

	Check this and shot the lock. It is not prone from deadlocks.
	Either shot noqueue qdisc, it is even simpler 8)
	 */
	if (dev->flags & IFF_UP) {
	    int cpu = smp_processor_id(); /* ok because BHs are off */

	    if (txq->xmit_lock_owner != cpu) {
	        //不能加入源CPU？
	        if (__this_cpu_read(xmit_recursion) > RECURSION_LIMIT)
	            goto recursion_alert;

	        HARD_TX_LOCK(dev, txq, cpu);

	        if (!netif_xmit_stopped(txq)) {
	            __this_cpu_inc(xmit_recursion);
	            //发到txq队列
	            rc = dev_hard_start_xmit(skb, dev, txq);
	            __this_cpu_dec(xmit_recursion);
	            //发送完了还要检查返回值是否OK
	            if (dev_xmit_complete(rc)) {
	                HARD_TX_UNLOCK(dev, txq);
	                goto out;
	            }
	        }
	        HARD_TX_UNLOCK(dev, txq);
	    }
	}

	rc = -ENETDOWN;
	rcu_read_unlock_bh();

	kfree_skb(skb);
	return rc;
out:
	rcu_read_unlock_bh();
	return rc;
```

```
}

static inline int __dev_xmit_skb(struct sk_buff *skb, struct Qdisc
*q,
                struct net_device *dev,
                struct netdev_queue *txq)
{//这个函数结合调度器来发包
    spinlock_t *root_lock = qdisc_lock(q);
    bool contended;
    int rc;

    qdisc_pkt_len_init(skb);
    qdisc_calculate_pkt_len(skb, q);
    /*
     * Heuristic to force contended enqueues to serialize on a
     * separate lock before trying to get qdisc main lock.
     * This permits __QDISC_STATE_RUNNING owner to get the lock more
often
     * and dequeue packets faster.
     */
    contended = qdisc_is_running(q);
    if (unlikely(contended))
        spin_lock(&q->busylock);

    spin_lock(root_lock);
    if (unlikely(test_bit(__QDISC_STATE_DEACTIVATED, &q->state))) {
        //调度器为未激活状态, 丢弃
        kfree_skb(skb);
        rc = NET_XMIT_DROP;
    } else if ((q->flags & TCQ_F_CAN_BYPASS) && !qdisc_qlen(q) &&
           qdisc_run_begin(q)) {
        //调度器说可以BYPASS, 且没有数据包再调度器里排队, 且调度器之前没有
运行
        /*
         * This is a work-conserving queue; there are no old skbs
         * waiting to be sent out; and the qdisc is not running -
         * xmit the skb directly.
         */
        if (!(dev->priv_flags & IFF_XMIT_DST_RELEASE))
            skb_dst_force(skb);

        qdisc_bstats_update(q, skb);
        //通过sched direct xmit发送skb
        if (sch_direct_xmit(skb, q, dev, txq, root_lock)) {
```

```
            //如果q里还有数据需要发送，需要启动qdisc
            if (unlikely(contended)) {
                spin_unlock(&q->busylock);
                contended = false;
            }
            __qdisc_run(q);
        } else
            qdisc_run_end(q);

        rc = NET_XMIT_SUCCESS;
    } else {
        skb_dst_force(skb);
        //入队调度，比如最简单的到的pfifo_enqueue，如果超出限制，丢弃，否则将包添加到队尾。
        rc = q->enqueue(skb, q) & NET_XMIT_MASK;
        if (qdisc_run_begin(q)) {
            if (unlikely(contended)) {
                spin_unlock(&q->busylock);
                contended = false;
            }
            __qdisc_run(q);
        }
    }
    spin_unlock(root_lock);
    if (unlikely(contended))
        spin_unlock(&q->busylock);
    return rc;
}

int sch_direct_xmit(struct sk_buff *skb, struct Qdisc *q,
            struct net_device *dev, struct netdev_queue *txq,
            spinlock_t *root_lock)
{
    int ret = NETDEV_TX_BUSY;

    /* And release qdisc */
    spin_unlock(root_lock);

    HARD_TX_LOCK(dev, txq, smp_processor_id());
    if (!netif_xmit_frozen_or_stopped(txq))
        ret = dev_hard_start_xmit(skb, dev, txq);//发包

    HARD_TX_UNLOCK(dev, txq);
```

```
    spin_lock(root_lock);
    //和dev_queue_xmit里面，loopback，tun口使用dev_hard_start_xmit发送
后还要检查返回值一样，这里也是
    if (dev_xmit_complete(ret)) {
        /* Driver sent out skb successfully or skb was consumed */
        ret = qdisc_qlen(q);
    } else if (ret == NETDEV_TX_LOCKED) {
        /* Driver try lock failed */
        ret = handle_dev_cpu_collision(skb, txq, q);
    } else {
        /* Driver returned NETDEV_TX_BUSY - requeue skb */
        if (unlikely(ret != NETDEV_TX_BUSY))
            net_warn_ratelimited("BUG %s code %d qlen %d\n",
                         dev->name, ret, q->q.qlen);
        //如果返回的是BUSY，要重新入队，loopback口可没这待遇
        ret = dev_requeue_skb(skb, q);
    }

    if (ret && netif_xmit_frozen_or_stopped(txq))
        ret = 0;

    return ret;
}

int dev_hard_start_xmit(struct sk_buff *skb, struct net_device *dev,
            struct netdev_queue *txq)
{
    const struct net_device_ops *ops = dev->netdev_ops;
    int rc = NETDEV_TX_OK;
    unsigned int skb_len;

    //有next的表示已经分过包了？不需要再GSO？？？貌似如此
    if (likely(!skb->next)) {
        //如果没有next，大概不需要Generic Segmentation Offload，有next，
一定要gso
        netdev_features_t features;

        /*
         * If device doesn't need skb->dst, release it right now
while
         * its hot in this cpu cache
         */
        if (dev->priv_flags & IFF_XMIT_DST_RELEASE)
            skb_dst_drop(skb);
```

```
	features = netif_skb_features(skb);

	if (vlan_tx_tag_present(skb) &&
	    !vlan_hw_offload_capable(features, skb->vlan_proto)) {
	    //有vlan tag的时候，需要在SKB数据区域添加上data，而且这里使用了memove来移动数据，浪费CPU，为嘛不提前预留
	    skb = __vlan_put_tag(skb, skb->vlan_proto,
	                         vlan_tx_tag_get(skb));
	    if (unlikely(!skb))
	        goto out;

	    skb->vlan_tci = 0;
	}

	/* If encapsulation offload request, verify we are testing
	 * hardware encapsulation features instead of standard
	 * features for the netdev
	 */
	if (skb->encapsulation)
	    features &= dev->hw_enc_features;

	if (netif_needs_gso(skb, features)) {
	    //单skb也需要gso的情况，先调用dev_gso_segment分包
	    if (unlikely(dev_gso_segment(skb, features)))
	        goto out_kfree_skb;
	    if (skb->next)
	        goto gso;
	} else {
	    if (skb_needs_linearize(skb, features) &&
	        __skb_linearize(skb))
	        goto out_kfree_skb;

	    /* If packet is not checksummed and device does not
	     * support checksumming for this protocol, complete
	     * checksumming here.
	     */
	     //设置checksum
	    if (skb->ip_summed == CHECKSUM_PARTIAL) {
	        if (skb->encapsulation)
	            skb_set_inner_transport_header(skb,
	                skb_checksum_start_offset(skb));
	        else
	            skb_set_transport_header(skb,
```

```
                    skb_checksum_start_offset(skb));
            if (!(features & NETIF_F_ALL_CSUM) &&
                 skb_checksum_help(skb))
                goto out_kfree_skb;
        }
    }

    //为taps copy一份出口数据, 例如抓包工具监听PF_PACKET会获取到
    //dev_queue_xmit_nit为out回调, netif_receive_skb为in回调
    if (!list_empty(&ptype_all))
        dev_queue_xmit_nit(skb, dev);

    skb_len = skb->len;
    //OK, 终于到了调用驱动xmit的地方
    rc = ops->ndo_start_xmit(skb, dev);
    trace_net_dev_xmit(skb, rc, dev, skb_len);
    if (rc == NETDEV_TX_OK)
        txq_trans_update(txq);
    return rc;
}

gso:
    do {
        struct sk_buff *nskb = skb->next;

        skb->next = nskb->next;
        nskb->next = NULL;

        //同上, copy一份给taps
        if (!list_empty(&ptype_all))
            dev_queue_xmit_nit(nskb, dev);

        skb_len = nskb->len;
        //一个一个的发
        rc = ops->ndo_start_xmit(nskb, dev);
        trace_net_dev_xmit(nskb, rc, dev, skb_len);
        if (unlikely(rc != NETDEV_TX_OK)) {
            if (rc & ~NETDEV_TX_MASK)
                goto out_kfree_gso_skb;
            nskb->next = skb->next;
            skb->next = nskb;
            return rc;
        }
        txq_trans_update(txq);
```

```
        if (unlikely(netif_xmit_stopped(txq) && skb->next))
            return NETDEV_TX_BUSY;
    } while (skb->next);//轮

out_kfree_gso_skb:
    if (likely(skb->next == NULL)) {
        skb->destructor = DEV_GSO_CB(skb)->destructor;
        consume_skb(skb);
        return rc;
    }
out_kfree_skb:
    kfree_skb(skb);
out:
    return rc;
}
```

而 E1000E 的 xmit 回调函数为：

```
static netdev_tx_t e1000_xmit_frame(struct sk_buff *skb,
                    struct net_device *netdev)
{
    struct e1000_adapter *adapter = netdev_priv(netdev);
    struct e1000_ring *tx_ring = adapter->tx_ring;
    unsigned int first;
    unsigned int tx_flags = 0;
    unsigned int len = skb_headlen(skb);
    unsigned int nr_frags;
    unsigned int mss;
    int count = 0;
    int tso;
    unsigned int f;

    //网口down的情况，不发送数据，并free skb
    if (test_bit(__E1000_DOWN, &adapter->state)) {
        dev_kfree_skb_any(skb);
        return NETDEV_TX_OK;
    }
    //长度不合法
    if (skb->len <= 0) {
        dev_kfree_skb_any(skb);
        return NETDEV_TX_OK;
    }

    /* The minimum packet size with TCTL.PSP set is 17 bytes so
```

```
 * pad skb in order to meet this minimum size requirement
 */
if (unlikely(skb->len < 17)) {
    if (skb_pad(skb, 17 - skb->len))
        return NETDEV_TX_OK;
    skb->len = 17;
    skb_set_tail_pointer(skb, 17);
}

mss = skb_shinfo(skb)->gso_size;
if (mss) {
    u8 hdr_len;

    /* TSO Workaround for 82571/2/3 Controllers -- if skb->data
     * points to just header, pull a few bytes of payload from
     * frags into skb->data
     */
    hdr_len = skb_transport_offset(skb) + tcp_hdrlen(skb);
    /* we do this workaround for ES2LAN, but it is un-necessary,
     * avoiding it could save a lot of cycles
     */
    if (skb->data_len && (hdr_len == len)) {
        unsigned int pull_size;

        pull_size = min_t(unsigned int, 4, skb->data_len);
        if (!__pskb_pull_tail(skb, pull_size)) {
            e_err("__pskb_pull_tail failed.\n");
            dev_kfree_skb_any(skb);
            return NETDEV_TX_OK;
        }
        len = skb_headlen(skb);
    }
}

/* reserve a descriptor for the offload context */
if ((mss) || (skb->ip_summed == CHECKSUM_PARTIAL))
    count++;
count++;

count += DIV_ROUND_UP(len, adapter->tx_fifo_limit);

nr_frags = skb_shinfo(skb)->nr_frags;
for (f = 0; f < nr_frags; f++)
    count +=
```

```
DIV_ROUND_UP(skb_frag_size(&skb_shinfo(skb)->frags[f]),
                      adapter->tx_fifo_limit);

    if (adapter->hw.mac.tx_pkt_filtering)
        e1000_transfer_dhcp_info(adapter, skb);

    /* need: count + 2 desc gap to keep tail from touching
     * head, otherwise try next time
     */
    if (e1000_maybe_stop_tx(tx_ring, count + 2))
        return NETDEV_TX_BUSY;

    if (vlan_tx_tag_present(skb)) {
        tx_flags |= E1000_TX_FLAGS_VLAN;
        tx_flags |= (vlan_tx_tag_get(skb) <<
E1000_TX_FLAGS_VLAN_SHIFT);
    }

    first = tx_ring->next_to_use;

    tso = e1000_tso(tx_ring, skb);
    if (tso < 0) {
        dev_kfree_skb_any(skb);
        return NETDEV_TX_OK;
    }
    //如果tso成功，要打上E1000_TX_FLAGS_TSO标记
    if (tso)
        tx_flags |= E1000_TX_FLAGS_TSO;
    else if (e1000_tx_csum(tx_ring, skb))
        tx_flags |= E1000_TX_FLAGS_CSUM;

    /* Old method was to assume IPv4 packet by default if TSO was
enabled.
     * 82571 hardware supports TSO capabilities for IPv6 as well...
     * no longer assume, we must.
     */
    if (skb->protocol == htons(ETH_P_IP))
        tx_flags |= E1000_TX_FLAGS_IPV4;

    if (unlikely(skb->no_fcs))
        tx_flags |= E1000_TX_FLAGS_NO_FCS;

    /* if count is 0 then mapping error has occurred */
    count = e1000_tx_map(tx_ring, skb, first, adapter->tx_fifo_limit,
```

```
                nr_frags);
    if (count) {
        if (unlikely((skb_shinfo(skb)->tx_flags & SKBTX_HW_TSTAMP) &&
                 !adapter->tx_hwtstamp_skb)) {
            skb_shinfo(skb)->tx_flags |= SKBTX_IN_PROGRESS;
            tx_flags |= E1000_TX_FLAGS_HWTSTAMP;
            adapter->tx_hwtstamp_skb = skb_get(skb);
            schedule_work(&adapter->tx_hwtstamp_work);
        } else {
            skb_tx_timestamp(skb);
        }

        netdev_sent_queue(netdev, skb->len);
        e1000_tx_queue(tx_ring, tx_flags, count);
        /* Make sure there is space in the ring for the next send. */
        e1000_maybe_stop_tx(tx_ring,
                    (MAX_SKB_FRAGS *
                     DIV_ROUND_UP(PAGE_SIZE,
                          adapter->tx_fifo_limit) + 2));
    } else {
        dev_kfree_skb_any(skb);
        tx_ring->buffer_info[first].time_stamp = 0;
        tx_ring->next_to_use = first;
    }

    return NETDEV_TX_OK;
}

static int e1000_tx_map(struct e1000_ring *tx_ring, struct sk_buff
*skb,
            unsigned int first, unsigned int max_per_txd,
            unsigned int nr_frags)
{
    struct e1000_adapter *adapter = tx_ring->adapter;
    struct pci_dev *pdev = adapter->pdev;
    struct e1000_buffer *buffer_info;
    unsigned int len = skb_headlen(skb);
    unsigned int offset = 0, size, count = 0, i;
    unsigned int f, bytecount, segs;

    i = tx_ring->next_to_use;
    //先处理none paged data
    while (len) {
        buffer_info = &tx_ring->buffer_info[i];
```

```
    size = min(len, max_per_txd);

    buffer_info->length = size;//这个length记录的是被dma map的长度
    buffer_info->time_stamp = jiffies;
    buffer_info->next_to_watch = i;
    buffer_info->dma = dma_map_single(&pdev->dev,
                      skb->data + offset,
                      size, DMA_TO_DEVICE);
    buffer_info->mapped_as_page = false;
    if (dma_mapping_error(&pdev->dev, buffer_info->dma))
        goto dma_error;

    len -= size;
    offset += size;
    count++;

    if (len) {
        i++;
        if (i == tx_ring->count)
            i = 0;
    }
}

//再处理paged data
for (f = 0; f < nr_frags; f++) {
    const struct skb_frag_struct *frag;

    frag = &skb_shinfo(skb)->frags[f];
    len = skb_frag_size(frag);
    offset = 0;

    while (len) {
        i++;
        if (i == tx_ring->count)
            i = 0;

        buffer_info = &tx_ring->buffer_info[i];
        size = min(len, max_per_txd);

        buffer_info->length = size;
        buffer_info->time_stamp = jiffies;
        buffer_info->next_to_watch = i;
        buffer_info->dma = skb_frag_dma_map(&pdev->dev, frag,
                        offset, size,
```

```
                                DMA_TO_DEVICE);
            buffer_info->mapped_as_page = true;
            if (dma_mapping_error(&pdev->dev, buffer_info->dma))
                goto dma_error;

            len -= size;
            offset += size;
            count++;
        }
    }

    segs = skb_shinfo(skb)->gso_segs ? : 1;
    /* multiply data chunks by size of headers */
    bytecount = ((segs - 1) * skb_headlen(skb)) + skb->len;

    tx_ring->buffer_info[i].skb = skb;
    tx_ring->buffer_info[i].segs = segs;            //这个segs记录了此
skb有多少个分片
    tx_ring->buffer_info[i].bytecount = bytecount;  //segs和skb一样,
只记录到最后一个slot，方便clean tx的时候统计发送了多少包和字节
    tx_ring->buffer_info[first].next_to_watch = i;//是一个范围，保留一
下这个skb从那个slot到哪个slot，那么first就是from，i就是to，到时候clean
tx的时候需要

    return count;

dma_error:
    dev_err(&pdev->dev, "Tx DMA map failed\n");
    buffer_info->dma = 0;
    if (count)
        count--;

    while (count--) {
        if (i == 0)
            i += tx_ring->count;
        i--;
        buffer_info = &tx_ring->buffer_info[i];
        e1000_put_txbuf(tx_ring, buffer_info);
    }

    return 0;
}

static void e1000_tx_queue(struct e1000_ring *tx_ring, int tx_flags,
```

```
int count)
{
    struct e1000_adapter *adapter = tx_ring->adapter;
    struct e1000_tx_desc *tx_desc = NULL;
    struct e1000_buffer *buffer_info;
    u32 txd_upper = 0, txd_lower = E1000_TXD_CMD_IFCS;
    unsigned int i;

    if (tx_flags & E1000_TX_FLAGS_TSO) {
        txd_lower |= E1000_TXD_CMD_DEXT | E1000_TXD_DTYP_D |
            E1000_TXD_CMD_TSE;
        txd_upper |= E1000_TXD_POPTS_TXSM << 8;

        if (tx_flags & E1000_TX_FLAGS_IPV4)
            txd_upper |= E1000_TXD_POPTS_IXSM << 8;
    }

    if (tx_flags & E1000_TX_FLAGS_CSUM) {
        txd_lower |= E1000_TXD_CMD_DEXT | E1000_TXD_DTYP_D;
        txd_upper |= E1000_TXD_POPTS_TXSM << 8;
    }

    if (tx_flags & E1000_TX_FLAGS_VLAN) {
        txd_lower |= E1000_TXD_CMD_VLE;
        txd_upper |= (tx_flags & E1000_TX_FLAGS_VLAN_MASK);
    }

    if (unlikely(tx_flags & E1000_TX_FLAGS_NO_FCS))
        txd_lower &= ~(E1000_TXD_CMD_IFCS);

    if (unlikely(tx_flags & E1000_TX_FLAGS_HWTSTAMP)) {
        txd_lower |= E1000_TXD_CMD_DEXT | E1000_TXD_DTYP_D;
        txd_upper |= E1000_TXD_EXTCMD_TSTAMP;
    }

    i = tx_ring->next_to_use;

    do {
        buffer_info = &tx_ring->buffer_info[i];
        tx_desc = E1000_TX_DESC(*tx_ring, i);
        tx_desc->buffer_addr = cpu_to_le64(buffer_info->dma);
        tx_desc->lower.data = cpu_to_le32(txd_lower |
                        buffer_info->length);
        tx_desc->upper.data = cpu_to_le32(txd_upper);
```

```
        i++;
        if (i == tx_ring->count)
            i = 0;
    } while (--count > 0);

    tx_desc->lower.data |= cpu_to_le32(adapter->txd_cmd);

    /* txd_cmd re-enables FCS, so we'll re-disable it here as
desired. */
    if (unlikely(tx_flags & E1000_TX_FLAGS_NO_FCS))
        tx_desc->lower.data &= ~(cpu_to_le32(E1000_TXD_CMD_IFCS));

    /* Force memory writes to complete before letting h/w
     * know there are new descriptors to fetch.  (Only
     * applicable for weak-ordered memory model archs,
     * such as IA-64).
     */
    wmb();

    tx_ring->next_to_use = i;

    if (adapter->flags2 & FLAG2_PCIM2PCI_ARBITER_WA)
        e1000e_update_tdt_wa(tx_ring, i);
    else
        writel(i, tx_ring->tail);

    /* we need this if more than one processor can write to our tail
     * at a time, it synchronizes IO on IA64/Altix systems
     */
    mmiowb();
}
```

完成发包

上面只是将包放到网卡的发包 slot 里了，但是发送成功与否，你并不知道，驱动一般有两种方式来获取是否成功，一是通过中断通知；二是在 recv poll 的时候顺便检测。

中断注册过程为：

```
int e1000_open(struct net_device *netdev)
    └── int e1000_request_irq(struct e1000_adapter *adapter) 注册中断
            if  ├── int e1000_request_msix(struct e1000_adapter *adapter)
                │       优先尝试msix, int_mode=E1000E_INT_MODE_MSIX=2
                │       ├── vector 0注册为e1000_intr_msix_rx, for rx
                │       ├── vector 1注册为e1000_intr_msix_tx, for tx
                │       └── vector 2注册为e1000_msix_other
            elif├── int e1000_request_msix(struct e1000_adapter *adapter)
                │       然后尝试msi, int_mode=E1000E_INT_MODE_MSI=1
            else└── irqreturn_t e1000_intr(int __always_unused irq, void *data)
                        都不行, 只能用传统模式, int_mode=E1000E_INT_MODE_LEGACY=0
```

recv poll 为 e1000e_poll，其调用 e1000_clean_tx_irq 来检查是否发包成功：

```
    if (!adapter->msix_entries ||
        (adapter->rx_ring->ims_val & adapter->tx_ring->ims_val))
        tx_cleaned  =  e1000_clean_tx_irq(adapter->tx_ring);//检查发包是否完成

static bool e1000_clean_tx_irq(struct e1000_ring *tx_ring)
{
    struct e1000_adapter *adapter = tx_ring->adapter;
    struct net_device *netdev = adapter->netdev;
    struct e1000_hw *hw = &adapter->hw;
    struct e1000_tx_desc *tx_desc, *eop_desc;
    struct e1000_buffer *buffer_info;
    unsigned int i, eop;
    unsigned int count = 0;
    unsigned int total_tx_bytes = 0, total_tx_packets = 0;
    unsigned int bytes_compl = 0, pkts_compl = 0;

#ifdef DEV_NETMAP
    //发包HOOK
    if (netmap_tx_irq(netdev, 0))
        return 1; /* cleaned ok */
#endif /* DEV_NETMAP */

    //next_to_clean, 是需要tx_clean的起始位置, 结束位置为next_to_use
    //next_to_watch是一个skb装到n个slot的结束slot位置, 这个位置里存放了skb, segs, bytecount等用于统计的信息
    i = tx_ring->next_to_clean;
    eop = tx_ring->buffer_info[i].next_to_watch;
    eop_desc = E1000_TX_DESC(*tx_ring, eop);

    while ((eop_desc->upper.data & cpu_to_le32(E1000_TXD_STAT_DD)) &&
           (count < tx_ring->count)) {
        bool cleaned = false;
        rmb();     /* read buffer_info after eop_desc */
```

```c
        for (; !cleaned; count++) {
            tx_desc = E1000_TX_DESC(*tx_ring, i);
            buffer_info = &tx_ring->buffer_info[i];
            cleaned = (i == eop);

            if (cleaned) {
                total_tx_packets += buffer_info->segs;
                total_tx_bytes += buffer_info->bytecount;
                if (buffer_info->skb) {
                    bytes_compl += buffer_info->skb->len;
                    pkts_compl++;
                }
            }

            e1000_put_txbuf(tx_ring, buffer_info);
            tx_desc->upper.data = 0;

            i++;
            if (i == tx_ring->count)
                i = 0;
        }

        if (i == tx_ring->next_to_use)//追上了next_to_use, OK, 结束
            break;
        eop = tx_ring->buffer_info[i].next_to_watch;
        eop_desc = E1000_TX_DESC(*tx_ring, eop);
    }

    tx_ring->next_to_clean = i;

    netdev_completed_queue(netdev, pkts_compl, bytes_compl);

#define TX_WAKE_THRESHOLD 32
    if (count && netif_carrier_ok(netdev) &&
        e1000_desc_unused(tx_ring) >= TX_WAKE_THRESHOLD) {
        /* Make sure that anybody stopping the queue after this
         * sees the new next_to_clean.
         */
        smp_mb();

        if (netif_queue_stopped(netdev) &&
            !(test_bit(__E1000_DOWN, &adapter->state))) {
            netif_wake_queue(netdev);
            ++adapter->restart_queue;
```

```
        }
    }

    if (adapter->detect_tx_hung) {
        /* Detect a transmit hang in hardware, this serializes the
         * check with the clearing of time_stamp and movement of i
         */
        adapter->detect_tx_hung = false;
        if (tx_ring->buffer_info[i].time_stamp &&
            time_after(jiffies, tx_ring->buffer_info[i].time_stamp
                   + (adapter->tx_timeout_factor * HZ)) &&
            !(er32(STATUS) & E1000_STATUS_TXOFF))
            schedule_work(&adapter->print_hang_task);
        else
            adapter->tx_hang_recheck = false;
    }
    adapter->total_tx_bytes += total_tx_bytes;
    adapter->total_tx_packets += total_tx_packets;
    return count < tx_ring->count;
}
```

tx_ring 索引

这里涉及到 tx_ring 的索引问题，我将其画为图阐述如下（head 忽略）：

head, tail　　为闭区间索引，假设 slot 个数为 16。
next_to_use　　是内存分配完毕，网卡可以将数据写入其描述符的最后下标。
next_to_clean　是网卡已经讲数据写入后，驱动从描述符中将数据取出的开始下标。

所以，next_to_clean 指示的位置，一定要 next_to_use 先初始化过。

1　代表驱动已经写入数据到网卡的 slot
空　代表啥都没做
0　代表只少填充过一次数据

1. e1000e_setup_tx_resources - 初始化阶段

```
head=tail=0
|
v
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
```

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
^
|
clean=use=0
```

2. e1000_xmit_frame - 发包
   考虑最简单的情况，不需分包，不需要 checksum
   于是进入 e1000_tx_map 和 e1000_tx_queue：

```
head=0
|  tail=1
|  |
v  v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
^  ^
|  |
|  use=1
clean=0
```

   每写一个包，更新 use 和 tail，两者相同
   假设一下子写了 11 个包：

```
head=0                           tail=11
|                                |
v                                v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |  |  |  |  |  |
|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
^                                ^
|                                |
clean=0                          use=11
```

3. e1000_clean_tx_irq 中断或 poll 来检查是否发包成功
   假设发包成功 4 个包，clean 指向。

```
head=0                           tail=11
|                                |
v                                v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|0 |0 |0 |0 |1 |1 |1 |1 |1 |1 |1 |  |  |  |  |  |
```

```
|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
            ^                    ^
            |                    |
            clean=4              use=11
```

此时，unused = ring->count + ring->next_to_clean - ring->next_to_use - 1 = 16 + 4 - 11 - 1 = 8，事实上位 9 个空位，但是为了保证不会冲掉 clean，都留了一个间隙。

假设都发送成功，再次调用 e1000_clean_tx_irq，检查 64 个，那么最多检查到 use 前面：

```
head=0                           tail=11
|                                |
v                                v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |  |  |  |  |  |
|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                 ^
                                 |
                                 clean=use=11
```

unused = 15，假设此次发送 15 个包：

```
head=0                        tail=10
|                             |
v                             v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |0 |1 |1 |1 |1 |1 |
|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                              ^  ^
                              |  |
                              |  clean=11
                              use=10
```

参考资料

1. 《SKB 解析》http://vger.kernel.org/~davem/skb_data.html
2. 《PCI 设备驱动》 http://blog.csdn.net/lamdoc/article/details/7698709

3. 《Linux 内核中 ioremap 映射的透彻理解》
http://blog.csdn.net/do2jiang/article/details/5450839

2014/11/18