http://www.pagefault.info/?p=403

intel 万兆网卡驱动简要分析

这里分析的驱动代码是给予 linux kernel 3.4.4

对应的文件在 drivers/net/ethernet/intel 目录下，这个分析不涉及到很细节的地方，主要目的是理解下数据在协议栈和驱动之间是如何交互的。

首先我们知道网卡都是 pci 设备，因此这里每个网卡驱动其实就是一个 pci 驱动。并且 intel 这里是把好几个万兆网卡(82599/82598/x540)的驱动做在一起的。

首先我们来看对应的 pci_driver 的结构体，这里每个 pci 驱动都是一个 pci_driver 的结构体，而这里是多个万兆网卡共用这个结构体 ixgbe_driver.

```
1  staticstructpci_driver ixgbe_driver = {
2      .name     = ixgbe_driver_name,
3      .id_table = ixgbe_pci_tbl,
4      .probe    = ixgbe_probe,
5      .remove   = __devexit_p(ixgbe_remove),
6  #ifdef CONFIG_PM
7      .suspend  = ixgbe_suspend,
8      .resume   = ixgbe_resume,
9  #endif
10     .shutdown = ixgbe_shutdown,
11     .err_handler = &ixgbe_err_handler
12 };
```

然后是模块初始化方法,这里其实很简单，就是调用 pci 的驱动注册方法，把 ixgbe 挂载到 pci 设备链中。这里不对 pci 设备的初始化做太多介绍，我以前的 blog 有这方面的介绍，想了解的可以去看看。这里我们只需要知道最终内核会调用 probe 回调来初始化 ixgbe。

```
1  charixgbe_driver_name[] ="ixgbe";
2  staticconstcharixgbe_driver_string[] =
3                  "Intel(R) 10 Gigabit PCI Express Network Driver";
4
5  staticint__init ixgbe_init_module(void)
6  {
7      intret;
8      pr_info("%s - version %s\n", ixgbe_driver_string, ixgbe_driver_version);
9      pr_info("%s\n", ixgbe_copyright);
10
```

```
#ifdef CONFIG_IXGBE_DCA
    dca_register_notify(&dca_notifier);
#endif

    ret = pci_register_driver(&ixgbe_driver);
    returnret;
}
```

这里不去追究具体如何调用 probe 的细节，我们直接来看 probe 函数，这个函数中通过硬件的信息来确定需要初始化那个驱动(82598/82599/x540),然后核心的驱动结构就放在下面的这个数组中。

```
staticconststructixgbe_info *ixgbe_info_tbl[] = {
    [board_82598] = &ixgbe_82598_info,
    [board_82599] = &ixgbe_82599_info,
    [board_X540] = &ixgbe_X540_info,
};
```

ixgbe_probe 函数很长，我们这里就不详细分析了，因为这部分就是对网卡进行初始化。不过我们关注下面几个代码片段。

首先是根据硬件的参数来取得对应的驱动值:

```
conststructixgbe_info *ii = ixgbe_info_tbl[ent->driver_data];
```

然后就是如何将不同的网卡驱动挂载到对应的回调中，这里做的很简单，就是通过对应的 netdev 的结构取得 adapter，然后所有的核心操作都是保存在 adapter 中的，最后将 ii 的所有回调拷贝给 adapter 就可以了。我们来看代码：

```
    structnet_device *netdev;
    structixgbe_adapter *adapter = NULL;
    structixgbe_hw *hw;
.....................................

    adapter = netdev_priv(netdev);
    pci_set_drvdata(pdev, adapter);

    adapter->netdev = netdev;
    adapter->pdev = pdev;
    hw = &adapter->hw;
    hw->back = adapter;
.....................................
    memcpy(&hw->mac.ops, ii->mac_ops,sizeof(hw->mac.ops));
    hw->mac.type  = ii->mac;

    /* EEPROM */
```

```
    memcpy(&hw->eeprom.ops, ii->eeprom_ops,sizeof(hw->eeprom.ops));
.....................................
```

最后需要关注的就是设置网卡属性，这些属性一般来说都是通过 ethtool 可以设置的属性(比如 tso/checksum 等),这里我们就截取一部分:

```
    netdev->features = NETIF_F_SG |
               NETIF_F_IP_CSUM |
               NETIF_F_IPV6_CSUM |
               NETIF_F_HW_VLAN_TX |
               NETIF_F_HW_VLAN_RX |
               NETIF_F_HW_VLAN_FILTER |
               NETIF_F_TSO |
               NETIF_F_TSO6 |
               NETIF_F_RXHASH |
               NETIF_F_RXCSUM;

    netdev->hw_features = netdev->features;

    switch(adapter->hw.mac.type) {
    caseixgbe_mac_82599EB:
    caseixgbe_mac_X540:
        netdev->features |= NETIF_F_SCTP_CSUM;
        netdev->hw_features |= NETIF_F_SCTP_CSUM |
                     NETIF_F_NTUPLE;
        break;
    default:
        break;
    }

    netdev->hw_features |= NETIF_F_RXALL;
.............................................

    netdev->priv_flags |= IFF_UNICAST_FLT;
    netdev->priv_flags |= IFF_SUPP_NOFCS;

    if(adapter->flags & IXGBE_FLAG_SRIOV_ENABLED)
        adapter->flags &= ~(IXGBE_FLAG_RSS_ENABLED |
                   IXGBE_FLAG_DCB_ENABLED);
.................................................................
    if(pci_using_dac) {
        netdev->features |= NETIF_F_HIGHDMA;
        netdev->vlan_features |= NETIF_F_HIGHDMA;
    }
```

```
39
40      if(adapter->flags2 & IXGBE_FLAG2_RSC_CAPABLE)
41          netdev->hw_features |= NETIF_F_LRO;
42      if(adapter->flags2 & IXGBE_FLAG2_RSC_ENABLED)
43          netdev->features |= NETIF_F_LRO;
```

然后我们来看下中断的注册，因为万兆网卡大部分都是多对列网卡(配合 msix)，因此对于上层软件来说，就好像有多个网卡一样，它们之间的数据是相互独立的，这里读的话主要是 napi 驱动的 poll 方法，后面我们会分析这个.

到了这里或许要问那么网卡是如何挂载回调给上层，从而上层来发送数据呢，这里是这样子的，每个网络设备都有一个回调函数表(比如 ndo_start_xmit)来供上层调用，而在 ixgbe 中的话，就是 ixgbe_netdev_ops，下面就是这个结构，不过只是截取了我们很感兴趣的几个地方.

不过这里注意，读回调并不在里面，这是因为写是软件主动的，而读则是硬件主动的。现在 ixgbe 是 NAPI 的，因此它的 poll 回调是 ixgbe_poll，是中断注册时候通过 netif_napi_add 添加进去的。

```
1   staticconststructnet_device_ops ixgbe_netdev_ops = {
2       .ndo_open       = ixgbe_open,
3       .ndo_stop       = ixgbe_close,
4       .ndo_start_xmit     = ixgbe_xmit_frame,
5       .ndo_select_queue   = ixgbe_select_queue,
6       .ndo_set_rx_mode    = ixgbe_set_rx_mode,
7       .ndo_validate_addr  = eth_validate_addr,
8       .ndo_set_mac_address    = ixgbe_set_mac,
9       .ndo_change_mtu     = ixgbe_change_mtu,
10      .ndo_tx_timeout     = ixgbe_tx_timeout,
11  ..............................................
12      .ndo_set_features = ixgbe_set_features,
13      .ndo_fix_features = ixgbe_fix_features,
14  };
```

这里我们最关注的其实就是 ndo_start_xmit 回调，这个回调就是驱动提供给协议栈的发送回调接口。我们来看这个函数.

它的实现很简单，就是选取对应的队列，然后调用 ixgbe_xmit_frame_ring 来发送数据。

```
1   staticnetdev_tx_t ixgbe_xmit_frame(structsk_buff *skb,
2                       structnet_device *netdev)
3   {
4       structixgbe_adapter *adapter = netdev_priv(netdev);
5       structixgbe_ring *tx_ring;
6
7       if(skb->len <= 0) {
8           dev_kfree_skb_any(skb);
9           returnNETDEV_TX_OK;
```

```
    }

    /*
     * The minimum packet size for olinfo paylen is 17 so pad the skb
     * in order to meet this minimum size requirement.
     */
    if(skb->len < 17) {
        if(skb_padto(skb, 17))
            returnNETDEV_TX_OK;
        skb->len = 17;
    }
//取得对应的队列
    tx_ring = adapter->tx_ring[skb->queue_mapping];
//发送数据
    returnixgbe_xmit_frame_ring(skb, adapter, tx_ring);
}
```

而在 ixgbe_xmit_frame_ring 中，我们就关注两个地方，一个是 tso(什么是 TSO，请自行 google)，一个是如何发送.

```
    tso = ixgbe_tso(tx_ring, first, &hdr_len);
    if(tso < 0)
        gotoout_drop;
    elseif(!tso)
        ixgbe_tx_csum(tx_ring, first);

    /* add the ATR filter if ATR is on */
    if(test_bit(__IXGBE_TX_FDIR_INIT_DONE, &tx_ring->state))
        ixgbe_atr(tx_ring, first);

#ifdef IXGBE_FCOE
xmit_fcoe:
#endif /* IXGBE_FCOE */
    ixgbe_tx_map(tx_ring, first, hdr_len);
```

调用 ixgbe_tso 处理完 tso 之后，就会调用 ixgbe_tx_map 来发送数据。而 ixgbe_tx_map 所做的最主要是两步，第一步请求 DMA，第二步写寄存器，通知网卡发送数据.

```
    dma = dma_map_single(tx_ring->dev, skb->data, size, DMA_TO_DEVICE);
    if(dma_mapping_error(tx_ring->dev, dma))
        gotodma_error;

    /* record length, and DMA address */
    dma_unmap_len_set(first, len, size);
```

```
    dma_unmap_addr_set(first, dma, dma);

    tx_desc->read.buffer_addr = cpu_to_le64(dma);

    for(;;) {
        while(unlikely(size > IXGBE_MAX_DATA_PER_TXD)) {
            tx_desc->read.cmd_type_len =
                cmd_type | cpu_to_le32(IXGBE_MAX_DATA_PER_TXD);

            i++;
            tx_desc++;
            if(i == tx_ring->count) {
                tx_desc = IXGBE_TX_DESC(tx_ring, 0);
                i = 0;
            }

            dma += IXGBE_MAX_DATA_PER_TXD;
            size -= IXGBE_MAX_DATA_PER_TXD;

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

.................................................
        data_len -= size;

        dma = skb_frag_dma_map(tx_ring->dev, frag, 0, size,
                               DMA_TO_DEVICE);
.....................................................

        frag++;
    }
.............................
    tx_ring->next_to_use = i;

    /* notify HW of packet */
    writel(i, tx_ring->tail);
..............
```

上面的操作是异步的，也就是说此时内核还不能释放 SKB，而是网卡硬件发送完数据之后，会再次产生中断通知内核，然后内核才能释放内存.接下来我们来看这部分代码。

首先来看的是中断注册的代码，这里我们假设启用了 MSIX,那么网卡的中断注册回调就是 ixgbe_request_msix_irqs 函数，这里我们可以看到调用 request_irq 函数来注册回调，并且每个队列

都有自己的中断号。

```
staticintixgbe_request_msix_irqs(structixgbe_adapter *adapter)
{
    structnet_device *netdev = adapter->netdev;
    intq_vectors = adapter->num_msix_vectors - NON_Q_VECTORS;
    intvector, err;
    intri = 0, ti = 0;

    for(vector = 0; vector < q_vectors; vector++) {
        structixgbe_q_vector *q_vector = adapter->q_vector[vector];
        structmsix_entry *entry = &adapter->msix_entries[vector];
.....................................................................
        err = request_irq(entry->vector, &ixgbe_msix_clean_rings, 0,
                q_vector->name, q_vector);
        if(err) {
            e_err(probe,"request_irq failed for MSIX interrupt "
                "Error: %d\n", err);
            gotofree_queue_irqs;
        }
        /* If Flow Director is enabled, set interrupt affinity */
        if(adapter->flags & IXGBE_FLAG_FDIR_HASH_CAPABLE) {
            /* assign the mask for this irq */
            irq_set_affinity_hint(entry->vector,
                        &q_vector->affinity_mask);
        }
    }

.........................................

    return0;

free_queue_irqs:
    ............................
    returnerr;
}
```

而对应的中断回调是 ixgbe_msix_clean_rings,而这个函数呢，做的事情很简单(需要熟悉 NAPI 的原理，我以前的 blog 有介绍),就是调用 napi_schedule 来重新加入软中断处理.

```
staticirqreturn_t ixgbe_msix_clean_rings(intirq,void*data)
{
    structixgbe_q_vector *q_vector = data;

    /* EIAM disabled interrupts (on this vector) for us */
```

```
6
7       if(q_vector->rx.ring || q_vector->tx.ring)
8           napi_schedule(&q_vector->napi);
9
10      returnIRQ_HANDLED;
11  }
```

而 NAPI 驱动我们知道，最终是会调用网卡驱动挂载的 poll 回调，在 ixgbe 中，对应的回调就是 ixgbe_poll，那么也就是说这个函数要做两个工作，一个是处理读，一个是处理写完之后的清理.

```
1   intixgbe_poll(structnapi_struct *napi,intbudget)
2   {
3       structixgbe_q_vector *q_vector =
4                   container_of(napi,structixgbe_q_vector, napi);
5       structixgbe_adapter *adapter = q_vector->adapter;
6       structixgbe_ring *ring;
7       intper_ring_budget;
8       boolclean_complete =true;
9
10  #ifdef CONFIG_IXGBE_DCA
11      if(adapter->flags & IXGBE_FLAG_DCA_ENABLED)
12          ixgbe_update_dca(q_vector);
13  #endif
14  //清理写
15      ixgbe_for_each_ring(ring, q_vector->tx)
16          clean_complete &= !!ixgbe_clean_tx_irq(q_vector, ring);
17
18      /* attempt to distribute budget to each queue fairly, but don't allow
19       * the budget to go below 1 because we'll exit polling */
20      if(q_vector->rx.count > 1)
21          per_ring_budget = max(budget/q_vector->rx.count, 1);
22      else
23          per_ring_budget = budget;
24  //读数据，并清理已完成的
25      ixgbe_for_each_ring(ring, q_vector->rx)
26          clean_complete &= ixgbe_clean_rx_irq(q_vector, ring,
27                               per_ring_budget);
28
29      /* If all work not completed, return budget and keep polling */
30      if(!clean_complete)
31          returnbudget;
32
33      /* all work done, exit the polling mode */
34      napi_complete(napi);
```

```
35      if(adapter->rx_itr_setting & 1)
36          ixgbe_set_itr(q_vector);
37      if(!test_bit(__IXGBE_DOWN, &adapter->state))
38          ixgbe_irq_enable_queues(adapter, ((u64)1 << q_vector->v_idx));
39
40      return0;
41  }
```