【问题描述】：

带VLAN的IPoE方式，ping包有数次连续丢十来个包的现象，但链路没有掉链，

环境配置：PC通过IpoE接口ping BAS，同时运行ftp的脚本。

Modem配置：配置两个voip帐号和一个ipoe接口。

【问题解决】

1. **更新加速器版本后，ping丢包故障解决**
2. **FTP故障，是由于在mirror函数中使用skb_copy，原始包和镜像包公用同一个数据区导致。**

【解决过程】

PC通过DHCP获取地址，租期为24小时。Modem通过DHCP到BAS上获取地址，租期为4分钟。我们出现故障的频率为1个小时一次，每次持续大约1分钟。

故障表现为，在FTP下载时出现很多错包。FTP下载时出错的包被丢掉然后重传，因此故障的现象不明显。但是我们认为这个故障比较有趣，难以理解，而且ping丢包的故障也与此有关系。所以还是有研究的必要的。

7月5日

1. WAN口的镜像包出错，但是Modem的转发包却是正确的。
2. FTP出错的镜像包和通过print_packet打印的出错的镜像包表现形式一样，都是头上4个字节出错。
3. 在刚开发FTP下载时会出现出错包，之后就没有了。故障可能与软加速有关系。
4. 硬加速的计数器可以读取pktcmf_sar_sw和pktcmf_sw_sar的计数器。这两个是硬加速的接口。
5. 硬加速是否走中断了？如果走中断，那么能否将进入中断时的包给打印出来？

调试经验

解决故障的一种方法，回退使用之前的某一个某日构造版本，然后验证故障是否也存在问题。通过这种方法，可以判断是否新修改的代码引入的故障。

7月6日

1. 关闭硬加速，让所有的包都能够走软件驱动，方便判断。

2. 将出错的包写到内存而不是串口，减少串口消耗的CPU。

3. 在桥模式下面可以复现故障，可以剔除协议栈直接在驱动之间转发包，缩小故障范围。

7月7日

完成了一个memlog的工具，用于将消息打印到内存中，然后使用proc文件读取。

最初设计这个工具，想和flashlog公用相同的代码，但实现起来麻烦重重。后来发现，根本不需要那么麻烦，不需要受限于flash那种只能从1写为0的特性，整一个很怪异的数据结构。因此有了现在的memlog方案，直接申请一块内存用于存放消息。

但是有一个问题，最开始使用vmalloc申请内存时，一直报错误。改用kmalloc就好了。

这导致memlog存放的消息数量极为有限，只能存放128k的数据。

7月8日

1. 我打算将所有包的头部信息记录下来，但发现加入这种记录代码后故障就不再出现，怀疑是这段代码耗时导致，因此在WAN口驱动skb_hdrinit后面加入了melay(5)，做一个5ms的延时，故障不再出现。

2. 打开print_packet后，故障不再出现。

3. 之后，我对记录包头部信息的memlog模块进行优化，减少耗时，成功地记录下所有包的头部信息。（关闭硬加速）

（1）在skb_hdrinit和mirror后，加入记录点，没有出现错包。

（2）在mirror，blog_init内，blog_init外加入记录点，没有出错包。

（3）在WAN口的mirror和LAN口的blog_emit后面加入记录点，没有出错包。这里我统计了一下各种包的个数：wireshark记录的正确包个数为447个，错误的为59个，而在blog_emit后统计的包的个数为503个，大致相符。

在LAN口驱动发送镜像包的时候都还没有错误，但是PC去抓到很多的错误镜像包。

基于以上分析，我有一个推论：

1. FTP的错包与加速没有关系。加速器加快了发包的速率。

下一步计划：

1. 明确关闭硬加速后，ping包是否仍然丢失。这一点是后续解决故障的前提。

2. 如何在内核代码中检测到ping包丢失了。

3. 如果内核中没有检测到ping包丢失，而PC检测到ping丢失了，那就是我们代码的问题，此时可以让代码中的一些记录点开始生效，观察代码流程到何处。反之，如果内核也没有检测到ping包丢失，那么检查WAN口驱动的寄存器、中断计数器等。确认Modem是否收到包。

如何查看ping包丢失时的上下文？

使用一个memlog记录所有的收发包信息，由于收发包很多，因此当检测到丢包时，将该memlog保存下来。

如何检测ping包丢失？

Ping的回应包1秒钟收到一个，每次收到后更新一下当前收到包的jiffies记录，因此可以在内核中添加一个定时器，定时检测这个变量和前面的jiffies时间差，如果超过3s没有收到回应包，则认为回应包丢失，开启相关变量。反之，当再次收到回应包后，关闭该变量。

由于每次发生ping包丢失时，持续时间长达一分钟，因此这种方法应该是可行的。

一、抓包分析

测试部环境中，ping包的同时打电话，以及ftp下载。下面是一次出现故障后的抓包。经过分析，有以下现象：

结论：持续ping包5324秒后，**modem向BAS发送的所有包都不能得到回应**，持续时间58s。

1. 从第5324秒，ping包开始出现错误

No.	Time	Source	Destination	Protocol	Info
29842	5323.347927	202.208.3.194	90.1.1.12	TCP	4712 > ftp [ACK] Seq=75 Ack=264 Win=65272 Len=0
29843	5324.000000	202.208.3.194	202.208.3.254	ICMP	Echo (ping) request
29844	5324.167531	202.208.3.194	202.208.3.254	ICMP	Echo (ping) request
29845	5329.335930	192.168.1.2	202.208.3.254	ICMP	Echo (ping) request
29846	5329.336688	202.208.3.194	202.208.3.254	ICMP	Echo (ping) request
29847	5334.333536	Zte_74:12:37	Zte_0a:95:a0	ARP	Who has 202.208.3.254? Tell 202.208.3.194

2. 5324、5329这两个包由PC发给Modem、然后Modem转发。通过分析包的MAC地址，此时Modem能够查询ARP地址的。但是BAS方面一直没有replay 5324和5329这两个包。00:22:93:74:12:37是Modem的MAC地址

3. 正常情况，每隔21秒Modem需要向BAS发送一个arp请求，确认其是否在线。大约100ms后，Modem能够收到BAS的arp回应包。

4953.011657	Zte_74:12:37	Zte_0a:95:a0	ARP	Who has 202.208.3.254? Tell 202.208.3.194
4974.021293	Zte_74:12:37	Zte_0a:95:a0	ARP	Who has 202.208.3.254? Tell 202.208.3.194
4995.030918	Zte_74:12:37	Zte_0a:95:a0	ARP	Who has 202.208.3.254? Tell 202.208.3.194

4. 出错情况的arp包情况。

5310s正常，间隔约24秒后，再次发送arp请求，由于此时202.208.3.254的MAC地址在arp表中有记录，因此包发送给zte_74:12:37。

5334s的arp包，BAS没有回应。因此之后每隔1s重复发送arp包。发送三次，也就是5334s,5335s,3536s时发送的arp包。BAS一直没有回应。

再次间隔4s后，Modem向外发送arp包，查询202.208.3.254。注意到此时的MAC地址为广播，因此可以断定，此时202.208.3.254—zte_74:12:37的arp记录已经在arp表中被删除。

但是BAS对这些arp表都没有回应。

5310.153952	Zte_74:12:37	Zte_0a:95:a0	ARP	Who has 202.208.3.254? Tell 202.208.3.194
5334.333536	Zte_74:12:37	Zte_0a:95:a0	ARP	Who has 202.208.3.254? Tell 202.208.3.194
5335.333517	Zte_74:12:37	Zte_0a:95:a0	ARP	Who has 202.208.3.254? Tell 202.208.3.194
5336.333501	Zte_74:12:37	Zte_0a:95:a0	ARP	Who has 202.208.3.254? Tell 202.208.3.194
5340.338512	Zte_74:12:37	Broadcast	ARP	Who has 202.208.3.254? Tell 202.208.3.194

5. 在出错的60s中，一直没有收到来自BAS的包。

二、进一步分析故障

1. 故障与FTP下载有关系，在不开启FTP下载时没有这个现象

2. 在本地环境，更换FTP服务器到本地后，故障不再复现。

3. 在测试部环境，更换BCM的版本后，隔一段时间出现一次，每次丢失三个包。

4. 故障和加速器有关系，关闭加速器后故障不在出现。

5. 在PPPoE和IPoE模式下有这个故障，~~在桥模式下没有FTP的错包。~~在实际上桥模式下也会有FTP错包。（**对故障的现象总结归纳不够准确，会导致后期分析走很多弯路。如果桥模式下没有故障，那么故障很有可能是由于连线跟踪导致，南辕北辙，浪费时间。**）

6. 在Tag模式下有FTP错包，untag模式下没有FTP的错包。

二、关于错包：

在测试过程中有一些错包，包结构如图所示。这个包是FTP服务器发给WAN口的（根据PC上面的抓包，每次出现这种错包后，FTP都会重发一个包。且关闭镜像后就没有这种错包了。），应该是带Tag的。

例如： b0:71:ad:b4:22:22 22:22:22:22:00:22 0x2222 Ethernet II

包结构分析：

包前面十几个字节的头部出错了。

0000 00 10 19 11 // 00 24 21 bd 3e a4// 00 10 19 11 22 33//

0010 08 00 //45 //00(DSCP)// 00 38(total length)// e7 56(id) 40 00(flag + offset) 7a (ttl) 06(protocol) 94 b1(checksum) 0a 3f

0020 b8 cb (src ip 10.63.184.203) c0 a8 01 05 (dst ip 192.168.1.5) 00 14 0d e1 dd fe d9 fa 0b 92

0030 a0 b1 50 18 80 00 af 44 00 00 4e ca a5 93 ce e9

0040 be f3 cc 77 ac 02 96 e9 f7 ed

另外一个错包

0000 ~~02 10 18 01~~ 00 24 21 bd 3e a4（目的MAC） 02 10 18 01 00 01（源MAC）

0010 08 00（IP协议） 45 00 05 78（包长度） 88 8b 40 00 7e（ttl） 06 52 3a 5a 01 ..

0020 01 0c（90.1.1.12，源IP） c0 a8 01 05（192.168.1.5目的IP） 00 14 0e aa 95 a5 1a 4e 2b 40

0030 84 80 50 10 ff ff 93 1f 00 00 71 67 00 5d da 8a

0040 a4 02 f6 16 f3 0f 46 9f 9e 0f 25 07 12 d1 26 81

可以看出，这个包是从modem转发出来的，理由：

1. 源IP和目的IP
2. ttl为0x7e，也就126。表示这个包已经经过了一个ttl。

相关Mac地址：

192.168.1.5 00:24:21:bd:3e:a4

br0 00:10:19:11:22:33

202.208.3.43 00:10:19:11:22:36

10.63.184.203 00:19:c6:0a:95:a0

三、关于硬加速

目前Modem加速时90%的包走硬加速，10%的包走软加速，只有极少数包在加速开始阶段走协议栈。

在WAN口驱动中添加了一些打印。运行ftp下载，在PC上抓到171501个包，考虑镜像因素，这个数值和ptm0的计数是吻合的。但是同期在blog_init中添加的计数显示，只有2275个包进到软加速中来了，和ptm0的计数相差巨大，所以可以判断有大量的包没有走驱动。另外，在做完这项测试后，打开print_packet功能，可以看到当前skb的计数为6763，考虑到该计数是上下行两个方向，也可以判断只有大约两到三千个包走到ptm驱动代码中来。

blog_init num=2274

blog_init num=2275

ptm0 Link encap:Ethernet HWaddr 00:10:19:11:22:34

RX packets:22733 errors:0 dropped:0 overruns:0 frame:0

TX packets:67225 errors:0 dropped:0 overruns:0 carrier:67225

packet:kfree_skbmem[370] skb6763, dev:<NULL>

1. 从FTP服务器到Modem，中间经过了DSLAM，如果出错的包是被BAS发出来的，那么其源MAC和目的MAC都是错误的，不可能被DSLAM转发成功。因此可以判断DSLAM发出来的包是正确的。

在DSLAM上showmac，可以看到对应用户线的mac地址是正常的。

在FTP服务器上抓包没有错包。

2. DSLAM发给Modem的包的结构是正确的，但是通过Modem镜像到PC的包的结构却是错误的。
3. **大量的包走硬加速了，但是走硬加速的包是如何被镜像的呢？**配置硬加速时会设置pDevCtx的mirrorIn和mirrorOut字段，没有写寄存器，硬加速应该无法识别吧。

```
static void MirrorPacket( struct sk_buff *skb, char *intfName )
{
    struct sk_buff *skbClone;
    struct net_device *netDev;
    if( (skbClone = skb_clone(skb, GFP_ATOMIC)) != NULL ) //在skb_clone中将blog_p转移到skb_clone中。这里应该使用skb_copy()。因为skb_clone只拷
```

贝skb管理字段，而不会拷贝skb->data。两个skb公用一个data，从而导致加速器修改了data区而镜像工具没有修改skb中相应的字段，从而出现多余的字节。

```
    {
        if( (netDev = __dev_get_by_name(intfName)) != NULL )
        {
            unsigned long flags;


            blog_xfer(skb, skbClone);  //该函数将skb的blog_p再转移给skb。否则eth的驱动再发送该包的时候，加速器会提前将blog_emit中的出包接口记录为eth0。
            skbClone->dev = netDev;
            skbClone->protocol = htons(ETH_P_802_3);
            local_irq_save(flags);
            local_irq_enable();
            dev_queue_xmit(skbClone) ;
            local_irq_restore(flags);
        }
        else
            dev_kfree_skb(skbClone);
    }
} /* MirrorPacket */
```