由PPPOE看Linux网络协议栈的实现

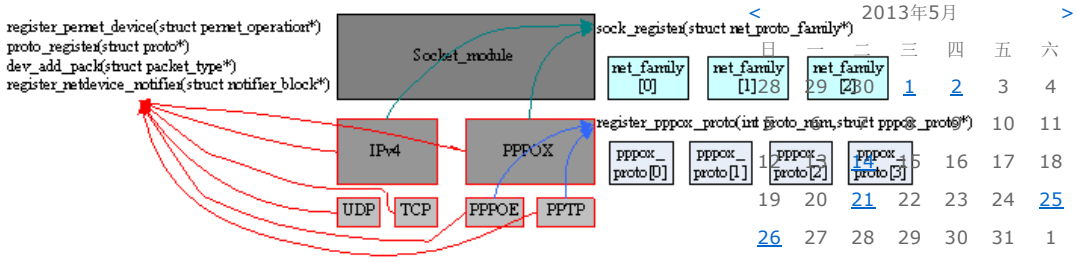
2013-05-01 20:28 by zmkeil, 1522 阅读, 2 评论, 收藏, 编辑

这个标题起得比较纠结，之前熟知的PPPOE是作为PPP协议的底层载体，而实际上它也是一个完整的协议，不过它的实现比较简单，由它出发，可以很容易理清楚Linux网络栈的实现方式。

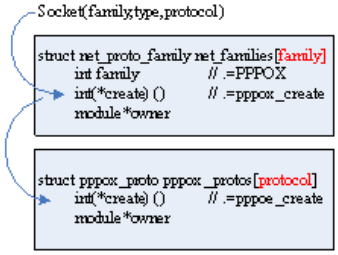
1.总述

Linux中用户空间的网络编程，是以socket为接口，一般创建一个sockfd = socket(family,type,protocol)，之后以该sockfd为参数，进行各种系统调用来实现网络通信功能。其中family指明使用哪种协议域（如INET、UNIX等），protocol指明该协议域中具体哪种协议（如INET中的TCP、UDP等），type表明该接口的类型（如STREAM、DGRAM等），一般设protocol=0，那么就会用该family中该type类型的默认协议（如INET中的STREAM默认就是TCP协议）。

Linux中利用module机制，层次分明地实现了这套协议体系，并具有很好的扩展性，其基本模块构成如下：



先看右边，顶层的socket模块提供一个sock_register()函数，供各个协议域模块使用，在全局的net_family[]数组中增加一项；各个协议域模块也提供一个类似的register_xx_proto()函数，供各个具体的协议使用，在该协议域私有的xx_proto[]数组中增加一项。这两个数组中的存放的都是指针，指向的数据结构如下图所示：



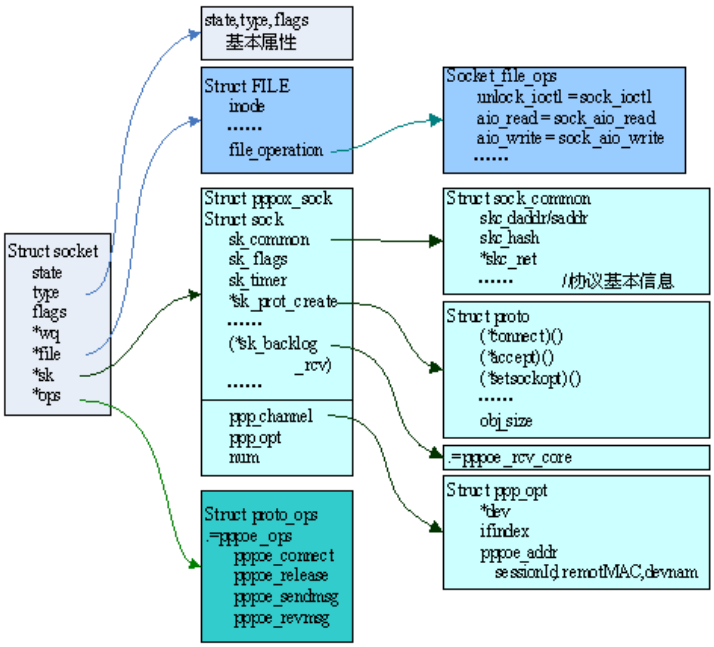
很明显它们是用来创建不同类型的socket接口的，且是一种分层次的创建过程，可想而知，顶层socket_create()完成一些共有的操作，如分配内存等，然后调用下一层create；协议域内的create()完成一些该协议域内共有的初始化工作；最后具体协议中的create()完成协议特有的初始化。具体的下一节讲。

再来看上图右边的，也是顶层socket模块提供的4个函数，前两个一般由具体协议模块调用，由于协议栈与应用层的交互，具体的后面会讲到。后两个一般有协议域模块调用，用于底层设备与协议栈间的交互。但这也不绝对，

如在PPPOE协议中，这4个函数都由具体协议模块调用，这是因为PPPOX协议域内的共有部分不多，各个协议间几乎独立。这4个函数的功能及所用到的数据结构，在后面具体用到时会详细说明。

2.socket插口创建

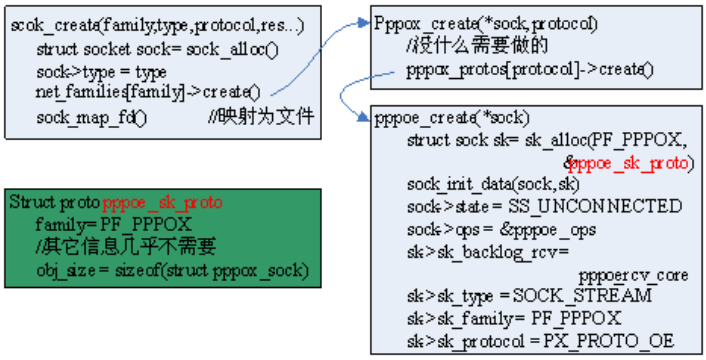
首先来看一下最终创建好的socket插口由哪些部分组成，该结构是相当庞大的，这里只给出框架：



1. 基本属性有state（listen、accept等），flags标志（blocked等），type类型，这里family和protocol都没有了，因为它们再创建时使用过了，已经被融入到socket结构中。
2. File指针指向一个file结构，在Linux中一个socket也被抽象为一个文件，所以在应用层一般通过标准的文件操作来操作它。
3. Ops指向一个struct proto_ops结构，它是每种协议特有的，应用层的系统调用，最终映射到网络栈中具体协议的操作方法。
4. Sk指向一个struct sock结构，而该结构在分配空间时，多分配了一点以作为该协议的私有部分，这里包含了该协议的具体信息，内容相当多。首先是一个struct sock_common结构，包含了协议的基本信息；然后是一个sk_prot_create指针，指向一个struct proto结构体，该结构体就是第一节中所述的，用proto_regsiter()注册到内核中的，它包含应用层到协议栈的交互操作和信息（也可以说成是App → transport layer的交互信息）；然后还有一个sk_backlog_rcv函数指针，所指函数在协议栈处理完接收到的包之后调用，一般仅是把数据包放到该socket的接收队列中，等待APP读取；最后协议的私有部分里存放该协议的私有信息，如pppoe的sessionID、daddr，tcp的连接4元组等，这些信息很重要，利用它们来区分同一个协议中的多个socket。

创建的总体过程，第一节已讲过了，下面以pppoe为例，描述一个socket插口的具体创建过程：

3. Linux下的虚拟Bridge实现(3963)
4. OpenWRT平台搭建及简单应用(3162)
5. Linux下VLAN功能的实现(1967)



之前所述的关键点这里几乎都涉及到了，要注意的是这里的struct proto结构非常简单，因为PPPOE协议几乎没有传输层，所以不需要有太多的中间操作，仅需要一个obj_size来指明struct sock结构后需分配的私有结构大小，关于私有结构的内容，一般在connect操作时才能初始化。

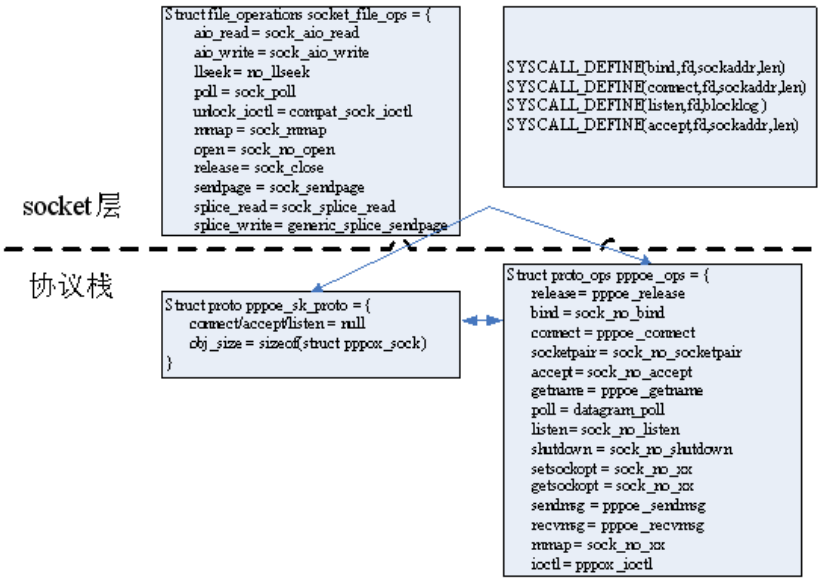
创建好socket之后，其中的fops，proto_ops，sk_backlog_rcv等操作是如何作用，来实现网络通信的功能？这是后面要讲述的内容。

3.主动过程

主动过程即在应用层中通过系统调用，触发socket完成某种动作，有些系统调用和标准的文件操作类似，因此可以直接用sockfd的fops来描述，如read、write、ioctl等，有些则是socket接口特有的，需重新定义系统调用接口，Linux中用SYSCALL_DEFINEn()宏来定义系统调用接口，如bind、accept等。这些系统调用一般都很简单，最终都会去调用socket内部proto_ops中的接口函数。

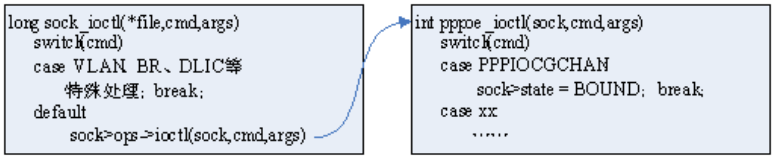
如下图所示，在socket层，并不是所有的文件操作都适用于socket，因此其特有的socket_file_ops中只指定了部分函数；另外还封装了几个系统调用，是我们熟悉的bind、listen、connect、accept。这些系统调用接口都是静态的，它们一般经过简单的处理，就调用具体socket中的proto_ops操作。

在协议栈中，主要是socket特有的proto_ops操作，但对于一些复杂的协议，如TCP，还需要其它一些操作来支持，这些接口都放在struct sock中的struct proto中。PPPOE协议比较简单，不需要struct proto的操作来支持，但其中的obj_size仍然重要，如前所述。



如上图所示，PPPOE协议中，并不是所有协议操作都需要，如bind、accept等，下面选几个来详细看一下socket的主动过程的工作。

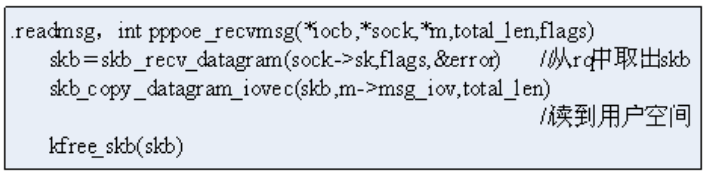
Ioctl系统调用：ioctl是通过标准的文件操作来调用的，具体如下图所示：



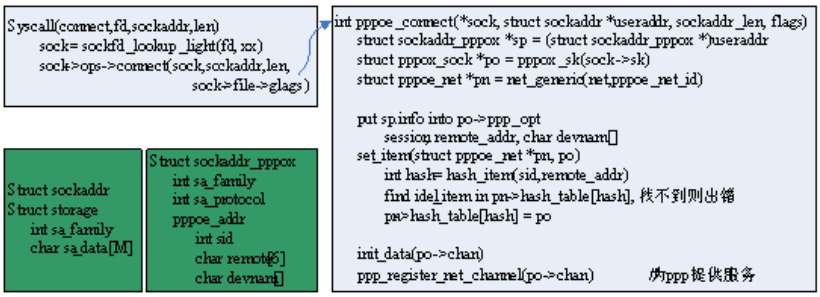
其中顶层sock_ioctl中，对于一些特殊情况，如VLAN、BRIDGE等，它们并不是要对socket插口本身操作，而是要调用VLAN、BRIDGE模块中的创建函数，这看起来有点格格不入，但为了操作方便，且保证网络相关的操作都封装在socket中，这么做也是不得已。

在pppoe_ioctl中，根据cmd进行相应操作，其中有一个值得注意的，就是PPPIOCGCHAN选项，它使得该pppoe_socket成为一个特殊的channel，这主要是pppoe为了给ppp协议提供服务而特有的，与网络协议栈关系不大，以后会具体看。

Read系统调用：read也是标准的文件操作，但要注意，在网络栈中，read并不是接收过程，而仅是从该sock的接收队列中取出skb，提交给应用层，如下图所示。而这些skb是如何获得的，那是一个复杂的被动过程，下面再讲。



Connect系统调用：connect是socket.c中封装的一个系统特用，其代码也很简单，最终调用协议栈中的pppoe_connect接口，该接口函数是pppoe协议中一个非常重要的操作，具体如下图所示：



首先先一下通配地址的问题，这是network programming中一个基本问题，因为各个协议用到的地址结构不同，在应用层，为了方便可读性，可以用协议特有的地址结构，只要符标准的模式即可（即第一个元素为family），然后强制转换成sockaddr*类型，传递给通用的系统调用接口。在最终调用协议模块中的接口函数时，再转换回来。

再看pppoe_connect中，首先由sock结构指针得到pn指针，它们是分配在一起的（如前所述），这很容易得到，同时还得到pppoe_net结构的指针（它是该协议中全局共有的）。然后把用户传递进来的addr的数据放到socket中来，并且执行一个set_item函数，该函数主要根据addr信息，把该socket指针放到协议全局的pppoe_net结构中（这一步对接收过程很重要，后面会细讲）。最后初始化了该socket中特有的chan结构，并调用ppp_register_net_channel()，这主要为ppp服务，以后再看。

4.发送流程

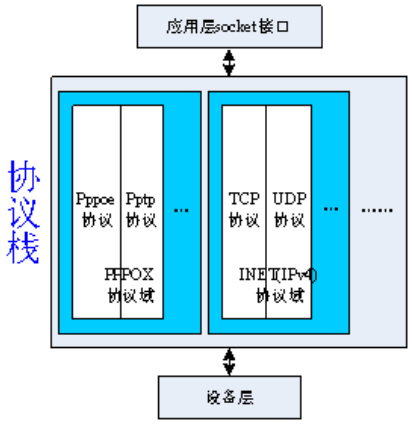
这也是一个主动过程，在协议体系中，它是一个比较重要的过程，所以单独列出来。Socket框架中，发送过程是通过标准的文件操作write完成的，socket的write操作为sock_aio_write()，最终会调用proto_ops->sendmsg()函数，即pppoe模块中的pppoe_sendmsg()，如下图所示：

```
int pppoe_sendmsg(*iocb, *sock, msghdr *m, total_len)
    sk=sock->sk
    po=pppox_sk(sk)
    dev=po->ppp_opt->dev

    skb=sock_wmalloc(sk, total_len+dev->hard_head_len+32+PAD,...)
    dev_hard_header(skb,dev,ppp_ses,po->ppp_opt->remote)      //ether header
    memcpy(&hdr,vertype,sid,code)                             //pppoe header
    memcpy_fromiovec(data_start,m->msg,total_len)             //data

    skb->dev=dev
    dev_queue_xmit(skb)                  //简单的协议，简单的过程
```

首先从sock中获得相关信息，最重要的当然是dev设备，因为pppoe的设备是选定的（由useraddr提供），而有些协议如IP，则会根据协议地址，有协议栈自动选择dev。然后分配skb，并准备其中的package，这是每个协议的关键，由于pppoe协议很简单，只需要设置好一个pppoe header即可。最后直接调用dev_queue_xmit(skb)，通过设备将该package发送出去。

5.网络协议栈结构小结

这里想讲一下的是，pppoe到底是什么层的协议，链路层。而通过上面的描述，更准确的说法应该是，pppoe是一个完整的协议，是从应用层到设备之间的协议模块，从这个意义上来讲，它和INET域中的协议是等价的。如下图所示：



这里讲的协议是从应用层往下直到物理设备的完整过程，有些协议具有一定的相似性，（如TCP、UDP，还包括裸IP等都以IP协议为基础），则把它们归为一个协议域内。至于协议分层，则是概念上的，如PPPOE协议的主要功能体现在链路层，则一般称它为链路层协议， 而狭义上称TCP、UDP为传输层协议（而前面讲的广义上的TCP、UDP则是包括传输层、以IP为基础的网络层、链路层的完整协议）。

有点饶人，不过没关系，只要理解协议栈的功能就是从socket接口得到数据，封装成一定的包结构，最终由物理设备发送出去（接收过程反过来）。至于具体的实现，则是由具体协议的特点决定的，对于一些复杂协议，分层方式则是一种比较好的选择。

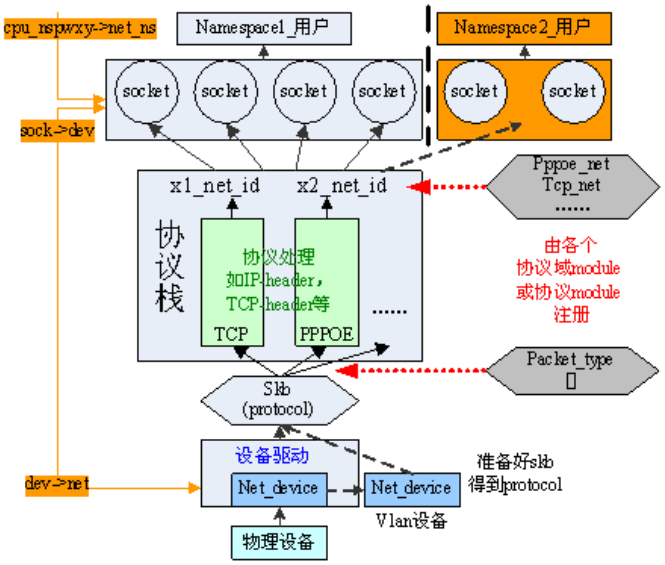
而其中有些协议会比较特殊，如之前讲的VLAN，它甚至从来都不会进入到协议栈，仅在设备驱动层，就被转化成以太网协议，协议栈中根本不需要为

它准备处理接口。再如比较典型的ICMP协议，它既可以是一个完整的协议，被应用层调用（如典型的Ping程序），也可以只作为TCP的附属协议（只被TCP处理，对应用层不可见）。这里的PPPOE与此很类似，本文讲述了其作为完整协议的工作方式，另外它也可以作为PPP协议的底层基础，在下一篇中会讲述其具体的实现方法。

6.被动过程-接收流程

接收过程是一个被动过程，在屋里设备层，它往往是由中断触发，其实现的复杂度也较发送过程高很多。在协议栈中，其实现也同样与发送过程很不对称。因为发送时，本身主机拥有控制权，而接收时，是一个数据包对多个接收模块（一对多），只能从数据包中的信息中一点一点分析，并去寻找接收模块。

先给出接收流程的框架，再逐步去分析其实现。如下图所示：



先不看橙色部分，一个接收流程由物理设备的中断触发，设备驱动程序进行相应处理，得到协议栈中标准的数据结构sk_buff（简称skb），并根据一个特殊的全局数据结构packet_type，将数据交给相应的协议；协议根据自身设计特点对skb数据进行处理，并通过全局变量xx_net_id和各个协议私有的特殊数据结构xx_net，寻找到该数据包对应的应用层socket插口，并将其放在该socket插口的接收队列中；最后应用层在某个时刻会通过read系统调用读取该数据（如第3节所讲）。

6.1设备驱动层的处理

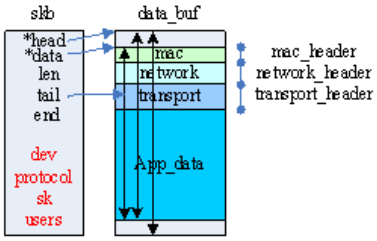
设备驱动层的接收过程在之前的篇章中已经讲过了，一般是由硬件中断触发，然后或是采用中断模式、或是采用NAPI模式，总之其根本任务就是：根据设备的特点（先验知识，如以太网设备驱动事先就是知道以太网帧的基本结构的），将接收到的裸数据转换成协议栈所认识的标准结构skb（从而实现底层设备对上层的透明性），然后提交给相应的协议。很明显问题有两个，skb是什么样的，要准备什么？怎么知道提交给谁？

准备**skb**结构。首先来看一下sk_buff的构成，如下图所示。Skb只是一个控制结构，实际的数据放在一个data_buf中，并由skb中一些列参数索引，具体见下图右所示，这之中有些参数是在分配data_buf、copy数据时就决定的，如head、end、data、tail等；有些则要经过一定的识别才能得到，如mac_header一般在设备驱动中得到，而network_header、transport_header则要到协议栈中才知道，且各个协议的处理各不相同，如PPPOE协议根本不

需要只需要指明network_header，而TCP协议则有复杂的头部信息。最终由skb->data指针和头部长可得到app_data的位置，因此应用层可以只读取应用数据即可。

Skb中另外一些参数也相当重要，如vlan_tci用于指明vlan的id，其用法在前面已讲过。dev参数则是要贯穿整个流程的，因为该庞大结构中的多个信息会在整个网络系统中用到，要注意的是该参数由设备驱动程序决定，一般就是接收的物理设备，但在Linux中，网络设备是由net_device结构指示的，一个物理设备可有多个协议设备，这在VLAN、BRIDGE中很明显，其实在PPP协议中，这也是一个关键点，后面会讲到。Sk参数指示了该数据包属于哪个应用层socket插口，它由具体协议根据特定方法得到，后面会讲到。Protocol参数是本节的重点，它由mac_header中的字节决定。

```
Struct sk_buff {
    sk_buff *next/*prev
    struct sock *sk          //协议栈中搜索
    net_device *dev          //何设备接收的
    uint len,data_len
    __u16 mac_len,hdr_len
    ......
    __be16 protocol          //是何协议
    __u16 vlan_tci           //用于VLAN的
typedef uint sk_buff_data_t
    sk_buff_data_t transport_header, network_header
                   mac_header, tail, end
    uchar *head,*data        //指示数据位置
    ......
    uint truesize            //data_buff的实际大小
    atomic_t users           //使用者数
```

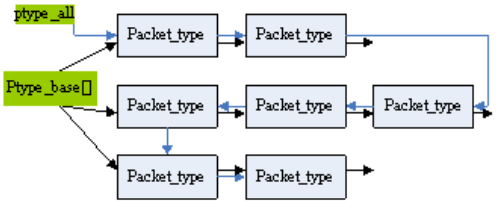


设备驱动程序只关心mac_header，即数据包最初始的部分。前面也提到了，这需要一定的先验知识，如ethernet设备驱动，它先验的指导以太网头部由DMAC、SMAC和两字节的协议构成，下面是一个RTL8012驱动的接收片段（~/dev/net/Ethernet/realtek/apt.c）：

```
apt.c  net_rx(dev)
    read_block(ioaddr,8,&rx_head,dev->if_port)
    int pkt_len = (rx_head.rx_count&0x7ff) - 4      //具体硬件决定
    *skb = dev_alloc_skb(pkt_len+2)                 //分配skb和data_buf
    read_block(ioaddr,pkt_len,skb_put(skb,pkt_len),dev->if_port)  //读数据到skb中
    skb->protocol = eth_type_trans(skb,dev)         //根据先验知识得到协议
                    x25_type_trans(skb,dev)         //用于x25协议的
    skb_reset_mac_header()                  //设置mac_header,使协议栈中只处理上层数据
    netif_rx(skb)                           //提交给协议栈
```

提交协议栈。主要就是根据skb->protocol参数，当然还需要另一个重要的数据结构packet_type。

```
Struct packet_type
    __be16 type              //protocol
    *dev                     //设备
    int(*func)(*skb,*dev,*ptype,*odev)
    ......
    list_head list
```



设备驱动中最后提交过程有netif_skb_receive()函数完成，它会遍历系统中所有的packet_type，找到protocol和dev（这个是啥意思）都相同，就调用该ptype中的func函数，如ip_packet中的func为ip_rcv()函数，这样skb就到了协议栈中。

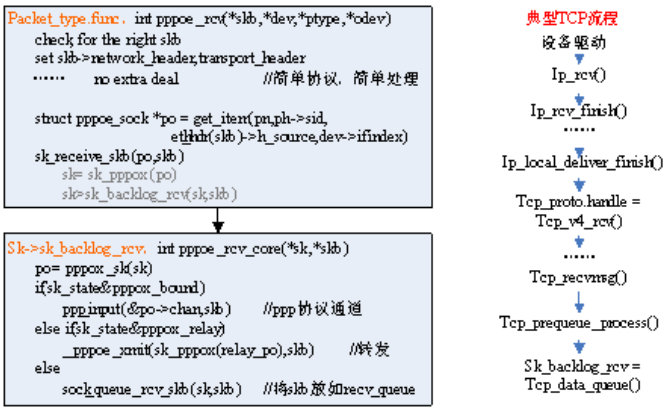
系统中所有全局的packet_type构成一个list，并由全局变量ptype_all索引，另外还提供ptype_base[]全局数组，将type相同的packet_type单独成链，为遍历提供方便。

这些全局的packet_type结构是从哪来的，这就要看第一节图中，左边4个函数中的一个dev_add_packet(struct packet_type*)。协议模块在加载时，调用该函数，将自己特有的packet_type结构注册进内核中，而其中的(*func)则有协议自己定义。

最后要注意的是，打开if_ether.h文件，可以看到现在已定义的协议protocol有_P_IP、_P_ARP、_P_8021Q、_P_PPP_SES、_P_PPP_DIS等，如果根据传统的分层协议来看它们，会觉得很乱，有网络层的、链路层的、甚至同一种协议还有两个，但如果用第5节的概念来看，则很容易理解。再看由什么模块注册，TCP、UDP都是以IP协议为基础，只要有INET协议域模块注册一个即可，而ARP虽然也属于INET域，但它却必须自己有一个packet_type，PPPOE协议虽然只是一个协议，但却有两个阶段，所以它有两个不同的packet_tpye。可见这种实现是很灵活的，根据具体协议的特点决定。

6.2协议栈接收处理

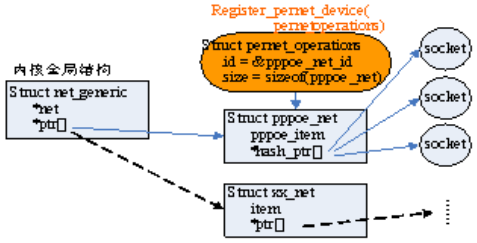
协议栈的处理由各协议决定，如TCP协议的处理过程是相当复杂的，而这里的pppoe的处理却非常简单，但由它却可以避开细节，更清楚地看到流程的梗概，如下图所示：



可以看到pppoe协议的处理过程几乎没有，仅是设置了skb的network_header, transport_header，然后就利用get_item()函数找到它所属的socket插口，直接把它提交给上层。如上图右所示，是典型的TCP接收流程，是相当复杂的，其中TCP与IP的接头处还需用到额外的私有数据结构。

提交函数sk_backlog_rcv，即这里的pppoe_rcv_core(sk,skb)函数，首先判断是否为ppp通道的数据，若是则提交给ppp协议。一般正常情况下，直接用sock_queue_rcv_skb(sk,skb)函数将它放在socket的接收队列中。

匹配应用层接口：协议栈在对数据包进行处理后，需要确定该包属于哪个socket插口，这个过程在内核中有一套完整的机制来完成，其框架如下图所示：



首先内核有个全局结构net_generic，其中一个最重要的元素是指针数组。然后每个协议module加载时，会调用register_pernet_device(struct

pernet_operations*)（见第一节图），pernet_operations结构中最关键的两个参数，一个是size，它指示内核为该模块分配一个私有数据结构（如pppoe即为struct pppoe_net），另一个是xx_net_id，它指示由net_generic.ptr[xx_net_id]来指向该数据结构，这样每个协议模块中，就可以根据自己的xx_net_id很容易寻找到内核分配给自己的私有结构。最后协议的私有模块中一般也有一个指针数组，用以索引属于它的各个socket。

工作流程就很清楚了，具体的工作方式还要看两个函数，

```
int set_item(struct pppoe_net *pn, struct pppox_sock *po)
int get_item(pn,ph->sid,eth_hdr(skb)->h_source,...)
```

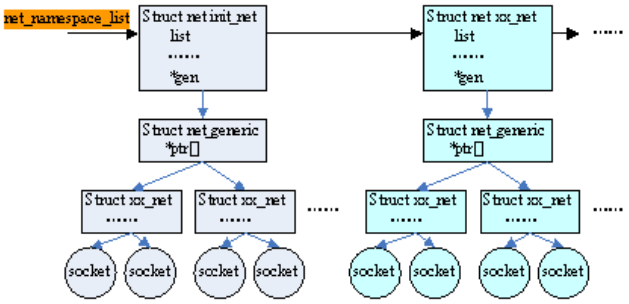
就不看细节了，仅看两个函数的原型就能明白，其中pn参数就是上面所述的用全局结构net_generic和各协议私有xx_net_id获得的。Set_item()函数在connect时调用（参见第3节），它根据pppox_sock中的sessionID、remoteMAC（这两个参数由*useraddr传入，详见下一篇协议分析），根据一个hash算法得到一个hashInt值，然后用pn->hash_ptr[hashInt]指向该socket结构。那反过来，接收时由这两个参数（由数据包的协议头中获得）得到hashInt，便能很容易找到对应的socket了。

各个协议使用的方法及参数都不同，但思路都一样，就是依据协议本身特有的参数（如TCP中的连接4元组），在socket创建、或连接的时候（接收数据之前），根据一定的算法，将它的指针放在该协议私有的xx_net中，这样接收时就可以由数据报的协议参数找到它了。

6.3命名空间namespace

上述的socket索引方法有个绕弯的地方：就是每个协议私有的xx_net结构可以直接由协议模块本身分配，索引起来也方便，不要用到全局的net_generic。而目前内核所用的方法，其实是为了另外的目的，那就是命名空间namespace。也就是虚拟多用户的一套机制，具体的也没细看，好像目前内核整个namespace还没有全部完成。

network的命名空间问题主要在于，每个协议模块的xx_net私有结构不仅是一个，而是由内核全局决定的，即每注册一个新的用户（有点像虚拟机机制），就分配一个新的xx_net结构，这样多用户间可以用参数相同的socket连接，但却指向不同的socket。



可以看到前面所述的很多内容中，都会有个net参数，就是为了这个作用，主要实现函数在namespace.c中。

7.总结

主要结合pppoe协议，学习了Linux中网络栈的实现。由于pppoe协议本身很简单，代码量少，更容易抓住协议实现的梗概。Linux网络栈，继承Unix，采用socket插口作为主线，主要包括创建、协议连接、主动过程、匹配机制、被动过程等内容。

要注意的是，实际应用中，很少有直接利用pppoe协议通信的，而是把它作为ppp协议的底层基础来用，而这需要协议实现中的一些技巧来支持，下一篇中讲述。

zmkeil
关注 - 0
粉丝 - 37
+加关注

1 0

« 上一篇：Linux下的虚拟Bridge实现
» 下一篇：PPP协议体系的实现

分类：网络相关

#1楼 woainilsr
2013-07-18 02:26

ADD YOUR COMMENT

写的真好，这些代码我都看过，但是不能像你这样从全局功能上来分析，所以之前看了也收获不多。真的很希望能向你学习这种分析问题的角度和方法。

支持(0) 反对(0)

#2楼[楼主] zmkeil
2013-07-30 16:43

@ woainilsr
呵呵！谢谢，有兴趣可以多交流交流，我的qq：805920692

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

· 程序猿媳妇儿注意事项
· 可是姑娘，你为什么要编程呢？
· 知其所以然（以算法学习为例）
· 如何给变量取个简短且无歧义的名字
· 编程的智慧
» 更多知识库文章...