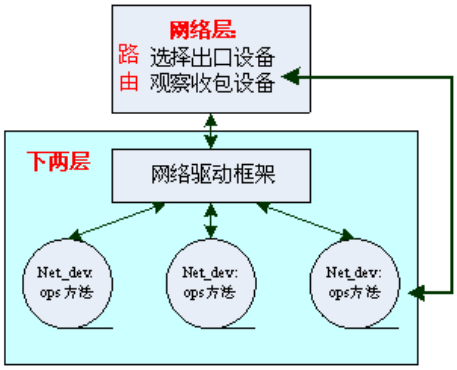
Linux下VLAN功能的实现

2013-04-18 20:19 by zmkeil, 1967 阅读, 0 评论, 收藏, 编辑

1.Linux网络栈下两层实现

1.1简介

VLAN是网络栈的一个附加功能，且位于下两层。首先来学习Linux中网络栈下两层的实现，再去看如何把VLAN这个功能附加上去。下两层涉及到具体的硬件设备，日趋完善的Linux内核已经做到了很好的代码隔离，对网络设备驱动也是如此，如下图所示：



这里要注意的是，Linux下的网络设备net_dev并不一定都对应实际的硬件设备，只要注册一个struct net_device{}结构体（netdevice.h）到内核中，那么这个网络设备就存在了。该结构体很庞大，其中包含设备的协议地址（对于IP即IP地址），这样它就能被网络层识别，并参与路由系统，最有名的当数loopback设备。不同的设备（包括硬件和非硬件）的ops操作方法各不相同，由驱动自己实现。一些通用性的、与设备无关的操作流程（如设备锁定等）则被Linux提炼出来，我们称为驱动框架。

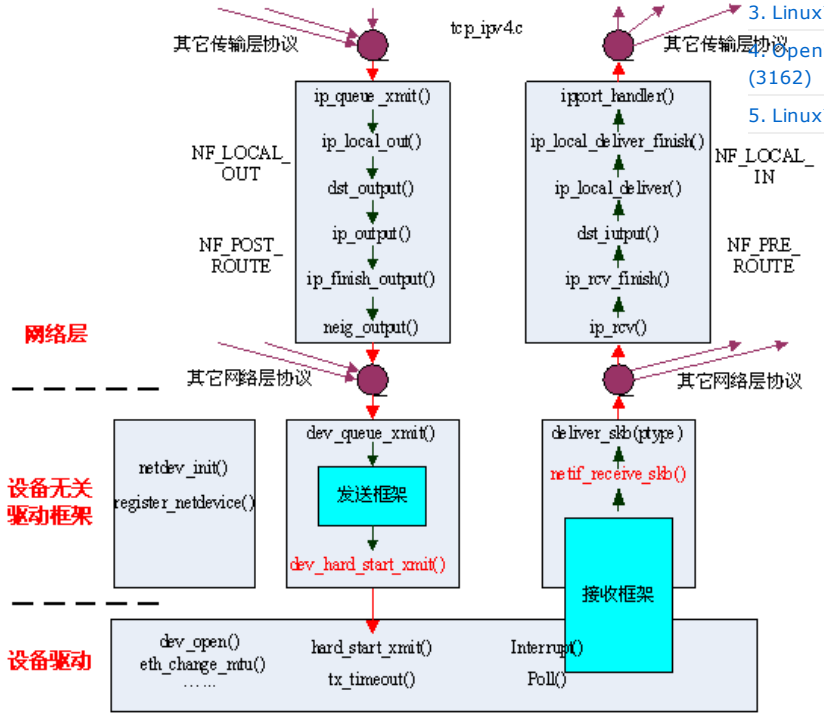
1.2代码框架

就是对于上图的扩展，从代码的角度看网络栈的实现。这里主要是学习的过程，一方面算是赏析Linux优美的代码结构，另一方面只有了解这些，才能更好地写网络设备的驱动，或者做平台移植。

与网络相关的代码主要在~/net，框架性的代码在~/net/core中，另外很多结构定义、宏、简单内联函数在~/include/net、~/include/linux中，具体设备的驱动在~/driver/net中。代码量很大，这里仅给出一些关键的代码流程，且有些流程比较复杂，放到下一节描述。如下图所示：

About

昵称：zmkeil
园龄：3年3个月
粉丝：37
关注：0
+加关注



网络层的代码比较清晰，实际上还有一个forward流程（即路过主机，传向他处），这里没画出，其中NF_函数就是Netfilter框架的钩子函数。这里以IP协议为例，发送流程的代码大多在ip_input.c文件中，接收流程的代码大多在ip_output.c文件中。其它网络层协议如IPv6、X25等，流程大致相同。各种协议在发送流程的最后，都会主动调用dev_queue_xmit()函数；而设备接收到数据包后，会根据包的类型，传送给相应的协议函数，如ip_rcv()，当然这里的实现还是比较复杂的，设计到一些全局的数据结构，不是重点，没看。

驱动框架的代码基本都在~/net/core/dev.c中。其中的代码分为3部分：

1. 全局性的代码，如netdev_init()是在系统启动时初始化网络环境的（注意并不是初始化具体的设备），register_netdevice()函数是添加\注册网络设备时调用的，它们中的一些细节直接关系到设备的工作过程，下一节针对具体模块时分别讲述；
2. 发送框架相关的，由上层调用dev_queue_xmit()函数，经过一系列处理（包括锁定设备、选择队列、**vlan**相关的处理等），最终调用设备的hard_start_xmit()函数，由它完成硬件的发送过程；
3. 接收框架相关的，因为接收是一个被动过程，一般通过中断来发起，但为了提高性能，Linux中的中断处理一般分为两部分（时间紧急的和时间不紧急的），即典型的UH+BH模型；另外近年来，人们发现大数据量时，连续的中断有损性能，现在越来越多的驱动都改用NPI接收模型，将BH部分直接在驱动中实现，比较复杂。不过不管通过什么流程接收到数据包后（封装成skb），都会把它交给netif_receive_skb()函数，该函数对数据包进行处理（包括**vlan**相关的），最终通过deliver_skb(ptype)交付给相应的上层。

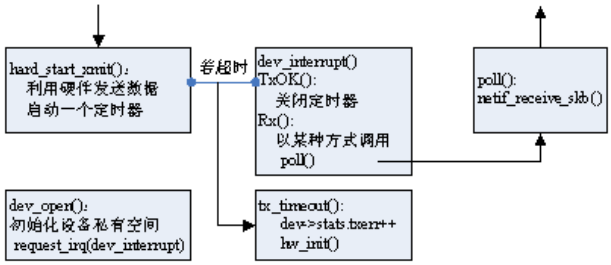
设备驱动的代码（即netdev->ops所指向的函数），各个设备不同，其中最最重要的有5个：dev_open()，hard_start_xmit()，tx_timeout()，interrupt()，poll()。另外其他一些函数如设置mtu，更改mac等，则根据具体设备的功能选择实现。

1.3代码细节

以tealtek的rtl8169驱动为例，首先介绍一般的设备驱动中实现那些功能，以及这些功能是如何组合起来的。然后分别从发送、接收流程出发，分析驱动框架中的代码是如何支持这些功能的实现的。

1.3.1设备驱动的功能组合

前面讲到了，设备驱动中最最重要的5个函数，这些函数有机组合在一起，实现了可靠的设备功能，如下图所示：



打开函数dev_open()中，首先初始化设备的私有空间。每个网络设备有一个net_device结构体，同时还有一个私有结构，由net_device.priv指针指向。在module_init()函数中，一般会调用netdev_alloc(sizeof(priv),name,setup_func)函数，该函数指明设备的唯一名称及一个初始化函数（对于以太网，一般用ethe_setup()），同时申请net_device结构和private结构的空间。Private结构由不同的设备自己决定，在dev_open()中，应初始化之。

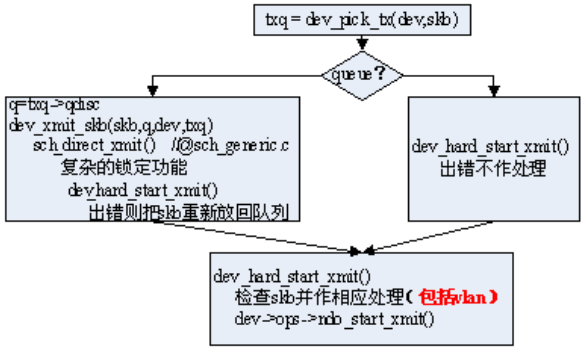
发送函数hard_start_xmit()中，首先利用硬件发送数据，注意这里仅是把数据写入设备的发送缓存中（或有些设备直接是利用dma的），然后写相应的寄存器，通知硬件开始发送，之后该函数就正确返回了，然后硬件到底有没有正确发送数据还不知道。

中断函数interrupt()，是在dev_open()时申请的，并根据实际硬件的中断号，与某个中断线联系并注册进内核。当该中断线上有中断时，CPU跳转到该函数执行。虽然是一根中断线，但设备中断的类型却不一样，这有具体设备决定，一般可通过读取硬件的状态寄存器获悉。若是接收中断，则以某种方式去调用poll()函数，把数据包传递给上层。

1.3.2发送流程细节

首先需要知道，Linux为每个网络设备准备了发送/接收队列，alloc_netdev(sizeof(priv),name,setup_func)实际上被定义为alloc_netdev_mqs(x,x,x,1,1)（netdevice.h），即默认为每个设备分配一个发送队列和一个接收队列，队列结构为struct netdev_queue，每个队列中有个重要的结构struct Qdisc。该结构的功能主要是提供多进程使用同一个设备时的锁定功能，在SMP架构（或多核架构）的机器中，这种锁定功能的实现变得尤为复杂，这也是现在内核设计的关键和难点，暂且不管。

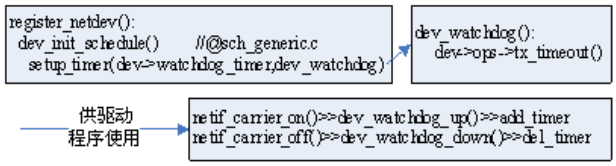
现在来看网络设备的发送流程，如下图所示：



对于没有队列的设备（主要是一些虚拟设备，如loopback），处理比较简单。对于一般设备，主要对它进行一些复杂的锁定功能，而且函数调用出错时需把该skb重新放回队列中。最终都会调用dev_hard_start_xmit()函数，该函数是发送流程中的关键，它是设备无关的，主要会检查并处理skb中的各种特性，一些新功能（如vlan）的实现都在这个函数中完成。该函数最终调用设备驱动中的设备相关的发送函数。

前面讲的出错，仅是函数调用的出错。正如前面讲述的，即使函数调用正确返回了，也并不代表硬件成功把数据包发送出去了。所以一般网卡设备都会在设备成功发送数据时产生中断，并在相应的寄存器中显示这是个TxOK中断。

Linux的网络设备驱动框架中，很好地利用了这点。每个设备的net_device结构体中都有一个watchdog_timer，在module_init()中注册该模块时，register_netdev()函数中会初始化该定时器，并注册其func为dev_watchdog()，该函数的内容就是运行设备驱动中实现的tx_timeout()。另外内核提供打开、消去该定时器的函数，供驱动程序在相应的位置使用。

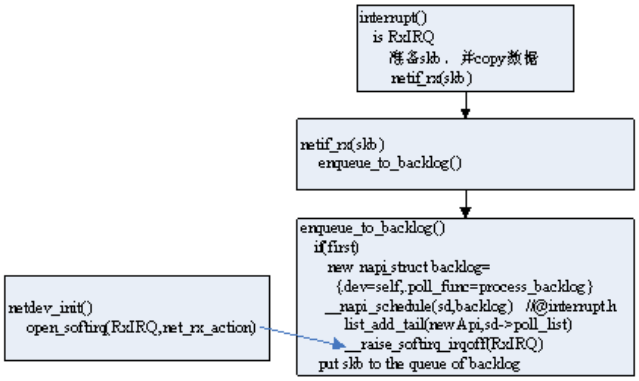


1.3.3接收流程细节

接收过程是被动触发的，一般由硬件的中断引发。Linux在处理这种IO时一般采用典型的UH+BH模型，即把一些实时性高的操作（如把设备缓存中的数据copy到内核中，以便设备可接收其它数据）发在中断处理函数中完成，而把实时性要求不高的操作（如处理数据）发在稍后的时间里完成（一般是另开一个线程）。

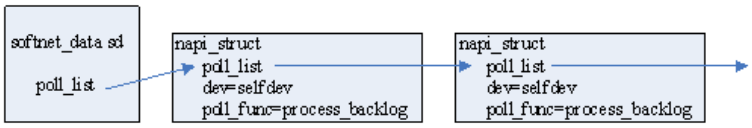
在经典的网络设备驱动中也经常使用这种模型，首先介绍两个数据结构，分别是struct napi_struct{}，里面主要有拥有该数据结构的设备的索引dev，和一个函数指针poll_func；另一个是struct softnet_data{}，该结构中主要维护一个napi_struct的队列。

内核准备了一个全局的struct softnet_data sd结构（实际上是为每个cpu准备了一个），另外准备了一个通用的poll_func函数process_backlog()。好了，现在来看驱动中的BH部分，如下图所示：

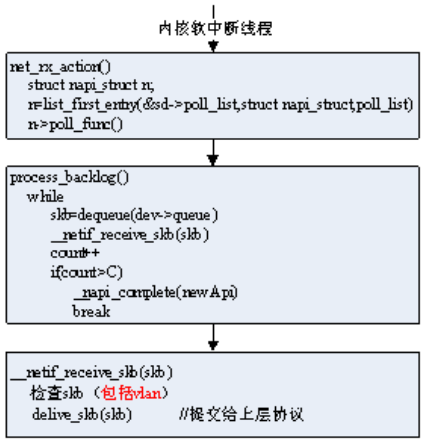


设备驱动中的interrupt函数，检查到接收了新数据包时，就准备新的skb，并把数据copy到skb中，然后调用驱动框架中的netif_rx(skb)函数；该函数主要调用enqueue_to_backlog()；该函数检查是否初次进入，是则准备一个新的napi_struct结构，其poll_func定义为通用函数process_backlog()，并调用__napi_schedule()函数，把准备好的结构体放入sd的poll_list中，然后调用__raise_softirq_irqoff(RxIRQ)打开软中断，且以后每次进入都把skb压入backlog的queue中。

这里要检索另一个函数netdev_init()（在系统启动时调用的），上述讲的sd结构就是在这个函数中分配的，另外该函数还注册了软中断函数net_rx_action()，软中断的原理没去看，应该就是利用Linux内核的tasklet机制实现的。__raise_softirq_irqoff(RxIRQ)函数讲软中断掩码mask中的RxIRQ置位，这样，BH部分就完成了，此时的sd结构如下图：



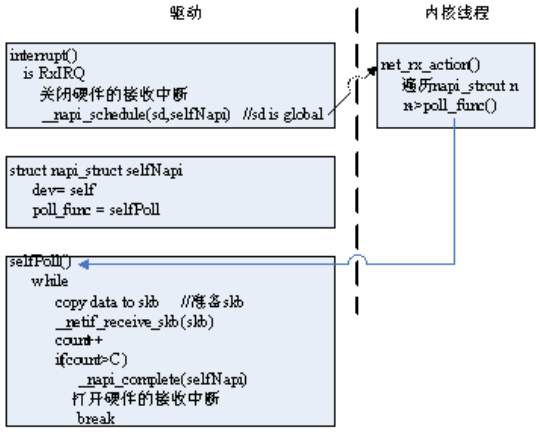
之后就是UH部分了，即系统在之后的某个时间，启动软中断线程，执行net_rx_action()函数，该函数遍历softnet_data sd结构中的poll_list，并执行每个napi_struct->poll_func()函数，由前面的叙述可知，这里的poll_func()函数都是process_backlog()，该函数采用while循环取下dev上的skb（因为在软中断执行前可能发生了多次接收中断），并调用__netif_receive_skb(skb)函数，讲skb传递给上层协议。当接收到一定数量的包后，就认为本次数据包接收完毕了，并把该napi_struct结构从sd中删除，如下图所示：



这就是传统的中断方式，可以参见RTL8012的驱动~/driver/net/ethernet/realtek/apt.c，就是利用的该方法，它的优点是，需要驱动程序 做的非常少，仅需准备好skb，调用net_rx(skb)即可，其它都有

驱动框架完成。缺点是，欠灵活，且数据量大时，会不停的中断，影响系统性能。

现在很多网络设备驱动已不再使用这种结构，而是采用NAPI结构，它完全摒弃了内核驱动框架中的UH+BH模型，并且不再用中断方式，而是在驱动内部使用轮询方式。



与中断方式最大的不同在于，每次发生接收中断时，关闭接收中断，启动软中断，在poll函数break前，重新打开接收中断，一遍下一轮的数据接收。其次，驱动程序自己定义napi_struct结构和poll_func函数。最后，poll_func函数和前面讲的在结构上差不多，都是while循环，但它要自己准备skb（因为它之前没有中断程序来准备skb），并且直接上传该skb，一般不会实现队列queue（因为它之后没有其它线程再去处理queue了）。

这就是所谓的NAPI方式，它避免了多次的硬件中断，一定程度上提高系统系能。但驱动程序也因此更加复杂，并且poll_func()函数中要做的事太多（摒弃了UH+BH模型），在数据量很大时，会出现丢包的现象（这好像是Linux的一个bug）。Rtl8169就是采用的这种方式，参见~/driver/net/Ethernet/realtek/r8169.c。

2.Linux中VLAN的实现

2.1Linux网络中的namespace

这个概念我不是了解的很清楚，不过可以简单地把它看成是一种分类，目前所了解的网络设备有3类：传统的网络设备，它们不需要依赖于其它设备而独自存在，如etho、loopback等；VLAN网络设备，它需要依赖于一个宿主设备，若宿主设备没了，它是不能工作的；Bridge网络设备，它也是虚拟的，它依赖于从设备。

与此相关的结构有struct net{}，相关文件包括namespace.h、namespace.c等。这3类网络设备都是以module的形式被加入内核中，它们可以看成是网络子系统的顶层module，下面实现的驱动模块等都依赖于它们。这3个顶层模块加载时分别执行的init函数为：netdev_init(),@dev.c；vlan_proto_init(),@vlan.c；br_init(),@br_device.c。

这3个函数中有自己特有的部分，如netdev_init中分配softnet_data等，它们也有相似的部分，如

```
Register_pernet_subsys(&net_ops)          //init the namespace,@namespce.c
Register_netdevice_notifier(&notifier_ops) //@namespace.c
Ioctl_set_func(&ioctl_handler_ops)         //@net/socket.c
```

这里要重点看的是ioctl_set函数，这涉及到Linux下网络设备的ioctl操作。在Linux中，所有网络设备的ioctl操作都被抽象成对/proc/net/下的文件的操作，最终调用内核中的sock_ioctl函数，该函数结构如下：

```
Sock_ioctl(*file,cmd,arg)
        case vlan
              vlanioctl_hook(cmd,arg);
              break
        case br
              brioctl_hook(cmd,arg);
              break
        case dlic
              ...
```

```
vlan_ioctl_hook    = vlan_ioctl_handler
br_ioctl_hook      = br_ioctl_deviceless_stub
……
```

其中各个hook函数就这里init()时利用ioctl_set_func()设置的。这种设计架构大大方便了用户空间对各类虚拟设备（如vlan，br等）的操作，如目前Linux下vlan的操作命令vconfig就是打开/proc/net/vlan/config文件，然后对它进行ioctl操作，详细参见vconfig的源码（非常简单）。Br也是差不多，以后学习br时再细看。

注意：这里关于namespace的概念可能错了，现在先不看，后面讲到协议族时，再一起看看整个网络栈顶层的实现框架，这里先关注底层的设备。另注：这里的vlan_ioctl的概念可能是错误的，它实际上是sock_ioctl的特殊情况，以后再看吧，包括应用层如何调用到它。

2.2VLAN的实现

Vlan的分析，主要从其ioctl入手，一步步看其源码就能大致理解了，为了叙述方便，这里首先给出我所理解的vlan实现框架，再去叙述其实现细节。

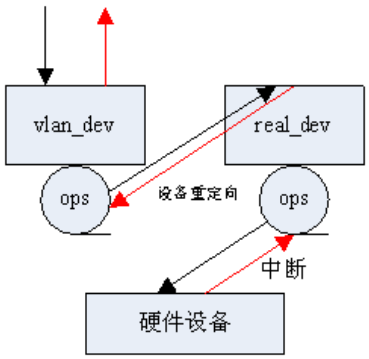
2.2.1Vlan的功能框图

如前面所述，Linux中VLAN是一种特殊的设备，首先简单看一下vconfig命令创建一个VLAN设备

vconfig add eth0 10

VLAN设备必须依赖于一个实际的宿主设备，并制定一个vlan_id，这样就创建出一个eth0.10设备。创建好后，就可以和实际网络设备一样，用ifconfig命令配置它。

它发送/接收数据的流程大致如下图所示：

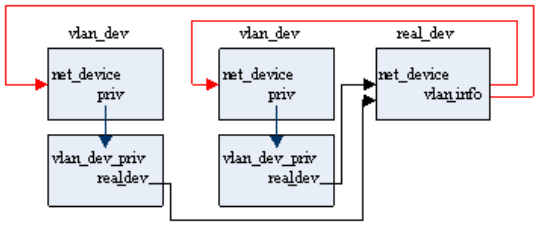


通过vlan_dev发送时，首先会调用它自己的驱动中的ndo_start_xmit()函数，就仿佛它是一个实际设备一样，而它的发送函数会将skb重定向到real_dev，并利用real_dev重启发送流程，这是内部实现的，后面会讲到，

且对上层是透明的。

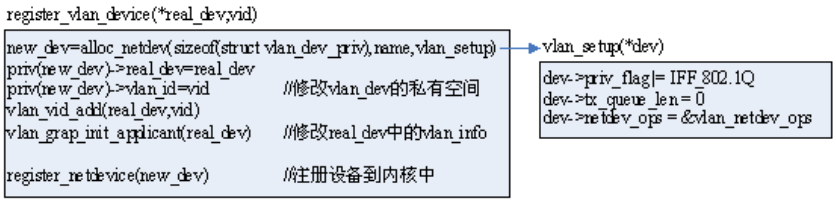
接收是有硬件中断触发的，所以一定是由real_dev的驱动接收到数据并打包成skb，若发现该数据是vlan的，则重定向skb->dev=vlan_dev，然后提交给上层。对上层而言，这也是透明的，就仿佛是vlan_dev收到了数据。注意vlan_dev的硬件地址必须和real_dev相同，这样，发往vlan_dev的数据包才能被实际的硬件设备接收到。

相关数据结构框图如下。Vlan设备的priv结构中有real_dev指针，同时实际设备中的vlan_info信息指明它所有的vlan设备。



2.2.2Vlan设备的创建

前面讲了，通过vconfig add命令可以创建一个vlan设备，该命令实际上是对/proc/net/vlan/config文件的ioctl操作，映射到内核中就是vlan.c中的vlan_ioctl_handler()函数，add命令最终调用register_vlan_device(*real_dev, vid)。



首先申请了一个新的struct net_device结构作为vlan设备，并为它分配一个struct vlan_dev_priv型的私有空间（vlan.h），并指明它的初始化函数为vlan_setup。该初始化函数设置该dev的flag为802.1q；并设置它的发送queue为0，这一点对vlan的发送流程很重要；设置其netdev_ops为一个通用的vlan_netdev_ops，它直接决定了vlan设备的工作方式，后面会细讲。

然后修改vlan设备的私有空间，指明它的宿主设备及vid；并且相应地修改宿主设备real_dev中的vlan_info信息。

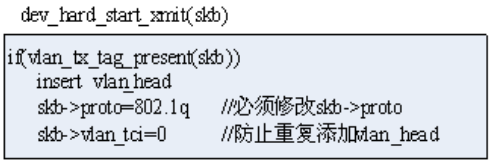
最后把它注册进内核中的netdevice链表中，从此它对上层协议栈而言，就仿佛是一个实际的设备，和其它所有设备有平等的地位，可以用ifconfig配置它，也可以把它加入bridge等。

相关函数集中在vlan.c中，里面还有其它一些ioctl功能函数；另外vlan_core.c中主要是和vlan相关的核心操作；vlan_dev.c中主要是vlan设备相关的代码。

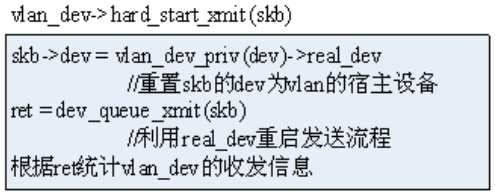
2.2.3Vlan设备的发送流程

Vlan设备对上层协议栈而言，和实际设备时平等的，所以它也会参与路由选择，若vlan设备被选中为出口设备，那么上层最终会调用dev_queue_xmit(vlan_dev)来发送数据，参见1.3.2节的图。上一节讲了，vlan设备的tx_queue被初始化为0，所以发送流程会直接调用hard_dev_start_xmit()函数，该函数首先对skb作一系列检查，包括vlan的检查，然后调用skb->dev->ops->ndo_start_xmit()发送。

首先来看对vlan的检查，参见dev.c中的hard_dev_start_xmit(skb)函数，其实很简单，检查skb中的vlan标志，若有，则插入vlan_tag，并修改skb->proto=802.1q，最后去除skb中的vlan标志。skb中为什么会有vlan标志，因为上层选择vlan_dev后，根据它的priv_flags（见上一节）可知道它是一个vlan设备，因此给它打上一个vlan标志。

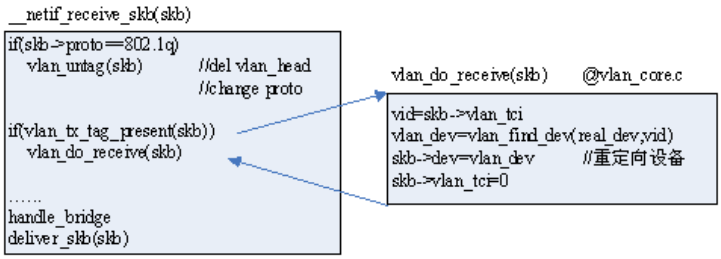


然后来看vlan设备的驱动中的发送函数，有上一节知道，所有vlan设备的netdev_ops都被初始化为vlan_netdev_ops，它的发送函数为设置为vlan_dev_hard_start_xmit()（vlan_dev.c）。也很简单，如下图所示



2.2.4Vlan设备的接收流程

在1.3.3节讲了网络设备的接收流程，不管采用中断方式，还是NAPI方式，最终都会准备好skb，并在一个内核线程中调用__netif_receive_skb(skb)函数，该函数检查skb，包括vlan的检查，然后把skb提交给上层。



若接收到的skb是802.1q协议的，即mac地址后面跟了0x8100，注意，网卡接收的仅是bit流，这里只能从bit流中的特定字节来判断它是否是vlan包。若是vlan包，则调用vlan_untag()函数，该函数读出数据流中的vlan_id，并填写入skb->vlan_tci中，然后删除vlan_head，从而实现对上层的透明。注意这里的skb->vlan_tci标志仅是为了把该skb交给vlan_dev（见下面），而skb中的数据是透明的以太网包。

发现skb->vlan_tci置位，则执行vlan_do_receive(skb)，该函数由skb->vlan_tci得到该skb包所要发往的vlan_dev，并且重定向skb->dev为该vlan_dev，最后消除skb中的vlan_tci标志。

这样之后vlan_do_receive()返回，流程继续回到netif_receive_skb()中，不过此时的skb已经是一个普通的数据包了（实现了对上层的透明），且它看起来就像是由vlan_dev接收的数据包。

2.2.5关于设备重定向的总结

在发送流程中，数据包由上层下发时（如IP经过路由后下发），首先是到

了虚拟设备（如这里的VLAN，包括以后讲的Bridge），这正是这种虚拟化技术所期望的对上层透明，要注意的是，此时数据包skb就已经准备好了，其中报文的MAC地址、IP地址就是这个虚拟设备的地址，并且不再改变，这就实现了对外仿佛是实际存在的。

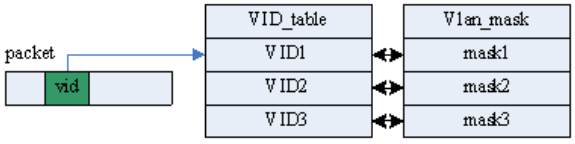
然后在dev_hard_start_xmit(skb,dev)中，对skb进行检查，发现是虚拟设备的数据，则做相应操作（如vlan_untag()，其它的都安正常发送流程走），接着就调用虚拟设备的ops->ndo_start_xmit()函数，在这个函数中，进行设备重定向。最后以real_dev重启发送流程（vlan是这样的，因为real_dev可能被多个vlan_dev使用，必须重新进行锁定等，而bridge则直接调用real_dev->ops->ndo_start_xmit()，因为它的端口从设备仅为它所用）。

在接收流程中，数据包skb首先在real_dev中被接收，并通过_netif_receive_skb()提交给上层，正是在这个函数中，检查数据包skb，若发现是发往虚拟设备的，则重定向skb->dev，再提交上层，从而实现对上层透明。

3.Linux中VLAN的应用场景

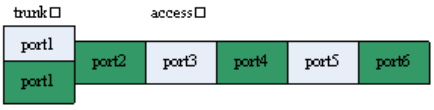
3.1一般交换机中的VLAN

Vlan最初的概念是应用与交换机中，并且由硬件来划分vlan。最传统的方法是基于port的vlan，即每个vlan虚拟网由一个vlan_id标示，并由一个vlan_mask来标示哪些port和它同处于一个vlan虚拟网，如下图所示：



其中vid和vlan_mask都存放在设备寄存器中，由硬件自动访问识别。

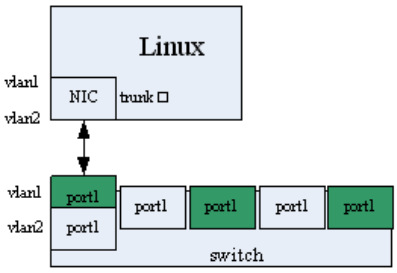
更简化一点，上图中packet中都可以不需要vid（即不需要vlan_head），硬件根据包是由哪个port收到的来索引VID_table，从而知道哪些port和它同处于一个vlan虚拟网。但对于有些应用，需要一个port同属于多个vlan的，如下图所示：



承载多个vlan的port称为trunk口，它上面收发的数据包必须含有vlan_head，以识别该包是属于哪个vlan的；只承载一个vlan的port称为access口，若硬件支持，可以不需要vlan_head就能完成vlan功能。

3.2一个应用场景分析

Linux中，Vlan设备建立在宿主设备的基础上，即该物理端口应该是trunk口，如下图所示：



由前面Linux中vlan的实现可知，发往vlan设备的数据包都被打上vlan_head，而vlan设备接收到的数据包都默认为有vlan_head，并将其去除。这是符合trunk口的定义的。

总之Linux下的VLAN模型，是一套虚拟化的架构，它为了虚拟出vlan端口，做得比较臃肿。如果把上图中下方的switch设备用Linux来驱动，该怎么模型化这个设备，还要充分利用硬件的特性，实现高效的vlan。

zmkeil
关注 - 0
粉丝 - 37
+加关注

1 0

« 上一篇：现代计算机体系架构带来的挑战
» 下一篇：Linux下的虚拟Bridge实现

分类：网络相关