tcp/ip发送接收总体框架

tcp/ip发送接收总体框架

对于接收到的报文，如果不被丢弃、不被网桥转发，会调用netif_receive_skb()提交给IP层；而对于IP层向外发送的报文，则通过调用dev_queue_xmit()提交给数据链路层。

本文就以netif_receive_skb()和dev_queue_xmit()为起始，先介绍下传输层（tcp）发送的处理过程：

sys_sendto(用户态发包调用函数)----》sock_sendmsg（sock发包函数，调用sock中对应的协议族的发包函数） ----》inet_sendmsg（以太网协议栈发包函数，被赋值于对应协议族的sendmsg字段）---》tcp_sendmsg（用户态发包函数，把用户层的数据，填充到skb中，然后加入到sock的发送队列）----》tcp_push---》__tcp_push_pending_frames（/* 尽可能的将发送队列中的skb发送出去，禁用nalge */ ）---》tcp_write_xmit（将发送队列上的SBK发送出去，返回值为0表示发送成功）---》tcp_transmit_skb（该函数会建立IP层的头，并将该头传递给网络层）---》ip_queue_xmit（此步骤已进入IP层了，首先为输入包建立IP包头, 经过本地包过滤器后,再将IP包分片输出(ip_fragment)）

再介绍下传输层（tcp）接收数据的处理流程：

三个过程：

1）网口--->触发软中断

网卡收到网络包后，触发网卡注册的硬中断处理函数，硬中断处理函数负责触发网络收包软中断。

2）软中断--->TCP

_do_softirq(软中断处理函数调用网络收报处理函数)---》net_rx_action(网络收包软中断函数)---》

简要介绍一下报文在IP层的处理过程。

先来一张图：

报文接收（图中橙色箭头所指）

netif_receive_skb()

对每一种已注册的协议类型调用deliver_skb()，从而调用到其packet_type->func()函数。

对于IP协议，会调用到ip_rcv()。

ip_rcv()

IP报头检查，调用ip_fast_csum()检查校验和。

netfilter：NF_INET_PRE_ROUTING。

调用ip_rcv_finish()。

ip_rcv_finish()

调用ip_route_input_noref()查找路由表，其结果会决定报文的去向。

调用ip_rcv_options()处理IP选项。

调用dst_input()，从而调用到rtable.dst_entry->input()。根据路由不同，主要有ip_forward（转发）、ip_local_deliver（接收）两种取值。

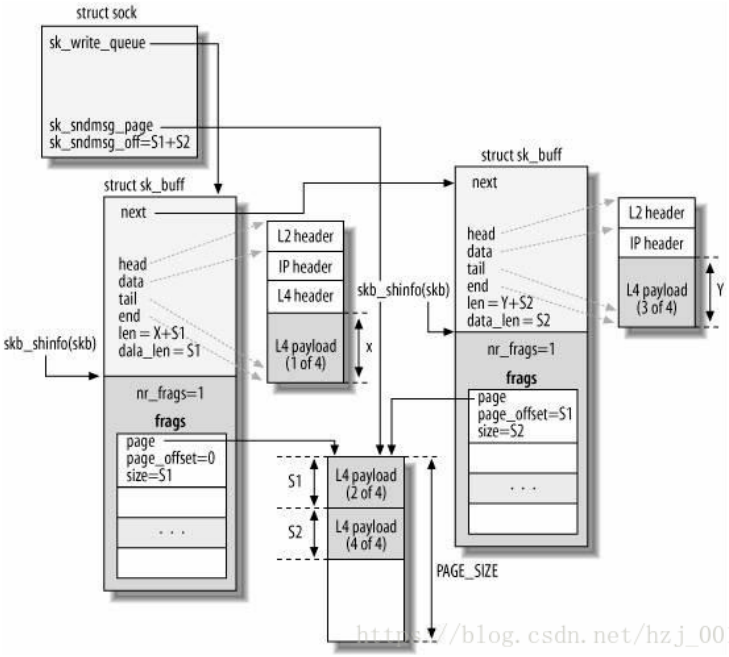
ip_local_deliver()

如果存在分片，调用ip_defrag()完成分片重组。如果分片暂未到齐则直接返回。

netfilter：NF_INET_LOCAL_IN。

调用ip_local_deliver_finish()。
ip_local_deliver_finish()
调用raw_local_deliver()试图按raw socket方式（直接收发IP报文的socket）递交给上层应用，递交成功则返回。
按ip_hdr->protocol取出相应L4协议，调用net_protocol.handler()，从而将报文提交给传输层。
主要有的L4协议handler有icmp_rcv()、udp_rcv()、tcp_v4_rcv()、等。

报文发送（图中紫色箭头所指）
传输层在发送报文时，会调用到IP层的接口。如：ip_queue_xmit()、ip_append_data()/ip_append_page()+ip_push_pending_frames()、ip_local_out()、等。
这些函数最终会调用到ip_local_out()。
在此之前，报文会被构造好。可能由传输层的代码自己构造、也可能通过调用ip_append_data()这样的辅助函数来构造。
具体构造报文的细节就不细说了，引用ULNI上的一张图，直接看结果：



struct sock是跟应用创建的socket相对应的结构，其中的sk_write_queue指向待发送的IP报文分片队列，每个分片由一个struct sk_buff来表示。
由于网络节点存在MTU，也就是最大传输单元，一次提交给数据链路层的报文不能太大（如1500字节），所以过大的IP报文需要分片后发送。
注意，只有第一个分片有L4的报头，因为对于L4协议来说，这些分片组装成的整体才是一个报文。
当然，最好的情况是没有分片，也就是L4协议总是发送尺寸小于MTU的报文。
struct sk_buff中有一组head、data、tail、end这样的指针，指向需要发送的报文buffer。
struct sk_buff之后会紧跟一个struct skb_shared_info结构。属于同一个分片的报文数据可能分散于多个碎片中，其中的第一个碎片由上述head、data等指针指向，后续碎片则由struct skb_shared_info来指示。
为什么要有多个碎片呢?
一方面，因为上层发送数据有可能就是一小片一小片的发送的，比如传送文件，每次读128字节并发送。而这些碎片可以在同一个IP报文中发送，只要总和不超过MTU。
另一方面，很多硬件支持这样的由多个碎片构成的缓冲区。如果某些硬件不支持的话，那构造sk_buff的时候就只能把每次提交的数据都拷贝到同一块连续的buffer中，这样可能就会多一次拷贝。
当然，就算硬件支持多个碎片，数据拷贝可能也无法避免。比如说当数据源来自于用户空间的时候，就必定存在用户空间到内核空间的一次拷贝（因为用户指定的地址可能存在错误，不能直接提交给网卡）。而在类似于sendfile这样的系统调用中，数据源本身就在内核空间，则可能做到真正的零拷贝。
ip_append_data()/ip_append_page()就是完成sk_buff构造的辅助函数，调用它往struct sock中塞数据，其内部会控制是否应该分配新的struct sk_buff作为分片、或者数据是否应该作为碎片放入struct skb_shared_info结构。
ip_local_out()
netfilter：NF_INET_LOCAL_OUT。
调用dst_output()，从而调用到ip_output()。
ip_output()
netfilter：NF_INET_POST_ROUTING。
调用ip_finish_output()。

ip_finish_output()
调用ip_fragment()对报文做分片后发送，或直接调用ip_finish_output2()发送。
ip_finish_output2()
调用邻居子系统neigh_output()，最终由邻居子系统调用dev_queue_xmit()发送报文。

报文转发（图中绿色箭头所指）
对于接收到的报文，如果路由子系统认为应该转发，则dst_input()会调用到ip_forward()。
ip_forward()
处理IP选项、递减ttl。
netfilter：NF_INET_FORWARD。
调用ip_forward_finish()。
ip_forward_finish()
调用ip_forward_options()处理IP选项。
调用dst_output()，从而调用到ip_output()。
在上述流程中，有几个点再额外说明一下：

netfilter：IP报文的处理流程中共有PRE_ROUTING、LOCAL_IN、FORWARD、LOCAL_OUT、POST_ROUTING这五个HOOK点。用户可以通过配置netfilter，在这几个节点上添加一些规则，实现对特定IP报文的干预。

route：路由子系统。决定IP报文下一步应该发送到哪个IP地址上（或者自己接收）。这个目的IP地址所对应的主机必定是与本机直接相连、或通过交换机相连的（也就是说，两台机器之前的通信不需要路由器转发，两台机器是“邻居”）。
举个简单的例子，路由表有如下配置：
Destination Gateway Genmask Flags Metric Ref Use Iface
10.20.150.0 * 255.255.255.0 U 0 0 0 eth0
default 10.20.150.254 0.0.0.0 UG 0 0 0 eth0
那么，目的地是10.20.150.0/24这个子网的报文，直接进行转发（目的主机和本机直接就是邻居）；而目的地是其他地址的报文，发往默认网关10.20.150.254，由它来继续转发。（注意，默认网关10.20.150.254也是本机的邻居。）

neighbour：邻居子系统。路由子系统确定了报文要发送到的IP地址，而在将报文提交给数据链路层之前，还需要知道目的主机的Mac地址。这就是邻居子系统干的事情。
简单来说，一台机器通过在子网中广播一个ARP报文，来询问目的IP地址的Mac地址是什么。比如目的IP地址是10.20.150.133，本机会广播"Who is 10.20.150.133/24"的ARP报文。像交换机这样的数据链路层设备会将ARP报文转发，从而让每一个邻居都收到。当10.20.150.133收到询问后，会向本机回复其Mac地址。
然后在此基础上，邻居子系统会实现一定的缓存策略，不会对于每个报文件都这么询问一下。