chulia20002001的空间

查看文章

linux-Tcp IP协议栈源码阅读笔记【转】

2009-11-27 15:09

http://liyanbingabc.zhmy.com/archives/2008/143590.html
linux-Tcp IP协议栈源码阅读笔记【转】
云清燕 发表于 2008-1-26 15:33:00

一.linux内核网络栈代码的准备知识

1. linux内核ipv4网络部分分层结构：

BSD socket层： 这一部分处理BSD socket相关操作，每个socket在内核中以struct socket结构体现。这一部分的文件

主要有：/net/socket.c /net/protocols.c etc

INET socket层：BSD socket是个可以用于各种网络协议的接口，而当用于tcp/ip，即建立了AF_INET形式的socket时，

还需要保留些额外的参数，于是就有了struct sock结构。文件主要

有：/net/ipv4/protocol.c /net/ipv4/af_inet.c /net/core/sock.c etc

TCP/UDP层：处理传输层的操作，传输层用struct inet_protocol和struct proto两个结构表示。文件主要

有：/net/ipv4/udp.c /net/ipv4/datagram.c /net/ipv4/tcp.c /net/ipv4/tcp_input.c /net/ipv4//tcp_output.c /net/ipv4/tcp_minisocks.c /net/ipv4/tcp_output.c /net/ipv4/tcp_timer.c

etc

IP层：处理网络层的操作，网络层用struct packet_type结构表示。文件主要有：/net/ipv4/ip_forward.c ip_fragment.c ip_input.c ip_output.c etc.

数据链路层和驱动程序：每个网络设备以struct net_device表示，通用的处理在dev.c中，驱动程序都在/driver/net目

录下。

2. 两台主机建立udp通信所走过的函数列表

```
^
|     sys_read              fs/read_write.c
|     sock_read             net/socket.c
|     sock_recvmsg          net/socket.c
|     inet_recvmsg          net/ipv4/af_inet.c
|     udp_recvmsg           net/ipv4/udp.c
|     skb_recv_datagram     net/core/datagram.c
|     --------------------------------------------
|     sock_queue_rcv_skb    include/net/sock.h
|     udp_queue_rcv_skb     net/ipv4/udp.c
|     udp_rcv               net/ipv4/udp.c
|     ip_local_deliver_finish net/ipv4/ip_input.c
|     ip_local_deliver      net/ipv4/ip_input.c
|     ip_recv               net/ipv4/ip_input.c
|     net_rx_action         net/dev.c
|     --------------------------------------------
|     netif_rx              net/dev.c
|     el3_rx                driver/net/3c309.c
|     el3_interrupt         driver/net/3c309.c

============================

|     sys_write             fs/read_write.c
|     sock_writev           net/socket.c
|     sock_sendmsg          net/socket.c
|     inet_sendmsg          net/ipv4/af_inet.c
|     udp_sendmsg           net/ipv4/udp.c
|     ip_build_xmit         net/ipv4/ip_output.c
|     output_maybe_reroute  net/ipv4/ip_output.c
|     ip_output             net/ipv4/ip_output.c
|     ip_finish_output      net/ipv4/ip_output.c
|     dev_queue_xmit        net/dev.c
|     --------------------------------------------
|     el3_start_xmit        driver/net/3c309.c
V
```

3. 网络路径图、重要数据结构sk_buffer及路由介绍

linux-net.pdf 第2.1章 第2.3章 第2.4章

4. 从连接、发送、到接收数据包的过程

linux-net.pdf 第4、5、6章详细阐述

二.linux的tcp-ip栈代码的详细分析

1.数据结构(msghdr,sk_buff,socket,sock,proto_ops,proto)

bsd套接字层,操作的对象是socket,数据存放在msghdr这样的数据结构：

创建socket需要传递family,type,protocol三个参数，创建socket其实就是创建一个socket实例，然后创建一 个文件描述符结构，并且互相建立一些关联，即建立互相连接的指针，并且初始化这些对文件的写读操作映射到socket的read，write函数上来。

同时初始化socket的操作函数（proto_ops结构),如果传入的type参数是STREAM类型，那么就初始化为 SOCKET->ops为inet_stream_ops，如果是DGRAM类型，则SOCKET-ops为inet_dgram_ops。对于 inet_stream_ops其实是一个结构体，包含了stream类型的socket操作的一些入口函数，在这些函数里主要做的是对socket进行 相关的操作，同时通过调用下面提到的sock中的相关操作完成socket到sock层的传递。比如在inet_stream_ops里有个 inet_release的操作，这个操作除了释放socket的类型空间操作外，还通过调用socket连接的sock的close操作，对于 stream类型来说，即tcp_close来关闭sock
释放sock。

创建socket同时还创建sock数据空间，初始化sock,初始化过程主要做的事情是初始化三个队列，receive_queue（接收到 的数据包sk_buff链表队列),send_queue(需要发送数据包的sk_buff链表队列),backlog_queue(主要用于tcp中三 次握手成功的那些数据包,自己猜的),根据family、type参数，初始化sock的操作，比如对于family为inet类型的，type为 stream类型的，sock->proto初始化为tcp_prot.其中包括stream类型的协议sock操作对应的入口函数。

在一端对socket进行write的过程中，首先会把要write的字符串缓冲区整理成msghdr的数据结构形式(参见linux内核 2.4版源代码分析大全),然后调用sock_sendmsg把msghdr的数据传送至inet层，对于msghdr结构中数据区中的每个数据包，创建 sk_buff结构，填充数据，挂至发送队列。一层层往下层协议传递。一下每层协议不再对数据进行拷贝。而是对sk_buff结构进行操作。

inet套接字及以下层 数据存放在sk_buff这样的数据结构里：

路由：

在linux的路由系统主要保存了三种与路由相关的数据，第一种是在物理上和本机相连接的主机地址信息表，第二种是保存了在网络访问中判断一个网络地址应该走什么路由的数据表；第三种是最新使用过的查询路由地址的缓存地址数据表。
1.neighbour结构 neighbour_table{ }是一个包含和本机所连接的所有邻元素的信息的数据结构。该结构中有个元素是neighbour结构的数组，数组的每一个元素都是一个对应于邻机的 neighbour结构，系统中由于协议的不同，会有不同的判断邻居的方式，每种都有neighbour_table{}类型的实例，这些实例是通过 neighbour_table{}中的指针next串联起来的。在neighbour结构中，包含有与该邻居相连的网络接口设备net_device的 指针，网络接口的硬件地址，邻居的硬件地址，包含有neigh_ops{}指针，这些函数指针是直接用来连接传输数据的，包含有 queue_xmit(struct * sk_buff)函数入口地址，这个函数可能会调用硬件驱动程序的发送函数。

2.FIB结构 在FIB中保存的是最重要的路由规则,通过对FIB数据的查找和换算，一定能够获得路由一个地址的方法。系统中路由一般采取的手段是：先到路由缓存中查找 表项，如果能够找到，直接对应的一项作为路由的规则；如果不能找到，那么就到FIB中根据规则换算传算出来，并且增加一项新的，在路由缓存中将项目添加进 去。
3.route结构（即路由缓存中的结构)

数据链路层：

net_device{}结构，对应于每一个网络接口设备。这个结构中包含很多可以直接获取网卡信息的函数和变量，同时包含很多对于网卡操作的函数，这些 直接指向该网卡驱动程序的许多函数入口，包括发送接收数据帧到缓冲区等。当这些完成后，比如数据接收到缓冲区后便由netif_rx(在net/core /dev.c各种设备驱动程序的上层框架程序)把它们组成sk_buff形式挂到系统接收的backlog队列然后交由上层网络协议处理。同样，对于上层 协议处理下来的那些sk_buff。便由dev_queue_xmit函数放入网络缓冲区，交给网卡驱动程序的发送程序处理。

在系统中存在一张链表dev_base将系统中所有的net_device{}结构连在一起。对应于内核初始化而言，系统启动时便为每个所有可能支持的网 络接口设备申请了一个net_device{}空间并串连起来，然后对每个接点运行检测过程，如果检测成功，则在dev_base链表中保留这个接点，否 则删除。对应于模块加载来说，则是调用register_netdev()注册net_device,在这个函数中运行检测过程，如果成功，则加到 dev_base链表。否则就返回检测不到信息。删除同理，调用
unregister_netdev。

2.启动分析

2.1 初始化进程 ：start-kernel(main.c)---->do_basic_setup(main.c)---->sock_init(/net/socket.c)---->do_initcalls(main.c)

```
void __init sock_init(void)
{
int i;

 printk(KERN_INFO "Linux NET4.0 for Linux 2.4\n");
printk(KERN_INFO "Based upon Swansea University Computer Society NET3.039\n");

 /*
* Initialize all address (protocol) families. 每一项表示的是针对一个地址族的操作集合，例如对于ipv4来说，在net/ipv4/af_inet.c文件中的函数 inet_proto_init()就调用sock_register()函数将inet_families_ops初始化到属于IPV4的 net_families数组中的一项。
*/

for (i = 0; i < NPROTO; i++)
net_families[i] = NULL;

 /*
* Initialize sock SLAB cache.初始化对于sock结构预留的内存的slab缓存。
*/
```

```
sk_init();

#ifdef SLAB_SKB
/*
* Initialize skbuff SLAB cache 初始化对于skbuff结构的slab缓存。以后对于skbuff的申请可以通过函数kmem_cache_alloc()在这个缓存中申请空间。
*/
skb_init();
#endif

 /*
* Wan router layer.
*/

#ifdef CONFIG_WAN_ROUTER
wanrouter_init();
#endif

 /*
* Initialize the protocols module. 向系统登记sock文件系统，并且将其安装到系统上来。
*/

 register_filesystem(&sock_fs_type);
sock_mnt = kern_mount(&sock_fs_type);
/* The real protocol initialization is performed when
*  do_initcalls is run.
*/

/*
* The netlink device handler may be needed early.
*/

#ifdef CONFIG_NET
rtnetlink_init();
#endif
#ifdef CONFIG_NETLINK_DEV
init_netlink();
#endif
#ifdef CONFIG_NETFILTER
netfilter_init();
#endif

#ifdef CONFIG_BLUEZ
bluez_init();
#endif

/*yfhuang ipsec*/
#ifdef CONFIG_IPSEC
pfkey_init();
#endif
/*yfhuang ipsec*/
}
```

2.2 do_initcalls() 中做了其它的初始化，其中包括

协议初始化，路由初始化，网络接口设备初始化

(例如inet_init函数以_init开头表示是系统初始化时做，函数结束后跟 module_init(inet_init),这是一个宏，在include/linux/init.c中定义，展开为 _initcall(inet_init),表示这个函数在do_initcalls被调用了)

2.3 协议初始化
此处主要列举inet协议的初始化过程。

```
static int __init inet_init(void)
{
struct sk_buff *dummy_skb;
struct inet_protocol *p;
struct inet_protosw *q;
struct list_head *r;

 printk(KERN_INFO "NET4: Linux TCP/IP 1.0 for NET4.0\n");

 if (sizeof(struct inet_skb_parm) > sizeof(dummy_skb->cb)) {
printk(KERN_CRIT "inet_proto_init: panic\n");
return -EINVAL;
}

 /*
* Tell SOCKET that we are alive... 注册socket，告诉socket inet类型的地址族已经准备好了
*/

(void) sock_register(&inet_family_ops);

 /*
* Add all the protocols. 包括arp,ip、ICMP、UPD、tcp_v4、tcp、igmp的初始化，主要初始化各种协议对应的inode和socket变量。
```

其中arp_init完成系统中路由部分neighbour表的初始化

ip_init完成ip协议的初始化。在这两个函数中，都通过定义一个packet_type结构的变量将这种数据包对应的协议发送数据、允许发送设备都做初始化。

```
*/

 printk(KERN_INFO "IP Protocols: ");
for (p = inet_protocol_base; p != NULL;) {
struct inet_protocol *tmp = (struct inet_protocol *) p->next;
inet_add_protocol(p);
printk("%s%s",p->name,tmp?", ":"\n");
p = tmp;
}

 /* Register the socket-side information for inet_create. */
for(r = &inetsw[0]; r < &inetsw[SOCK_MAX]; ++r)
INIT_LIST_HEAD(r);

 for(q = inetsw_array; q < &inetsw_array[INETSW_ARRAY_LEN]; ++q)
inet_register_protosw(q);

 /*
* Set the ARP module up
*/

 arp_init();

   /*
* Set the IP module up
*/

 ip_init();

 tcp_v4_init(&inet_family_ops);

 /* Setup TCP slab cache for open requests. */
tcp_init();

/*
* Set the ICMP layer up
*/

 icmp_init(&inet_family_ops);

 /* I wish inet_add_protocol had no constructor hook...
I had to move IPIP from net/ipv4/protocol.c :-( --ANK
*/
#ifdef CONFIG_NET_IPIP
ipip_init();
#endif
#ifdef CONFIG_NET_IPGRE
ipgre_init();
#endif

 /*
* Initialise the multicast router
*/
#if defined(CONFIG_IP_MROUTE)
ip_mr_init();
#endif

 /*
* Create all the /proc entries.
*/
#ifdef CONFIG_PROC_FS
proc_net_create ("raw", 0, raw_get_info);
proc_net_create ("netstat", 0, netstat_get_info);
proc_net_create ("snmp", 0, snmp_get_info);
proc_net_create ("sockstat", 0, afinet_get_info);
proc_net_create ("tcp", 0, tcp_get_info);
proc_net_create ("udp", 0, udp_get_info);
#endif  /* CONFIG_PROC_FS */

 ipfrag_init();

 return 0;
}
module_init(inet_init);
```

2.4 路由初始化(包括neighbour表、FIB表、和路由缓存表的初始化工作)

2.4.1 rtcache表 ip_rt_init()函数 在net/ipv4/ip_output中调用，net/ipv4/route.c中定义

2.4.2 FIB初始化 在ip_rt_init()中调用 在net/ipv4/fib_front.c中定义

2.4.3 neigbour表初始化 arp_init （） 函数中定义

2.5 网络接口设备初始化

在系统中网络接口都是由一个dev_base链表进行管理的。通过内核的启动方式也是通过这个链表进行操作的。在系 统启动之初，将所有内核能够支持的网络接口都初始化成这个链表中的一个节点，并且每个节点都需要初始化出init函数指针，用来检测网络接口设备。然后， 系统遍历整个dev_base链表，对每个节点分别调用init函数指针，如果成功，证明网络接口设备可用，那么这个节点就可以进一步初始化，如果返回失 败，那么证明该网络设备不存在或是不可用，只能将该节点删除。启动结束之后，在dev_base中剩下的都是可以用的网络接口设备。

2.5.1 do_initcalls---->net_dev_init()(net/core /dev.c)------>ethif_probe()(drivers/net/Space.c,在netdevice{}结构的init中调 用，这边ethif_probe是以太网卡针对的调用)

3.网络设备驱动程序（略)

4.网络连接

4.1 连接的建立和关闭

tcp连接建立的代码如下:

```
server=gethostbyname(SERVER_NAME);
sockfd=socket(AF_INET,SOCK_STREAM,0);
address.sin_family=AF_INET；
address.sin_port=htons(PORT_NUM);
memcpy(&address.sin_addr,server->h_addr,server->h_length);
connect(sockfd,&address,sizeof(address));
```

连接的初始化与建立期间主要发生的事情如下：

1）sys_socket调用：调用socket_creat(),创建出一个满足传入参数family、type、和 protocol的socket,调用sock_map_fd()获取一个未被使用的文件描述符，并且申请并初始化对应的file{}结构。

2）sock_creat()：创建socket结构，针对每种不同的family的socket结构的初始化，就需要调用不同 的create函数来完成。对应于inet类型的地址来说，在网络协议初始化时调用sock_register()函数中完成注册的定义如下：

```
struct net_proto_family inet_family_ops={
    PF_INET;
    inet_create
};
```

所以inet协议最后会调用inet_create函数。

3）inet_create: 初始化sock的状态设置为SS_UNCONNECTED,申请一个新的sock结构，并且初始化socket的成员ops初始化为 inet_stream_ops,而sock的成员prot初始化为tcp_prot。然后调用sock_init_data,将该socket结构的变 量sock和sock类型的变量关联起来。

4）在系统初始化完毕后便是进行connect的工作，系统调用connect将一个和socket结构关联的文件描述符和一个 sockaddr{}结构的地址对应的远程机器相关联，并且调用各个协议自己对应的connect连接函数。对应于tcp类型，则 sock->ops->connect便为inet_stream_connect。

5）inet_stream_connect: 得到sk,sk=sock->sk,锁定sk，对自动获取sk的端口号存放在sk->num中，并且用htons()函数转换存放在sk-& gt;sport中。然后调用sk->prot->connect()函数指针，对tcp协议来说就是tcp_v4_connect()函 数。然后将sock->state状态字设置为SS_CONNECTING,等待后面一系列的处理完成之后，就将状态改成 SS_CONNECTTED。

6) tcp_v4_connect()：调用函数ip_route_connect()，寻找合适的路由存放在rt中。ip_route_connect找两 次，第一次找到下一跳的ip地址，在路由缓存或fib中找到，然后第二次找到下一跳的具体邻居，到neigh_table中找到。然后申请出tcp头的空 间存放在buff中。将sk中相关地址数据做一些针对路由的变动，并且初始化一个tcp连接的序列号，调用函数tcp_connect（），初始化tcp 头，并设置tcp处理需要的定时器。一次connect（）建立的过程就结束了。

连接的关闭主要如下：

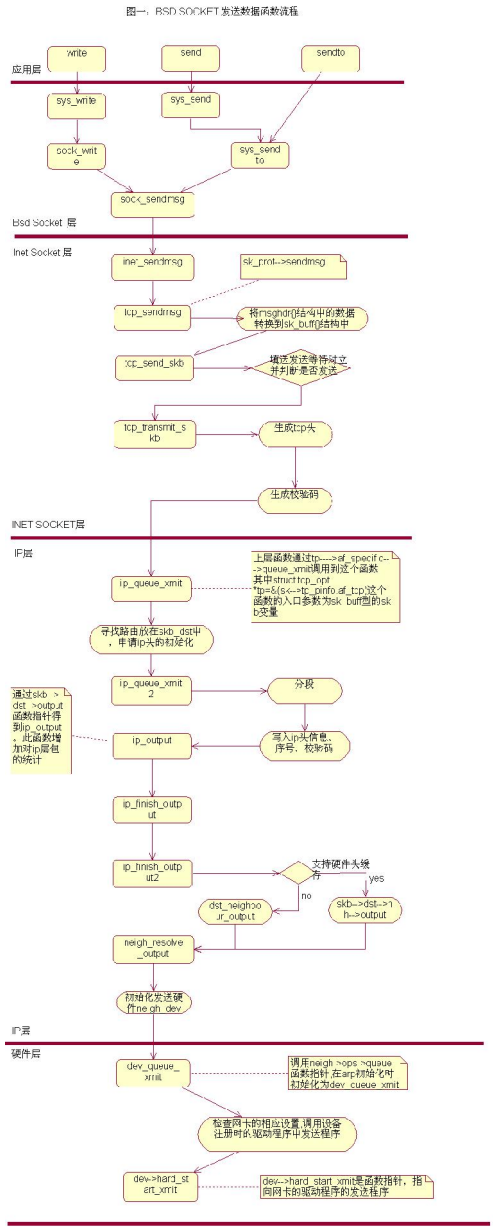
1）close: 一个socket文件描述符对应的file{}结构中，有一个file_operations{}结构的成员f_ops，它的初始化关闭函数为sock_close函数。

2）sock_close：调用函数sock_release(),参数为一个socket{}结构的指针。

3）sock_release：调用inet_release，并释放socket的指针和文件空间

4）inet_release: 调用和该socket对应协议的关闭函数inet_release,如果是tcp协议，那么调用的是tcp_close；最后释放sk。

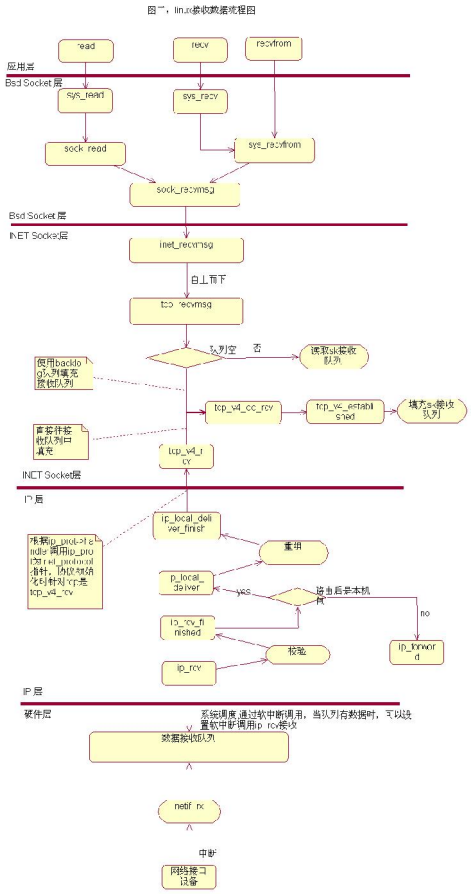
4.2 **数据发送流程图**



各层主要函数以及位置功能说明：

1)sock_write：初始化msghdr{}结构 net/socket.c
2)sock_sendmsg:net/socket.c
3)inet_sendmsg:net/ipv4/af_net.c
4)tcp_sendmsg：申请sk_buff{}结构的空间，把msghdr{}结构中的数据填入sk_buff空间。net/ipv4/tcp.c
5)tcp_send_skb:net/ipv4/tcp_output.c
6)tcp_transmit_skb:net/ipv4/tcp_output.c
7)ip_queue_xmit:net/ipv4/ip_output.c
8)ip_queue_xmit2:net/ipv4/ip_output.c
9)ip_output:net/ipv4/ip_output.c
10)ip_finish_output:net/ipv4/ip_output.c
11)ip_finish_output2:net/ipv4/ip_output.c
12)neigh_resolve_output:net/core/neighbour.c
13)dev_queue_xmit:net/core/dev.c

4.3 数据接收流程图



各层主要函数以及位置功能说明：

1)sock_read:初始化msghdr{}的结构类型变量msg，并且将需要接收的数据存放的地址传给msg.msg_iov->iov_base. net/socket.c

2)sock_recvmsg: 调用函数指针sock->ops->recvmsg()完成在INET Socket层的数据接收过程.其中sock->ops被初始化为inet_stream_ops,其成员recvmsg对应的函数实现为 inet_recvmsg()函数. net/socket.c

3)sys_recv()/sys_recvfrom():分别对应着面向连接和面向无连接的协议两种情况. net/socket.c

4)inet_recvmsg:调用sk->prot->recvmsg函数完成数据接收,这个函数对于tcp协议便是tcp_recvmsg net/ipv4/af_net.c

5)tcp_recvmsg:从网络协议栈接收数据的动作,自上而下的触发动作一直到这个函数为止,出现了一次等待的过程.函 数tcp_recvmsg可能会被动地等待在sk的接收数据队列上,也就是说,系统中肯定有其他地方会去修改这个队列使得tcp_recvmsg可以进行 下去.入口参数sk是这个网络连接对应的sock{}指针,msg用于存放接收到的数据.接收数据的时候会去遍历接收队列中的数据,找到序列号合适的.

但读取队列为空时tcp_recvmsg就会调用tcp_v4_do_rcv使用backlog队列填充接收队列.

6)tcp_v4_rcv:tcp_v4_rcv被ip_local_deliver函数调用,是从IP层协议向INET Socket层提交的"数据到"请求,入口参数skb存放接收到的数据,len是接收的数据的长度,这个函数首先移动skb->data指针,让它 指向tcp头,然后更新tcp层的一些数据统计,然后进行tcp的一些值的校验.再从INET Socket层中已经建立的sock{}结构变量中查找正在等待当前到达数据的哪一项.可能这个sock{}结构已经建立,或者还处于监听端口、等待数据 连接的状态。返回的sock结构指针存放在sk中。然后根据其他进程对sk的操作情况,将skb发送到合适的位置.调用如下:

TCP包接收器(tcp_v4_rcv)将TCP包投递到目的套接字进行接收处理. 当套接字正被用户锁定,TCP包将暂时排入该套接字的后备队列(sk_add_backlog).这时如果某一用户线程企图锁定该套接字 (lock_sock),该线程被排入套接字的后备处理等待队列(sk->lock.wq).当用户释放上锁的套接字时 (release_sock,在tcp_recvmsg中调用),后备队列中的TCP包被立即注入TCP包处理器(tcp_v4_do_rcv)进行处 理,然后唤醒等待队列中最先的一个用户来获得其锁定权. 如果套接字未被上锁,当用户正在读取该套接字时, TCP包将被排入套接字的预备队列(tcp_prequeue),将其传递到该用户线程上下文中进行处理.如果添加到sk->prequeue不成 功,便可以添加到 sk->receive_queue队列中(用户线程可以登记到预备队列,当预备队列中出现第一个包时就唤醒等待线程.) /net/tcp_ipv4.c

7)ip_rcv、ip_rcv_finish:从以太网接收数据，放到skb里，作ip层的一些数据及选项检查，调用 ip_route_input()做路由处理,判断是进行ip转发还是将数据传递到高一层的协议.调用skb->dst->input函数指 针,这个指针的实现可能有多种情况,如果路由得到的结果说明这个数据包应该转发到其他主机,这里的input便是ip_forward;如果数据包是给本 机的,那么input指针初始化为ip_local_deliver函数./net/ipv4/ip_input.c

8)ip_local_deliver、ip_local_deliver_finish:入口参数skb存放需要传送到上层 协议的数据,从ip头中获取是否已经分拆的信息,如果已经分拆,则调用函数ip_defrag将数据包重组。然后通过调用 ip_prot->handler指针调用tcp_v4_rcv(tcp)。ip_prot是inet_protocol结构指针,是用来ip层登 记协议的，比如由udp,tcp,icmp等协议。 /net/ipv4/ip_input.c

（完）

相关文章：

- 源码公开的TCP/IP协议栈在远程监...
- linux-Tcp IP协议栈源码阅读笔记...
- linux-Tcp IP协议栈源码阅读笔记...

最近读者：

登录后，您就出现在这里。

jhliang2008

周星驰喜剧

火狐Foxy

COCAGU11

瓜子脸和小酒窝

gbs111998877

网友评论：

发表评论：

内　容：

发表评论