**Ipv6初始化和处理流程分析**

**一.Ipv6的初始化**

**1.网络子系统概述**

Linux 内核中，与网络相关的代码是一个相对独立的子系统，称为网络子系统。

网络子系统是一个层次化的结构，可分为以下几个层次：

**1）*Socket 层***

Linux 在发展过程中，采用 BSD socket APIs 作为自己的网络相关的 API 接口。同时，Linux 的目标又要能支持各种不同的协议族，而且这些协议族都可以使用 BSD socket APIs 作为应用层的编程接口。因此，在 socket APIs 与协议族层之间抽象出一个 socket 层，用于将 user space 的 socket API 调用，转给具体的协议族做处理。

**2）*协议族层（INET协议族、INET6协议族等）***

Linux 网络子系统功能上相当完备，它不仅支持 INET 协议族（也就是通常所说的 TCP/IP stack），而且还支持其它很多种协议族，如 DECnet, ROSE, NETBEUI 等。INET6 就是一种新增加的协议族。

对于 INET、INET6 协议族来说， 又进一步划分为传输层和网络层。

**3）*设备驱动层***

设备驱动层则主要将协议族层与物理的网络设备隔离开。它不在本文的讨论范围之内。

下图是 Linux 网络系统层次结构图。

A description...

**2.网络子系统的初始化**

**1） *Socket 层的初始化*：**

Init()->do\_basic\_setup()->sock\_init()

Sock\_init()：对 sock 和 skbuff 结构进行 SLAB 内存的初始化工作

**2）*各种网络协议族的初始化*：**

**Do\_initcalls()：**

对于编译到内核中的功能模块（而不是以模块的形式动态加载），它的初始化函数会在这个地方被调用到。

内核映象中专门有一个初始化段，所有编译到内核中的功能模块的初始化函数都会加入到这个段中；而 do\_initcalls() 就是依次执行初始化段中的这些函数。

INET 协议族通常是被编译进内核的；它的模块初始化函数是 net/ipv4/af\_inet.c 中的 inet\_init()

而INET6 是作为一个模块编译的。它的模块初始化函数是 net/ipv6/af\_inet6.c 中的 inet6\_init()

**3.协议族**

Linux 网络子系统可以支持不同的协议族，Linux 所支持的协议族定义在 include/linux/socket.h

**1）*协议族数据结构***

协议族数据结构是 struct net\_proto\_family。

struct net\_proto\_family {  
 int family;  
 int (\*create)(struct socket \*sock, int protocol);  
 short authentication;  
 short encryption;  
 short encrypt\_net;  
 struct module \*owner;  
 };

这个结构中，最重要的是create 函数，一个新的协议族，必须提供此函数的实现。这是因为：

不同的网络协议族，从user space的使用方法来说，都是一样的，都是先调用socket()来创建一个 socket fd，然后通过这个fd发送/接收数据。

在user space 通过 socket() 系统调用进入内核后，根据第一个参数协议族类型，来调用相应协议族create()函数。对INET6来说，这个函数inet6\_create()。

因此，要实现一个新的协议族，首先需要提供一个create() 的实现。关于 create() 里面具体做了什么，后面再叙述。

Linux 系统通过这种方式，可以很方便的支持新的网络协议族，而不用修改已有的代码。这很好的符合了 “开-闭原则”，对扩展开放，对修改封闭。

**2）*协议族注册***

Linux 维护一个struct net\_proto\_family 的数组net\_families[]

如果要支持一个新的网络协议族，那么需要定义自己的struct net\_proto\_family，并且通过调用 sock\_register 将它注册到 net\_families[] 中。

**4.sock层**

socket 层又叫 “socket access protocol layer”。它处于 BSD socket APIs 与底层具体的协议族之间。这是一个抽象层，它起着承上启下的作用。在这一层的数据结构也有着这种特点

**1）*数据结构***

在 user space，通过 socket() 创建的socket fd，在内核中对应的就是一个 struct socket。

struct socket {  
 socket\_state state;  
 unsigned long flags;  
 struct proto\_ops \*ops;  
 struct fasync\_struct \*fasync\_list;  
 struct file \*file;  
 struct sock \*sk;  
 wait\_queue\_head\_t wait;  
 short type;  
 };

它定义于 include/linux/net.h 中。

Struct socket 的 ops 域指向一个 struct proto\_ops 结构，struct proto\_ops定义于 include/linux/net.h 中，它是 socket 层提供给上层的接口，这个结构中，都是 BSD socket API 的具体实现的函数指针。

一个 socket API 通过系统调用进入内核后，首先由 socket 层处理。Socket 层找到对应的 struct socket，通过它找到 struct proto\_ops，然后由它所指向的函数进行进一步处理。

以 sendmsg() 这个函数为例，从 user space 通过系统调用进入 kernel 后，由 sys\_sendmsg()、sock\_sendmsg() 依次处理，然后交给 struct proto\_ops 的 sendmsg() 处理。

**2）*sock层和传输层的关联***

INET 和 INET6 这两种协议族，可以支持多种传输层协议，包括 TCP、UDP、RAW，在 2.6 内核中，又增加了一种新的传输层协议：SCTP。

从内核角度看，要实现 INET6 协议族的某种传输层协议，则必须既提供 socket 层的 struct proto\_ops 的实现，也提供 struct proto 的实现。除此之外，还需要提供一种手段，把这两个结构关联起来，也就是把 socket 层和传输层关联起来。

Linux 提供了一个 struct inet\_protosw 的结构，用于 socket 层与传输层的关联。

struct inet\_protosw {  
 struct list\_head list;  
 /\* These two fields form the lookup key. \*/  
 unsigned short type; /\* This is the 2nd argument to socket(2). \*/  
 int protocol; /\* This is the L4 protocol number. \*/   
 struct proto \*prot;  
 struct proto\_ops \*ops;  
 int capability; /\* Which (if any) capability do  
 \* we need to use this socket  
 \* interface?\*/  
 char no\_check; /\* checksum on rcv/xmit/none? \*/  
 unsigned char flags; /\* See INET\_PROTOSW\_\* below. \*/  
 };

这个结构定义于 include/net/protocol.h 中，从它的命名上可以看到它属于 INET 和 INET6 协议族，但是没有查到资料为什么叫做 protosw。

这个结构中，ops 指向 socket 层的 struct proto\_ops，prot 指向传输层的 struct proto。

因此，对 INET6 这种要支持多种传输层协议的协议族，从内核的角度来说，只需要为每一种传输层协议定义相应的 struct proto\_ops、struct proto，然后再定义 struct inet\_protosw，并将三者关联起来即可。

以 INET6 所支持的 TCP 为例：

static struct proto\_ops inet6\_sockraw\_ops = {  
 .family = PF\_INET6,  
 .owner = THIS\_MODULE,  
 .release = inet6\_release,  
 .bind = inet6\_bind,  
 .connect = inet\_dgram\_connect, /\* ok \*/  
 .socketpair = sock\_no\_socketpair, /\* a do nothing \*/  
 .accept = sock\_no\_accept, /\* a do nothing \*/  
 .getname = inet6\_getname,  
 .poll = datagram\_poll, /\* ok \*/  
 .ioctl = inet6\_ioctl, /\* must change \*/  
 .listen = sock\_no\_listen, /\* ok \*/  
 .shutdown = inet\_shutdown, /\* ok \*/  
 .setsockopt = sock\_common\_setsockopt, /\* ok \*/  
 .getsockopt = sock\_common\_getsockopt, /\* ok \*/  
 .sendmsg = inet\_sendmsg, /\* ok \*/  
 .recvmsg = sock\_common\_recvmsg, /\* ok \*/  
 .mmap = sock\_no\_mmap,  
 .sendpage = sock\_no\_sendpage,  
 };

struct proto tcpv6\_prot = {  
 .name = "TCPv6",  
 .owner = THIS\_MODULE,  
 .close = tcp\_close,  
 .connect = tcp\_v6\_connect,  
 .disconnect = tcp\_disconnect,  
 .accept = inet\_csk\_accept,  
 .ioctl = tcp\_ioctl,  
 .init = tcp\_v6\_init\_sock,  
 .destroy = tcp\_v6\_destroy\_sock,  
 .shutdown = tcp\_shutdown,  
 .setsockopt = tcp\_setsockopt,  
 .getsockopt = tcp\_getsockopt,  
 .sendmsg = tcp\_sendmsg,  
 .recvmsg = tcp\_recvmsg,  
 .backlog\_rcv = tcp\_v6\_do\_rcv,  
 .hash = tcp\_v6\_hash,  
 .unhash = tcp\_unhash,  
 .get\_port = tcp\_v6\_get\_port,  
 .enter\_memory\_pressure = tcp\_enter\_memory\_pressure,  
 .sockets\_allocated = &tcp\_sockets\_allocated,  
 .memory\_allocated = &tcp\_memory\_allocated,  
 .memory\_pressure = &tcp\_memory\_pressure,  
 .orphan\_count = &tcp\_orphan\_count,  
 .sysctl\_mem = sysctl\_tcp\_mem,  
 .sysctl\_wmem = sysctl\_tcp\_wmem,  
 .sysctl\_rmem = sysctl\_tcp\_rmem,  
 .max\_header = MAX\_TCP\_HEADER,  
 .obj\_size = sizeof(struct tcp6\_sock),  
 .twsk\_obj\_size = sizeof(struct tcp6\_timewait\_sock),  
 .rsk\_prot = &tcp6\_request\_sock\_ops,  
 };

static struct inet\_protosw tcpv6\_protosw = {  
 .type = SOCK\_STREAM,  
 .protocol = IPPROTO\_TCP,  
 .prot = &tcpv6\_prot,  
 .ops = &inet6\_stream\_ops,  
 .capability = -1,  
 .no\_check = 0,  
 .flags = INET\_PROTOSW\_PERMANENT,  
 };

Linux 为 INET6 协议族定义一个 struct inet\_protosw 的链表数组inetsw6[] 。

要支持某种传输层协议，首先实现相应的 struct proto\_ops、struct proto，然后实现struct inet\_protosw，将两者关联，最后，通过 inet6\_register\_protosw() ，将此 struct inet\_protosw 注册到 inet6\_sw[] 中。

注册的时候，根据 struct inet\_protosw 的 type ，将它放到 inet6\_sw[type] 所在的链表中，相同的 type, 不同的 protocol，会在同一个链表上。

**3）*数据结构之间的联系***

从 user space 角度看，要使用 INET6 协议族的某种传输层协议，首先需要通过 socket() 调用创建一个相应的 socket fd，然后再通过这个 socket fd，接收和发送数据。

socket() 的原型是：

int socket(int domain, int type, int protocol);

domain 指定了协议族.

type 表明在网络中通信所遵循的模式。主要的值有：SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW等。

SOCK\_STREAM 是面向流的通信方式，而 SOCK\_DGRAM 是面向报文的通信方式。不同的通信方式，在接收数据和发送数据时，具有不同的处理方式。

Protocol 则指明具体的传输层协议。不同的传输层协议，可能具有相同的 type，例如 TCP 和 SCTP 都是 SOCK\_STREAM 类型。

以 socket(PF\_INET6, SOCK\_STREAM, 0) 为例，在进入内核空间后，

根据 domain，找到inet6\_family\_ops。

创建 struct socket

调用inet6\_family\_opsde create()，也就是inet6\_create()

inet6\_create() 根据 type 和 protocol 在 inet6\_sw[] 中找到对应的 struct inet\_protosw，也就是tcpv6\_protosw

创建 struct sock，将 struct socket 和 struct sock 关联起来

将 struct socket 和 tcpv6\_protosw 的 ops ，也就是inet6\_stream\_ops 关联起来

将 struct sock 和 tcpv6\_protosw 的 prot，也就是tcpv6\_prot 关联起来。

这样，socket 层和传输层的数据结构之间的关系建立起来了，此后，应用层通过 socket fd 接收或者发送数据的时候，就会先经过 socket 层 inet6\_stream\_ops 处理，然后经过传输层的 tcpv6\_prot 处理。

**二.网卡接收数据**

这部分是说明数据报文在在链路层的处理，以及如何将报文送交给对应的网络层协议来处理。这些功能基本都是在驱动中实现的。

**1. 网络中接收数据报文的两个终端：硬中断和软中断**

(1).硬中断的中断处理函数是在驱动中注册，一般在device open()函数或者 device init()函数中注册，使用request\_irq()来注册硬中断处理函数。当网卡接收到数据的时候，就会调用这个终端处理函数来处理。比如8139too.c函数就用 retval = request\_irq (dev->irq, rtl8139\_interrupt, SA\_SHIRQ, dev->name, dev)来注册硬中断处理函数。

(2).软中断是通过NET\_RX\_SOFTIRQ信号来触发的，处理函数是net\_rx\_action。注册函数是open\_softirq(NET\_RX\_SOFTIRQ, net\_rx\_action, NULL)。触发这个中断信号（raise irq）一般是在硬中断处理流程中，当硬中断处理基本结束的时候，通过调用\_\_raise\_softirq\_irqoff(NET\_RX\_SOFTIRQ)来触发这个中断。

**2.接收软中断**

接收软中断(net\_rx\_action)主要还是通过调用驱动中的poll的方法进行接收。在poll方法中，会提取接收包，根据它所在的设备和协议类型传递给各自的包处理器。以rtl8139\_poll为例，它会调用rtl8139\_rx()来把尽可能多的数据在一次中断处理中处理完，而不是一个软中断只处理一个数据包，这样可以提高效率。每个数据包都会通过netif\_receive\_skb()函数，根据报文的协议类型，调用上层的包处理器。如果网卡本身驱动没有poll函数，将是调用bakclog\_dev的process\_backlog函数。

**3.包处理器注册**

包处理器用dev\_add\_pack()注册,如果注册的设备号是零则表明它接收所有设备的包, 如果注册的包类型是(ETH\_P\_ALL),则表示它接收所有类型的包。 netif\_receive\_skb()函数会根据接受数据的协议类型，在ptype\_all和ptype\_base列表中去查找对应的处理协议，再将数据包传递给对应的协议处理函数。

对于ipv4,就是在af\_inet.c中的inet\_init()函数中，初始化了ip\_packet\_type.func = ip\_rcv，因此，ip\_rcv()将接收ipv4的报文。在inet\_init()中调用dev\_add\_pack(&ip\_packet\_type)去在ptype\_all和ptype\_base中注册协议处理函数。ipv4的IP头类型值是ETH\_P\_IP：0x0800。

对于ipv6,则在af\_inet6.c中的inet6\_init()函数中完成初始化， inet6\_init()调用ipv6\_packet\_init()来注册协议处理报文。Ipv6的IP头类型值是 ETH\_P\_IPV6 0x86DD。其注册的接收处理程序是ipv6\_rcv()。

**三.网络层的处理**

这部分是说明数据报文在网络层的处理。上面一部分已经说明了在链路层的处理。在链路层的处理，基本都是在驱动中已经实现了的。接着链路层的处理，对于ipv6协议，处理过程在ipv6\_recv()中。

ipv6\_rcv()中，会做一些必要的检查和更新MIB的一些信息，接着处理hopbyhop报头。然后进入NF\_HOOK(PF\_INET, NF\_IP\_PRE\_ROUTING, skb, dev, NULL, ip6\_rcv\_finish); 对于NF\_HOOK的作用解释。 如果没有配置netfilter，可以简单认为NF\_HOOK就等于直接调用ip6\_rcv\_finish (skb)。

Ipv6\_rcv()会处理hopbyhop报头，在ipv6\_parse\_hopopts()函数中处理。注意ip6\_parse\_tlv()的处理过程，它本身只处理PAD0和PAD1的type，就是rfc2460里面最早定义的两个选项，其它选项都是通过tlvprochopopt\_lst中定义的回调函数来处理的。这样就能够根据将来协议的发展，灵活的添加新的hopbyhop类型，而不需要修改这个函数本身。

对于除hopbyhop以外的扩展报头的处理，是通过路由表来进行的。在ip6\_rcv\_finish()里面，会调用ip6\_route\_input(skb)，这个函数返回的是路由表中对应的fib6\_node，这个节点的input函数，就会根据不同的目的地调用不同的函数来处理。具体说来：

（1）If the destination address matches FE80::<EUI64>

skb->dst->input=ip6\_input

skb->dst->output=ip6\_output

（2）Else if the destination address’s first 10 bits matches FE80::

skb->dst->input=ip6\_forward

skb->dst->output=ip6\_output

（3）Else if the destination address’s first 8 bits matches FF00::

skb->dst->input=ip6\_mc\_input

skb->dst->output=ip6\_output

（4）Else ( no match)

skb->dst->input= ip6\_pkt\_discard

skb->dst->output=ip6\_pkt\_discard

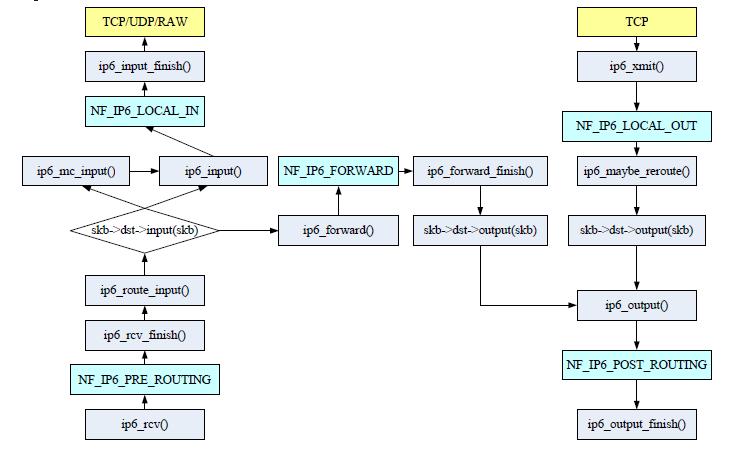
对于ip6\_route\_input(skb)函数的分析，我们后面讲路由查找的时候再叙述，这里先跳过去。

说明一下到本机的路由表项的初始化过程。到本机的路由表的初始化是在给网卡分配ipv6地址的时候初始化的，代码在addrconf.c中。比如当用户在给网卡手动赋ipv6地址的时候，会通过netlink接口，传递到内核以后，就由rtnetlink\_rcv\_msg()来处理。Rtnetlink\_recv\_msg()会根据family的值，在rtnetlink\_links[family]表中进行查找，找到对应协议簇的处理表。对于ipv6而言是PF\_INET6协议簇，调用的是inet6\_rtnetlink\_table[]。在inet6\_rtnetlink\_table[]表中，对应添加网卡ipv6地址的处理函数是inet6\_rtm\_newaddr()函数，因此整个处理过程是inet6\_rtm\_newaddr()ipv6\_add\_addr()  addrconf\_dst\_alloc()  rt->u.dst.input = ip6\_input()，从而转入ip6\_input()函数的处理。

转发路由表项的初始化和到本机的路由表的初始化过程类似。从netlink再到ip6\_route\_add()，添加ip6\_forward()的处理函数。

接上面第2点，在ip6\_route\_input(skb)函数调用中，返回的是路由表中对应的fib6\_node结构，它会调用 skb->dst->input()函数。如果数据报文是到本机，这个函数就是ip6\_input()函数。扩展报头的处理就在ip6\_input()函数中。然后调用ip6\_input\_finish()  ipprot->handler(&skb) ， 然后调用在inet6\_protos[]里面注册了的ipv6各个扩展报头的处理函数。

在ipv6协议簇中的函数调用流程如下图所示。



在tcpv6\_init()函数中，通过inet6\_add\_protocol()向inet6\_protos[]注册了处理函数tcp\_v6\_rcv()，这样，协议就会交给tcp\_v6\_rcv（）处理了。这样就交给了传输层的协议栈来处理了。ICMPv6和UPDv6协议的处理类似。

这里注意各个处理函数的返回值，像icmpv6\_rcv（）返回0，表示这个数据报不再处理了，已经处理完了。而ipv6\_destopt\_rcv()则返回-1/1，-1表示出错了，就不再处理；1表示当前的报头已经处理完了，要接着处理这个数据报的下一个报头。这样，就把报文传送到了传输层了。对于传输层，我们选择一个简单的updv6协议，它注册的处理函数是udpv6\_rcv(),这部分会在传输层的处理中论述。

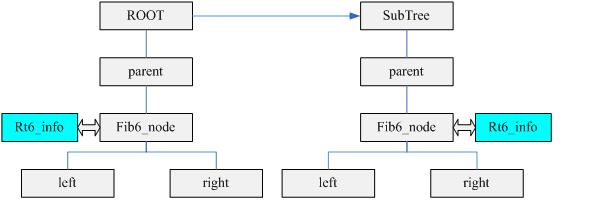
**四.路由模块的处理**

路由节点结构是fib6\_node的结构，通过这个结构来组织成一棵路由树。这个结构主要是用来组织路由结构树的，具体的路由信息是存放在fib6\_node->leaf结构中，这是一个rt6\_info的结构体。每个fib6\_node伴随着一个rt6\_info。查找路由的时候，遍历整个路由树，根据每个fib6\_node节点的rt6\_info信息，判断是否是自己需要的节点。如果是，则返回，然后根据这个节点的rt6\_info信息进行路由。

路由表的组织结构如下图所示。

这里多说两句，定义CONFIG\_IPV6\_SUBTREES情况，fib6\_lookup\_1会递归调用，但最多只能递归一次(因为subtree里不会再有subtree)。递归的那次fib6\_lookup\_1调用只对src进行了匹配，因为args[1]里的addr是src。

下图中的蓝色部分，表示每个fib6\_node都伴随着一个rt6\_info结构用来携带具体的路由信息。



ipv6的路由表是是一个radix树，根对应默认路由，结点的层次和路由prefix\_len对应。在fib6\_lookup\_1()中下面的循环把fn设为叶子结点，然后从他开始匹配，如果不符就fn = fn->parent。这样就做到了最长匹配原则。

for (;;)

{  
 struct fib6\_node \*next;  
 dir = addr\_bit\_set(args->addr, fn->fn\_bit);  
 next = dir ? fn->right : fn->left;  
 if (next)

{  
 fn = next;

continue;

}  
 break;  
 };

关于radix树的介绍，可以google，这里简单介绍一下，参考了blog：<http://wurong81.spaces.live.com/blog/cns!5EB4A630986C6ECC!393.entry?sa=419936170>　。Radix tree 是一种搜索树，采用二进制数据进行查找，但对于路由表，采用的是二叉树的方式，只有一个Left和right两个子节点。（好像fn\_bit表示的是prefix\_len，就是路由前缀的长度，不确定？）

**五.数据包接收流程分析**

接收的流程为：ipv6\_rcv--->ipv6\_rcv\_finish---->dst\_input-àip6\_input-àip6\_input\_finish

或者ipv6\_rcv--->ipv6\_rcv\_finish---->ip6\_route\_input或者ipv6\_rcv--->ipv6\_rcv\_finish---->dst\_input-àip6\_forward-àip6\_forward\_finish

static struct packet\_type ipv6\_packet\_type\_\_read\_mostly = {

.type= cpu\_to\_be16(ETH\_P\_IPV6),

.func= ipv6\_rcv,

.gso\_send\_check= ipv6\_gso\_send\_check,

.gso\_segment= ipv6\_gso\_segment,

.gro\_receive= ipv6\_gro\_receive,

.gro\_complete= ipv6\_gro\_complete,

};

//执行一些检查，判断数据包是否转发、有效性、正确性

int ipv6\_rcv(struct sk\_buff \*skb, struct net\_device \*dev, struct packet\_type \*pt,struct net\_device \*orig\_dev)

{

struct ipv6hdr \*hdr;

u32 pkt\_len;

struct inet6\_dev \*idev;

//获取数据包网卡

structnet \*net = dev\_net(skb->dev);

//丢弃发送给其他主机的数据包

if (skb->pkt\_type == PACKET\_OTHERHOST) {

kfree\_skb(skb);

return0;

}

rcu\_read\_lock();

idev= \_\_in6\_dev\_get(skb->dev);

IP6\_UPD\_PO\_STATS\_BH(net,idev, IPSTATS\_MIB\_IN, skb->len);

if ((skb = skb\_share\_check(skb, GFP\_ATOMIC)) == NULL ||

!idev ||unlikely(idev->cnf.disable\_ipv6)) {

IP6\_INC\_STATS\_BH(net,idev, IPSTATS\_MIB\_INDISCARDS);

gotodrop;

}

memset(IP6CB(skb),0, sizeof(struct inet6\_skb\_parm));

/\*

\* Store incoming device index. When the packetwill

\* be queued, we cannot refer to skb->devanymore.

\*

\* BTW, when we send a packet for our own localaddress on a

\* non-loopback interface (e.g. ethX), it isbeing delivered

\* via the loopback interface (lo) here;skb->dev = loopback\_dev.

\* It, however, should be considered as if itis being

\* arrived via the sending interface (ethX),because of the

\* nature of scoping architecture. --yoshfuji

\*/

//保存入口设备索引

IP6CB(skb)->iif = skb\_dst(skb) ? ip6\_dst\_idev(skb\_dst(skb))->dev->ifindex : dev->ifindex;

//检查数据包长度是否为IP报头的长度

if (unlikely(!pskb\_may\_pull(skb, sizeof(\*hdr))))

gotoerr;

//获取IPv6报头位置

hdr = ipv6\_hdr(skb);

//检查版本是否为IPv6

if(hdr->version != 6)

gotoerr;

/\*

\* RFC4291 2.5.3

\* A packet received on an interface with adestination address

\* of loopback must be dropped.

\*/

//丢弃环路数据包

if (!(dev->flags & IFF\_LOOPBACK) &&

ipv6\_addr\_loopback(&hdr->daddr))

gotoerr;

skb->transport\_header = skb->network\_header + sizeof(\*hdr);

IP6CB(skb)->nhoff = offsetof(struct ipv6hdr, nexthdr);

pkt\_len = ntohs(hdr->payload\_len);

//处理Jumbo负载选项

/\*pkt\_len may be zero if Jumbo payload option is present \*/

if (pkt\_len || hdr->nexthdr != NEXTHDR\_HOP) {

if (pkt\_len + sizeof(struct ipv6hdr) > skb->len) {

IP6\_INC\_STATS\_BH(net,

idev, IPSTATS\_MIB\_INTRUNCATEDPKTS);

gotodrop;

}

if (pskb\_trim\_rcsum(skb, pkt\_len + sizeof(struct ipv6hdr))) {

IP6\_INC\_STATS\_BH(net,idev,IPSTATS\_MIB\_INHDRERRORS);

gotodrop;

}

hdr = ipv6\_hdr(skb);

}

if (hdr->nexthdr == NEXTHDR\_HOP) {

if (ipv6\_parse\_hopopts(skb) < 0) {

IP6\_INC\_STATS\_BH(net,idev,IPSTATS\_MIB\_INHDRERRORS);

rcu\_read\_unlock();

return 0;

}

}

rcu\_read\_unlock();

/\*Must drop socket now because of tproxy. \*/

skb\_orphan(skb);

//由过滤器调用IP6\_rcv\_finish函数进一步处理数据包

return NF\_HOOK(PF\_INET6, NF\_INET\_PRE\_ROUTING, skb, dev, NULL,

ip6\_rcv\_finish);

err:

IP6\_INC\_STATS\_BH(net,idev, IPSTATS\_MIB\_INHDRERRORS);

drop:

rcu\_read\_unlock();

kfree\_skb(skb);

return 0;

}

//如果路由表项信息已经缓存在套接字缓冲区的dst字段，则直接用skb->dst->input指向的函数；否则，调用ip6\_rout\_input函数查找路由表，返回skb->dst->input的具体内容

inl ine int ip6\_rcv\_finish( struct sk\_buff \*skb)

{

if (skb\_dst(skb) == NULL)

ip6\_route\_input(skb);

return dst\_input(skb);

}

//获取目的地址描述符

static inline struct dst\_entry \*skb\_dst(const struct sk\_buff \*skb)

{

return (struct dst\_entry\*)skb->\_skb\_dst;

}

/\* Input packetfrom network to transport.

将数据包从网络层送到传输层

\*/

static inline int dst\_input(struct sk\_buff \*skb)

{

return skb\_dst(skb)->input(skb);

}

//交给路由模块

void ip6\_route\_input(struct sk\_buff \*skb)

{

struct ipv6hdr \*iph = ipv6\_hdr(skb);

struct net \*net = dev\_net(skb->dev);

int flags = RT6\_LOOKUP\_F\_HAS\_SADDR;

struct flowi fl = {

.iif = skb->dev->ifindex,

.nl\_u = {

.ip6\_u = {

.daddr =iph->daddr,

.saddr =iph->saddr,

.flowlabel= (\* (\_\_be32 \*) iph)&IPV6\_FLOWINFO\_MASK,

},

},

.mark = skb->mark,

.proto = iph->nexthdr,

};

if (rt6\_need\_strict(&iph->daddr)&& skb->dev->type != ARPHRD\_PIMREG)

flags |= RT6\_LOOKUP\_F\_IFACE;

skb\_dst\_set(skb, fib6\_rule\_lookup(net,&fl, flags, ip6\_pol\_route\_input));

}

int ip6\_input(struct sk\_buff \*skb)

{

returnNF\_HOOK(PF\_INET6, NF\_INET\_LOCAL\_IN, skb, skb->dev, NULL,

ip6\_input\_finish);

}

static int ip6\_input\_finish(struct sk\_buff \*skb)

{

struct inet6\_protocol \*ipprot;

unsigned int nhoff;

int nexthdr, raw;

u8 hash;

struct inet6\_dev \*idev;

struct net \*net = dev\_net(skb\_dst(skb)->dev);

/\*

\* Parseextension headers解析扩展头

\*/

rcu\_read\_lock();

resubmit:

//指向目的设备

idev= ip6\_dst\_idev(skb\_dst(skb));

//检查数据包长度是否是传输层头部

if(!pskb\_pull(skb, skb\_transport\_offset(skb)))

gotodiscard;

// #define IP6CB(skb) ((struct inet6\_skb\_parm\*)((skb)->cb))

nhoff= IP6CB(skb)->nhoff;

//获取下一个扩展头部

nexthdr= skb\_network\_header(skb)[nhoff];

***/\****

***int raw6\_local\_deliver(struct sk\_buff \*skb,int nexthdr)***

***{***

***structsock \*raw\_sk;***

***//判断可否通过原始套接字接受数据，如果可以，则返回对应原始套接字，并通过ipv6\_raw\_deliver函数接受套接字缓冲区中的数据内容***

***raw\_sk= sk\_head(&raw\_v6\_hashinfo.ht[nexthdr & (MAX\_INET\_PROTOS - 1)]);***

***if(raw\_sk && !ipv6\_raw\_deliver(skb, nexthdr))***

***raw\_sk= NULL;***

***returnraw\_sk != NULL;***

***}***

***\*/***

raw = raw6\_local\_deliver(skb, nexthdr);

//查找inet6\_protos表，确定是否注册过第四层协议，如果有，则调用对应的函数来接受数据包

hash = nexthdr & (MAX\_INET\_PROTOS - 1);

if ((ipprot = rcu\_dereference(inet6\_protos[hash])) != NULL) {

intret;

if (ipprot->flags & INET6\_PROTO\_FINAL) {

structipv6hdr \*hdr;

/\*Free reference early: we don't need it any more,

and it may hold ip\_conntrack module loaded

indefinitely. \*/

nf\_reset(skb);

skb\_postpull\_rcsum(skb,skb\_network\_header(skb),

skb\_network\_header\_len(skb));

hdr = ipv6\_hdr(skb);

if(ipv6\_addr\_is\_multicast(&hdr->daddr) &&

!ipv6\_chk\_mcast\_addr(skb->dev,&hdr->daddr,

&hdr->saddr) &&

!ipv6\_is\_mld(skb, nexthdr))

gotodiscard;

}

if (!(ipprot->flags & INET6\_PROTO\_NOPOLICY) &&

!xfrm6\_policy\_check(NULL, XFRM\_POLICY\_IN,skb))

goto discard;

//通过ipprot的handler指针调用上层协议的接受函数；

//对于TCP，调用tcp\_v6\_rcv;对于UDP，调用udpv6\_rcv;对于ICMP，调用icmpv6\_rcv

ret = ipprot->handler(skb);

if(ret > 0)

goto resubmit;

else if (ret == 0)

IP6\_INC\_STATS\_BH(net,idev, IPSTATS\_MIB\_INDELIVERS);

} else {

if (!raw) {

if (xfrm6\_policy\_check(NULL, XFRM\_POLICY\_IN, skb)) {

IP6\_INC\_STATS\_BH(net,idev,

IPSTATS\_MIB\_INUNKNOWNPROTOS);

icmpv6\_send(skb,ICMPV6\_PARAMPROB,

ICMPV6\_UNK\_NEXTHDR, nhoff,

skb->dev);

}

} else

IP6\_INC\_STATS\_BH(net,idev, IPSTATS\_MIB\_INDELIVERS);

kfree\_skb(skb);

}

rcu\_read\_unlock();

return 0;

discard:

IP6\_INC\_STATS\_BH(net,idev, IPSTATS\_MIB\_INDISCARDS);

rcu\_read\_unlock();

kfree\_skb(skb);

return 0;

}

**六.数据包发送及转发流程分析**

发送及转发的流程为：dst\_out--->ipv6\_output----> ipv6\_output 2à ipv6\_output\_finish

或者ip6\_forward-àip6\_forward\_finish-à dst\_out --->ipv6\_output----> ipv6\_output 2à ipv6\_output\_finish

==###########################################################################==

//如果需要转发数据包，则调用ip6\_forward执行转发过程，最后通过ip6\_forward\_finish函数把数据包交给ipv6模块的ip6\_output函数，进入发送流程。

==###########################################################################==

最终生成的IP数据报的路由称为目的入口(dst\_entry)，目的入口反映了相邻的外部主机在主机内部的一种“映象”，目的入口在内核中的定义如下

struct dst\_entry

{

struct rcu\_head rcu\_head;

struct dst\_entry \*child;

struct net\_device \*dev;

short error;

short obsolete;

int flags;

#define DST\_HOST 1

#define DST\_NOXFRM 2

#define DST\_NOPOLICY 4

#define DST\_NOHASH 8

unsigned long expires;

unsigned short header\_len; /\* more space at head required \*/

unsigned short trailer\_len; /\* space to reserve at tail \*/

unsigned int rate\_tokens;

unsigned long rate\_last; /\* rate limiting for ICMP \*/

struct dst\_entry \*path;

struct neighbour \*neighbour;

struct hh\_cache \*hh;

#ifdef CONFIG\_XFRM

struct xfrm\_state \*xfrm;

#else

void \*\_\_pad1;

#endif

int (\*input)(structsk\_buff\*);

int (\*output)(structsk\_buff\*);

struct dst\_ops \*ops;

u32 metrics[RTAX\_MAX];

#ifdef CONFIG\_NET\_CLS\_ROUTE

\_\_u32 tclassid;

#else

\_\_u32 \_\_pad2;

#endif

/\*

\* Align \_\_refcnt to a 64 bytes alignment

\* (L1\_CACHE\_SIZE would be too much)

\*/

#ifdef CONFIG\_64BIT

long \_\_pad\_to\_align\_refcnt[2];

#else

long \_\_pad\_to\_align\_refcnt[1];

#endif

/\*

\* \_\_refcnt wants to be on a different cacheline from

\* input/output/ops or performance tanks badly

\*/

atomic\_t \_\_refcnt; /\* client references \*/

int \_\_use;

unsigned long lastuse;

union {

struct dst\_entry \*next;

struct rtable \*rt\_next;

struct rt6\_info \*rt6\_next;

struct dn\_route \*dn\_next;

};

};

如果接收了转发给其他主机的数据包，则ip6\_rcv\_finish通过dst\_input接口把数据包传递给函数ip6\_forward。该函数执行一些检测：确定设备是否支持转发、判断跳数限制是否失效。最后调用ip6\_forwart\_finish执行转发

int ip6\_forward(struct sk\_buff \*skb)

{

struct dst\_entry \*dst = skb\_dst(skb);

struct ipv6hdr \*hdr = ipv6\_hdr(skb);

struct inet6\_skb\_parm \*opt = IP6CB(skb);

struct net \*net = dev\_net(dst->dev);

//检测设备是否支持转发IPv6数据包

if (net->ipv6.devconf\_all->forwarding == 0)

gotoerror;

if (skb\_warn\_if\_lro(skb))

goto drop;

//ipsec策略检测

if (!xfrm6\_policy\_check(NULL, XFRM\_POLICY\_FWD, skb)) {

IP6\_INC\_STATS(net,ip6\_dst\_idev(dst), IPSTATS\_MIB\_INDISCARDS);

goto drop;

}

***/\****

***static inline void skb\_forward\_csum(structsk\_buff \*skb)***

***{***

***/\*Unfortunately we don't support this one. Any brave souls? \*/***

***if (skb->ip\_summed == CHECKSUM\_COMPLETE)***

***skb->ip\_summed = CHECKSUM\_NONE;***

***}***

***\*/***

skb\_forward\_csum(skb);

/\*

\* WeDO NOT make any processing on

\* RApackets, pushing them to user level AS IS

\* withoutane WARRANTY that application will be able

\* tointerpret them. The reason is that we

\* cannotmake anything clever here.

\*

\* Weare not end-node, so that if packet contains

\* AH/ESP,we cannot make anything.

\* Defragmentationalso would be mistake, RA packets

\* cannotbe fragmented, because there is no warranty

\* thatdifferent fragments will go along one path. --ANK

\*对RA数据包不做处理，提交给用户态。

***/\****

***static int ip6\_call\_ra\_chain(struct sk\_buff \*skb, int sel)***

***{***

***struct ip6\_ra\_chain \*ra;***

***struct sock \*last = NULL;***

***read\_lock(&ip6\_ra\_lock);***

***for (ra = ip6\_ra\_chain; ra; ra =ra->next) {***

***struct sock \*sk = ra->sk;***

***if (sk && ra->sel ==sel &&***

***(!sk->sk\_bound\_dev\_if ||***

***sk->sk\_bound\_dev\_if ==skb->dev->ifindex)) {***

***if (last) {***

***struct sk\_buff \*skb2= skb\_clone(skb, GFP\_ATOMIC);***

***if (skb2)***

***rawv6\_rcv(last,skb2);***

***}***

***last = sk;***

***}***

***}***

***if (last) {***

***rawv6\_rcv(last, skb);***

***read\_unlock(&ip6\_ra\_lock);***

***return 1;***

***}***

***read\_unlock(&ip6\_ra\_lock);***

***return 0;***

***}***

***\*/***

if (opt->ra) {

u8\*ptr = skb\_network\_header(skb) + opt->ra;

if (ip6\_call\_ra\_chain(skb, (ptr[2]<<8) + ptr[3]))

return 0;

}

//检查和递减TTL

/\*

\* checkand decrement ttl

\*/

//如果跳数限制小于1，则发出icmpv6\_time\_exceed消息

if (hdr->hop\_limit <= 1) {

/\*Force OUTPUT device used as source address \*/

skb->dev = dst->dev;

icmpv6\_send(skb,ICMPV6\_TIME\_EXCEED, ICMPV6\_EXC\_HOPLIMIT,

0, skb->dev);

IP6\_INC\_STATS\_BH(net,

ip6\_dst\_idev(dst), IPSTATS\_MIB\_INHDRERRORS);

kfree\_skb(skb);

return-ETIMEDOUT;

}

/\*XXX: idev->cnf.proxy\_ndp? \*/

if (net->ipv6.devconf\_all->proxy\_ndp &&

pneigh\_lookup(&nd\_tbl, net,&hdr->daddr, skb->dev, 0)) {

intproxied = ip6\_forward\_proxy\_check(skb);

if(proxied > 0)

return ip6\_input(skb);

else if (proxied < 0) {

IP6\_INC\_STATS(net,ip6\_dst\_idev(dst),

IPSTATS\_MIB\_INDISCARDS);

goto drop;

}

}

//ipsec路由转发

if (!xfrm6\_route\_forward(skb)) {

IP6\_INC\_STATS(net,ip6\_dst\_idev(dst), IPSTATS\_MIB\_INDISCARDS);

goto drop;

}

dst = skb\_dst(skb);

/\*IPv6 specs 规格say nothing about it, but it is clear that we cannot

send redirects to source routed frames.

We don't send redirects to framesdecapsulated拆分 from IPsec.

\*/

if(skb->dev == dst->dev && dst->neighbour &&opt->srcrt == 0 &&

!skb\_sec\_path(skb)) {

struct in6\_addr \*target = NULL;

struct rt6\_info \*rt;

struct neighbour \*n = dst->neighbour;

/\*

\* incomingand outgoing devices are the same

\* senda redirect.

\*/

rt = (struct rt6\_info \*) dst;

if ((rt->rt6i\_flags & RTF\_GATEWAY))

target = (struct in6\_addr\*)&n->primary\_key;

else

target = &hdr->daddr;

/\*Limit redirects both by destination (here)

and by source (inside ndisc\_send\_redirect)

\*/

if (xrlim\_allow(dst, 1\*HZ))

ndisc\_send\_redirect(skb, n, target);

} else {

intaddrtype = ipv6\_addr\_type(&hdr->saddr);

//丢弃源地址是多播、环回和本地链路类型的数据包

/\*This check is security critical. \*/

if (addrtype == IPV6\_ADDR\_ANY ||

addrtype & (IPV6\_ADDR\_MULTICAST |IPV6\_ADDR\_LOOPBACK))

goto error;

if (addrtype & IPV6\_ADDR\_LINKLOCAL) {

icmpv6\_send(skb,ICMPV6\_DEST\_UNREACH,

ICMPV6\_NOT\_NEIGHBOUR,0, skb->dev);

goto error;

}

}

//如果数据包长度大于MTU，发送ICMPV6\_PKT\_TOOBIG消息

if (skb->len > dst\_mtu(dst)) {

/\*Again, force OUTPUT device used as source address \*/

skb->dev = dst->dev;

icmpv6\_send(skb, ICMPV6\_PKT\_TOOBIG, 0, dst\_mtu(dst), skb->dev);

IP6\_INC\_STATS\_BH(net,

ip6\_dst\_idev(dst), IPSTATS\_MIB\_INTOOBIGERRORS);

IP6\_INC\_STATS\_BH(net,

ip6\_dst\_idev(dst), IPSTATS\_MIB\_FRAGFAILS);

kfree\_skb(skb);

return -EMSGSIZE;

}

//一般而言，skb通过引用计数实现共享，前提是大家不能修改skb head 和data的内容。 如果需要修改的话，就有必要调用skb\_cow重新申请一个啦

if(skb\_cow(skb, dst->dev->hard\_header\_len)) {

IP6\_INC\_STATS(net,ip6\_dst\_idev(dst), IPSTATS\_MIB\_OUTDISCARDS);

goto drop;

}

//获取ip头部

hdr = ipv6\_hdr(skb);

/\*Mangling hops number delayed to point after skb COW \*/

//跳数限制减一

hdr->hop\_limit--;

IP6\_INC\_STATS\_BH(net,ip6\_dst\_idev(dst), IPSTATS\_MIB\_OUTFORWDATAGRAMS);

//调用ip6\_forward\_finish完成转发最后的操作

return NF\_HOOK(PF\_INET6, NF\_INET\_FORWARD, skb, skb->dev, dst->dev,

ip6\_forward\_finish);

error:

IP6\_INC\_STATS\_BH(net,ip6\_dst\_idev(dst), IPSTATS\_MIB\_INADDRERRORS);

drop:

kfree\_skb(skb);

return -EINVAL;

}

static inline int ip6\_forward\_finish(structsk\_buff \*skb)

{

return dst\_output(skb);

}

/\* Output packet to network fromtransport. \*/

static inline int dst\_output(struct sk\_buff\*skb)

{

return skb\_dst(skb)->output(skb);

}

==#################################################################====

数据包发送流程

Dst\_output是由路由项注册的外出函数，指向ip6\_output

static inline int dst\_output(struct sk\_buff \*skb)

{

return skb\_dst(skb)->output(skb);

}

int ip6\_output(struct sk\_buff \*skb)

{

struct inet6\_dev \*idev = ip6\_dst\_idev(skb\_dst(skb));

if(unlikely(idev->cnf.disable\_ipv6)) {

IP6\_INC\_STATS(dev\_net(skb\_dst(skb)->dev),idev,

IPSTATS\_MIB\_OUTDISCARDS);

kfree\_skb(skb);

return 0;

}

//如果需要分片，调用ip6\_fragment函数处理

if((skb->len > ip6\_skb\_dst\_mtu(skb) && !skb\_is\_gso(skb)) ||

dst\_allfrag(skb\_dst(skb)))

return ip6\_fragment(skb, ip6\_output2);

else

return ip6\_output2(skb);

}

static int ip6\_output2(struct sk\_buff \*skb)

{

struct dst\_entry \*dst = skb\_dst(skb);

struct net\_device \*dev = dst->dev;

// 把数据包的类型设置为IPv6类型

skb->protocol= htons(ETH\_P\_IPV6);

skb->dev= dev;

//检查是否为多播地址

if (ipv6\_addr\_is\_multicast(&ipv6\_hdr(skb)->daddr)) {

//sk\_buff->sk这是一个指向拥有这个sk\_buff的sock结构的指针。这个指针在网络包由本机发出或者由本机进程接收时有效，因为插口相关的信息被L4(TCP或 UDP)或者用户空间程序使用。如果sk\_buff只在转发中使用(这意味着，源地址和目的地址都不是本机地址)，这个指针是NULL

struct ipv6\_pinfo\* np = skb->sk ? inet6\_sk(skb->sk) : NULL;

struct inet6\_dev \*idev = ip6\_dst\_idev(skb\_dst(skb));

if (!(dev->flags & IFF\_LOOPBACK) && (!np || np->mc\_loop)&&

((mroute6\_socket(dev\_net(dev)) &&

!(IP6CB(skb)->flags &IP6SKB\_FORWARDED)) ||

ipv6\_chk\_mcast\_addr(dev,&ipv6\_hdr(skb)->daddr,

&ipv6\_hdr(skb)->saddr))) {

structsk\_buff \*newskb = skb\_clone(skb, GFP\_ATOMIC);

/\*Do not check for IFF\_ALLMULTI; multicast routing

is not supported in any case.

\*/

if(newskb)

//调用ip6\_dev\_loopback\_xmit环回发送数据包

NF\_HOOK(PF\_INET6,NF\_INET\_POST\_ROUTING, newskb,

NULL,newskb->dev,

ip6\_dev\_loopback\_xmit);

if (ipv6\_hdr(skb)->hop\_limit == 0) {

IP6\_INC\_STATS(dev\_net(dev),idev,

IPSTATS\_MIB\_OUTDISCARDS);

kfree\_skb(skb);

return0;

}

}

IP6\_UPD\_PO\_STATS(dev\_net(dev),idev, IPSTATS\_MIB\_OUTMCAST,

skb->len);

}

//调用ip6\_output\_finish进一步处理数据包

return NF\_HOOK(PF\_INET6, NF\_INET\_POST\_ROUTING, skb, NULL, skb->dev,

ip6\_output\_finish);

}

static int ip6\_output\_finish(struct sk\_buff \*skb)

{

/\*dst\_entry可以理解为路由表的缓冲区,每次主机发送数据时询问路由表后,都会将记录记在一个cache内.dst中有能指向其neighbour的指针,通过neighbour可以找到下一跳地址\*/

struct dst\_entry \*dst = skb\_dst(skb);

//如果有缓存指针hh，则通过neigh\_hh\_output发送数据；否则通过dst->neighbour->output发送数据；hh\_cache中存储的是链路头的一些相关信息,可以加快数据包的传输(因为有些情况下不用查看路由表,直接到此缓冲区查看).\*/

if (dst->hh)

return neigh\_hh\_output(dst->hh, skb);

else if (dst->neighbour)

return dst->neighbour->output(skb);

IP6\_INC\_STATS\_BH(dev\_net(dst->dev),

ip6\_dst\_idev(dst), IPSTATS\_MIB\_OUTNOROUTES);

kfree\_skb(skb);

return -EINVAL;

}

static inline int neigh\_hh\_output(struct hh\_cache \*hh, struct sk\_buff \*skb)

{

unsigned seq;

int hh\_len;

do {

int hh\_alen;

**/\***

**static \_\_always\_inline unsigned read\_seqbegin(constseqlock\_t \*sl)**

*{*

*unsignedret;*

*repeat:*

*ret =sl->sequence;*

*smp\_rmb();*

*if (unlikely(ret & 1)) {*

*cpu\_relax();*

*gotorepeat;*

*}*

*return ret;*

**}**

**\*/**

seq = read\_seqbegin(&hh->hh\_lock);

hh\_len = hh->hh\_len;

hh\_alen = HH\_DATA\_ALIGN(hh\_len);

//将缓冲区数据拷贝到skb中

memcpy(skb->data - hh\_alen, hh->hh\_data, hh\_alen);

} while (read\_seqretry(&hh->hh\_lock, seq));

skb\_push(skb,hh\_len);

return hh->hh\_output(skb);

}

unsigned char \*skb\_push(struct sk\_buff \*skb, unsigned int len)

{

skb->data -= len;

skb->len += len;

if (unlikely(skb->data<skb->head))

skb\_under\_panic(skb, len,\_\_builtin\_return\_address(0));

return skb->data;

}

==###################################################################==

UDP发送到IP层的函数

int ip6\_push\_pending\_frames(struct sock \*sk)

{

struct sk\_buff \*skb, \*tmp\_skb;

struct sk\_buff \*\*tail\_skb;

struct in6\_addr final\_dst\_buf, \*final\_dst = &final\_dst\_buf;

struct inet\_sock \*inet = inet\_sk(sk);

struct ipv6\_pinfo \*np = inet6\_sk(sk);

struct net \*net = sock\_net(sk);

struct ipv6hdr \*hdr;

struct ipv6\_txoptions \*opt = np->cork.opt;

struct rt6\_info \*rt = (struct rt6\_info \*)inet->cork.dst;

struct flowi \*fl = &inet->cork.fl;

unsigned char proto = fl->proto;

int err = 0;

//检查发送队列是否为空，并返回队首的套接字缓冲区skb

if ((skb = \_\_skb\_dequeue(&sk->sk\_write\_queue)) == NULL)

goto out;

tail\_skb= &(skb\_shinfo(skb)->frag\_list);

//如果有扩展头部信息，则调整skb->data指向IP包头部

/\*move skb->data to ip header from ext header \*/

if (skb->data < skb\_network\_header(skb))

\_\_skb\_pull(skb,skb\_network\_offset(skb));

//遍历套接字发送队列，调整数据长度

while ((tmp\_skb = \_\_skb\_dequeue(&sk->sk\_write\_queue)) != NULL) {

\_\_skb\_pull(tmp\_skb,skb\_network\_header\_len(skb));

\*tail\_skb= tmp\_skb;

tail\_skb= &(tmp\_skb->next);

skb->len+= tmp\_skb->len;

skb->data\_len+= tmp\_skb->len;

skb->truesize+= tmp\_skb->truesize;

tmp\_skb->destructor= NULL;

tmp\_skb->sk= NULL;

}

/\*Allow local fragmentation. \*/

if (np->pmtudisc < IPV6\_PMTUDISC\_DO)

skb->local\_df= 1;

ipv6\_addr\_copy(final\_dst,&fl->fl6\_dst);

\_\_skb\_pull(skb,skb\_network\_header\_len(skb));

//填充ipv6的扩展头部

if (opt && opt->opt\_flen)

ipv6\_push\_frag\_opts(skb,opt, &proto);

if (opt && opt->opt\_nflen)

ipv6\_push\_nfrag\_opts(skb,opt, &proto, &final\_dst);

//记录IPv6头部起始位置

skb\_push(skb,sizeof(struct ipv6hdr));

skb\_reset\_network\_header(skb);

hdr = ipv6\_hdr(skb);

//设置流标签

\*(\_\_be32\*)hdr = fl->fl6\_flowlabel |

htonl(0x60000000 |((int)np->cork.tclass << 20));

//设置跳数限制

hdr->hop\_limit = np->cork.hop\_limit;

//设置下一个包头

hdr->nexthdr = proto;

//为ipv6设置地址

ipv6\_addr\_copy(&hdr->saddr,&fl->fl6\_src);

ipv6\_addr\_copy(&hdr->daddr,final\_dst);

//设置属性和子网掩码

skb->priority = sk->sk\_priority;

skb->mark = sk->sk\_mark;

//给套接字缓冲区skb指定路由表项信息；为数据包的进入ipv6发送流程设置具体的方法

skb\_dst\_set(skb,dst\_clone(&rt->u.dst));

IP6\_UPD\_PO\_STATS(net,rt->rt6i\_idev, IPSTATS\_MIB\_OUT, skb->len);

if(proto == IPPROTO\_ICMPV6) {

struct inet6\_dev \*idev = ip6\_dst\_idev(skb\_dst(skb));

ICMP6MSGOUT\_INC\_STATS\_BH(net,idev, icmp6\_hdr(skb)->icmp6\_type);

ICMP6\_INC\_STATS\_BH(net,idev, ICMP6\_MIB\_OUTMSGS);

}

//程序执行到这里，已经为dst\_output配置完skb的处理信息，内核将从这里跳转到dst\_output函数，通过ip6\_output函数进入ipv6流程

err = ip6\_local\_out(skb);

if (err) {

If (err > 0)

err = np->recverr ? net\_xmit\_errno(err) : 0;

if(err)

goto error;

}

out:

ip6\_cork\_release(inet,np);

return err;

error:

gotoout;

}

int ip6\_local\_out(struct sk\_buff \*skb)

{

interr;

err = \_\_ip6\_local\_out(skb);

if (likely(err == 1))

err = dst\_output(skb);

return err;

}

int \_\_ip6\_local\_out(struct sk\_buff \*skb)

{

int len;

len= skb->len - sizeof(struct ipv6hdr);

if (len > IPV6\_MAXPLEN)

len= 0;

// 设置载荷长度为0；unsigned short payload\_len; //载荷长度16位

ipv6\_hdr(skb)->payload\_len= htons(len);

return nf\_hook(PF\_INET6, NF\_INET\_LOCAL\_OUT, skb, NULL, skb\_dst(skb)->dev,

dst\_output);

}

==###################################################################==

TCP发送到IP层的函数

int ip6\_xmit(struct sock \*sk, struct sk\_buff \*skb, struct flowi \*fl,

struct ipv6\_txoptions \*opt, int ipfragok)

{

struct net \*net = sock\_net(sk);

struct ipv6\_pinfo \*np = inet6\_sk(sk);

struct in6\_addr \*first\_hop = &fl->fl6\_dst;

struct dst\_entry \*dst = skb\_dst(skb);

struct ipv6hdr \*hdr;

u8 proto = fl->proto;

int seg\_len = skb->len;

int hlimit, tclass;

u32 mtu;

//如果需要填充ipv6扩展头部，则调整存储头部空间

if (opt) {

unsigned int head\_room;

/\*First: exthdrs may take lots of space (~8K for now)

MAX\_HEADER is not enough.

\*/

head\_room= opt->opt\_nflen + opt->opt\_flen;

seg\_len+= head\_room;

head\_room+= sizeof(struct ipv6hdr) + LL\_RESERVED\_SPACE(dst->dev);

if(skb\_headroom(skb) < head\_room) {

structsk\_buff \*skb2 = skb\_realloc\_headroom(skb, head\_room);

if(skb2 == NULL) {

IP6\_INC\_STATS(net,ip6\_dst\_idev(skb\_dst(skb)),

IPSTATS\_MIB\_OUTDISCARDS);

kfree\_skb(skb);

return -ENOBUFS;

}

kfree\_skb(skb);

skb = skb2;

if (sk)

skb\_set\_owner\_w(skb,sk);

}

//填充IPv6的扩展头部信息

if (opt->opt\_flen)

ipv6\_push\_frag\_opts(skb,opt, &proto);

if (opt->opt\_nflen)

ipv6\_push\_nfrag\_opts(skb,opt, &proto, &first\_hop);

}

//记录ipv6头部的起始位置

skb\_push(skb,sizeof(struct ipv6hdr));

skb\_reset\_network\_header(skb);

hdr = ipv6\_hdr(skb);

//设置分片运行标志

/\*Allow local fragmentation. \*/

if (ipfragok)

skb->local\_df = 1;

/\*

\* Fillin the IPv6 header

\*/

//计算跳转限制

hlimit = -1;

if (np)

hlimit = np->hop\_limit;

if (hlimit < 0)

hlimit = ip6\_dst\_hoplimit(dst);

tclass = -1;

if (np)

tclass = np->tclass;

if (tclass < 0)

tclass = 0;

//设置流标签

\*(\_\_be32\*)hdr = htonl(0x60000000 | (tclass << 20)) | fl->fl6\_flowlabel;

//设置载荷长度，下一个扩展头协议，跳转限制

hdr->payload\_len = htons(seg\_len);

hdr->nexthdr = proto;

hdr->hop\_limit= hlimit;

//设置ipv6头部得ip地址，属性，子网掩码

ipv6\_addr\_copy(&hdr->saddr,&fl->fl6\_src);

ipv6\_addr\_copy(&hdr->daddr,first\_hop);

skb->priority = sk->sk\_priority;

skb->mark = sk->sk\_mark;

//考虑MTU值，如果包太大，就要发送ICMPV6\_PKT\_TOOBIG消息

mtu = dst\_mtu(dst);

if ((skb->len <= mtu) || skb->local\_df || skb\_is\_gso(skb)) {

IP6\_UPD\_PO\_STATS(net,ip6\_dst\_idev(skb\_dst(skb)),

IPSTATS\_MIB\_OUT, skb->len);

return NF\_HOOK(PF\_INET6, NF\_INET\_LOCAL\_OUT, skb, NULL, dst->dev,

dst\_output);

}

if(net\_ratelimit())

printk(KERN\_DEBUG"IPv6: sending pkt\_too\_big to self\n");

skb->dev = dst->dev;

icmpv6\_send(skb,ICMPV6\_PKT\_TOOBIG, 0, mtu, skb->dev);

IP6\_INC\_STATS(net,ip6\_dst\_idev(skb\_dst(skb)), IPSTATS\_MIB\_FRAGFAILS);

kfree\_skb(skb);

return -EMSGSIZE;

}

**七.总结**

经过前面的分析，现在可以理解 INET6 协议族在初始化的时候要做哪些事情：

1、注册 INET6 协议族，提供协议族的创建函数。

2、为所支持的传输层协议分别提供 struct proto\_ops、struct proto和struct inet\_protosw 结构，并注册到关联表中。

3、向设备驱动层注册 IPv6 数据包的处理函数

4、向网络层注册 TCP、UDP、RAW 等传输层的处理函数。

5、其它初始化工作

**1）.*注册INET6协议族***

对于 INET6 的实现来说，第一步是要注册 INET6 协议族。

static struct net\_proto\_family inet6\_family\_ops = {  
 .family = PF\_INET6,  
 .create = inet6\_create,  
 .owner = THIS\_MODULE,  
 };  
 sock\_register(&inet6\_family\_ops);

inet6\_create() 的实现：TBW

**2）.*为TCP ,UDP等传输层协议提供关联变***

1、初始化关联表

for(r = &inetsw6[0]; r < &inetsw6[SOCK\_MAX]; ++r)  
 INIT\_LIST\_HEAD(r);

2、RAW 的关联

static struct proto\_ops inet6\_sockraw\_ops = {  
 .family = PF\_INET6,  
 .owner = THIS\_MODULE,  
 .release = inet6\_release,  
 .bind = inet6\_bind,  
 .connect = inet\_dgram\_connect, /\* ok \*/  
 .socketpair = sock\_no\_socketpair, /\* a do nothing \*/  
 .accept = sock\_no\_accept, /\* a do nothing \*/  
 .getname = inet6\_getname,  
 .poll = datagram\_poll, /\* ok \*/  
 .ioctl = inet6\_ioctl, /\* must change \*/  
 .listen = sock\_no\_listen, /\* ok \*/  
 .shutdown = inet\_shutdown, /\* ok \*/  
 .setsockopt = sock\_common\_setsockopt, /\* ok \*/  
 .getsockopt = sock\_common\_getsockopt, /\* ok \*/  
 .sendmsg = inet\_sendmsg, /\* ok \*/  
 .recvmsg = sock\_common\_recvmsg, /\* ok \*/  
 .mmap = sock\_no\_mmap,  
 .sendpage = sock\_no\_sendpage,  
 };  
 struct proto rawv6\_prot = {  
 .name = "RAWv6",  
 .owner = THIS\_MODULE,  
 .close = rawv6\_close,  
 .connect = ip6\_datagram\_connect,  
 .disconnect = udp\_disconnect,  
 .ioctl = rawv6\_ioctl,  
 .init = rawv6\_init\_sk,  
 .destroy = inet6\_destroy\_sock,  
 .setsockopt = rawv6\_setsockopt,  
 .getsockopt = rawv6\_getsockopt,  
 .sendmsg = rawv6\_sendmsg,  
 .recvmsg = rawv6\_recvmsg,  
 .bind = rawv6\_bind,  
 .backlog\_rcv = rawv6\_rcv\_skb,  
 .hash = raw\_v6\_hash,  
 .unhash = raw\_v6\_unhash,  
 .obj\_size = sizeof(struct raw6\_sock),  
 };  
 static struct inet\_protosw rawv6\_protosw = {  
 .type = SOCK\_RAW,  
 .protocol = IPPROTO\_IP, /\* wild card \*/  
 .prot = &rawv6\_prot,  
 .ops = &inet6\_sockraw\_ops,  
 .capability = CAP\_NET\_RAW,  
 .no\_check = UDP\_CSUM\_DEFAULT,  
 .flags = INET\_PROTOSW\_REUSE,  
 };   
 inet6\_register\_protosw(&rawv6\_protosw);

3、UDP 的关联

struct proto\_ops inet6\_dgram\_ops = {  
 .family = PF\_INET6,  
 .owner = THIS\_MODULE,  
 .release = inet6\_release,  
 .bind = inet6\_bind,  
 .connect = inet\_dgram\_connect, /\* ok \*/  
 .socketpair = sock\_no\_socketpair, /\* a do nothing \*/  
 .accept = sock\_no\_accept, /\* a do nothing \*/  
 .getname = inet6\_getname,  
 .poll = udp\_poll, /\* ok \*/  
 .ioctl = inet6\_ioctl, /\* must change \*/  
 .listen = sock\_no\_listen, /\* ok \*/  
 .shutdown = inet\_shutdown, /\* ok \*/  
 .setsockopt = sock\_common\_setsockopt, /\* ok \*/  
 .getsockopt = sock\_common\_getsockopt, /\* ok \*/  
 .sendmsg = inet\_sendmsg, /\* ok \*/  
 .recvmsg = sock\_common\_recvmsg, /\* ok \*/  
 .mmap = sock\_no\_mmap,  
 .sendpage = sock\_no\_sendpage,  
 };  
 struct proto udpv6\_prot = {  
 .name = "UDPv6",  
 .owner = THIS\_MODULE,  
 .close = udpv6\_close,  
 .connect = ip6\_datagram\_connect,  
 .disconnect = udp\_disconnect,  
 .ioctl = udp\_ioctl,  
 .destroy = udpv6\_destroy\_sock,  
 .setsockopt = udpv6\_setsockopt,  
 .getsockopt = udpv6\_getsockopt,  
 .sendmsg = udpv6\_sendmsg,  
 .recvmsg = udpv6\_recvmsg,  
 .backlog\_rcv = udpv6\_queue\_rcv\_skb,  
 .hash = udp\_v6\_hash,  
 .unhash = udp\_v6\_unhash,  
 .get\_port = udp\_v6\_get\_port,  
 .obj\_size = sizeof(struct udp6\_sock),  
 };  
 static struct inet\_protosw udpv6\_protosw = {  
 .type = SOCK\_DGRAM,  
 .protocol = IPPROTO\_UDP,  
 .prot = &udpv6\_prot,  
 .ops = &inet6\_dgram\_ops,  
 .capability = -1,  
 .no\_check = UDP\_CSUM\_DEFAULT,  
 .flags = INET\_PROTOSW\_PERMANENT,  
 };  
 inet6\_register\_protosw(&udpv6\_protosw);

4、TCP 的关联

前面已经看过 TCP 相关的结构。

inet6\_register\_protosw(&tcpv6\_protosw);

**3）.*注册IPv6包的接收函数***

static struct packet\_type ipv6\_packet\_type = {  
 .type = \_\_constant\_htons(ETH\_P\_IPV6),  
 .func = ipv6\_rcv,  
 };

ipv6\_packet\_init()

dev\_add\_pack(&ipv6\_packet\_type);

**4）.*注册传输层协议***

static struct inet6\_protocol udpv6\_protocol = {  
 .handler = udpv6\_rcv,  
 .err\_handler = udpv6\_err,  
 .flags = INET6\_PROTO\_NOPOLICY|INET6\_PROTO\_FINAL,  
 };  
 static struct inet6\_protocol tcpv6\_protocol = {  
 .handler = tcp\_v6\_rcv,  
 .err\_handler = tcp\_v6\_err,  
 .flags = INET6\_PROTO\_NOPOLICY|INET6\_PROTO\_FINAL,  
 };  
 inet6\_add\_protocol(&udpv6\_protocol, IPPROTO\_UDP);  
 inet6\_add\_protocol(&tcpv6\_protocol, IPPROTO\_TCP);

RAW 不需要注册。

**5）.*其他***

此外，还要做其它初始化工作，包括 ICMPv6、IGMPv6、Neighbor discovery、route 等等的初始化。

**八.附录**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Structure | Register functions | description |
|  | net\_proto\_family | sock\_register | 注册协议族 |
|  | packet\_type | dev\_add\_pack | 向设备驱动层注册网络层协议处理函数 |
|  | inet6\_protocol | inet6\_add\_protocol | 向网络层注册传输层协议处理函数 |
|  | proto\_ops  BSD APIs 与 socket 层的接口 |  |  |
|  | Proto  Socket 层与传输层的接口 |  |  |
|  | inet\_protosw  将 struct proto\_ops 与 struct proto 对应起来 | inet6\_register\_protosw | 注册到系统的 struct inet\_protosw数组 inetsw6 中  此数组用于创建 socket 之用。 |
|  | Proto  Socket 层与传输层的接口 | proto\_register | 将传输层协议处理函数注册到系统中的 struct proto 的链表 proto\_list。  这个目的是为了在 proc 系统中显示各种协议的信息 |

本文是集合网上相关文档整理而来,版权归原作者