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）

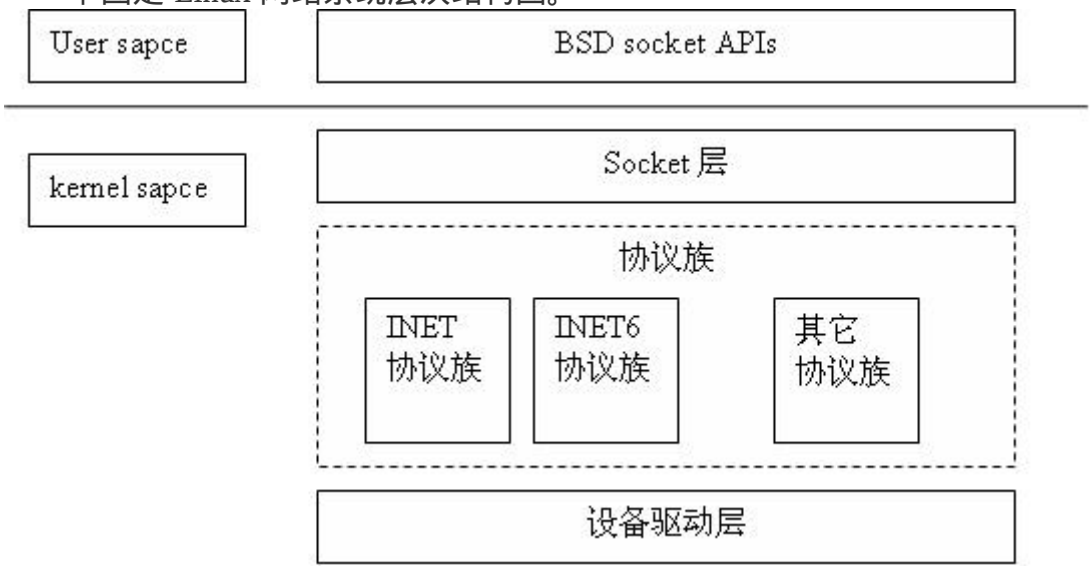
Linux 网络子系统功能上相当完备，它不仅支持 INET 协议族（也就是通常所说的 TCP/IP stack），而且还支持其它很多种协议族，如 DECnet, ROSE, NETBEUI 等。INET6 就是一种新增加的协议族。

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。



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

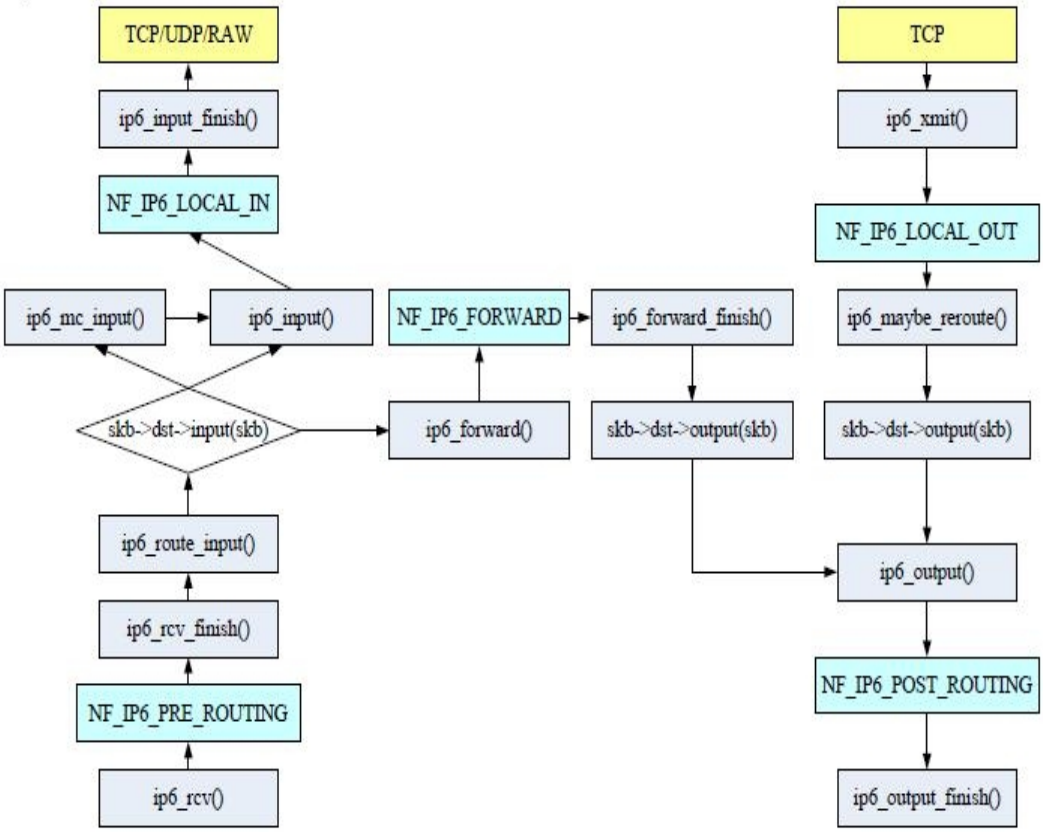
对于 ip6_route_input(skb)函数的分析，我们后面讲路由查找的时候再叙述，这里先跳过去。

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。

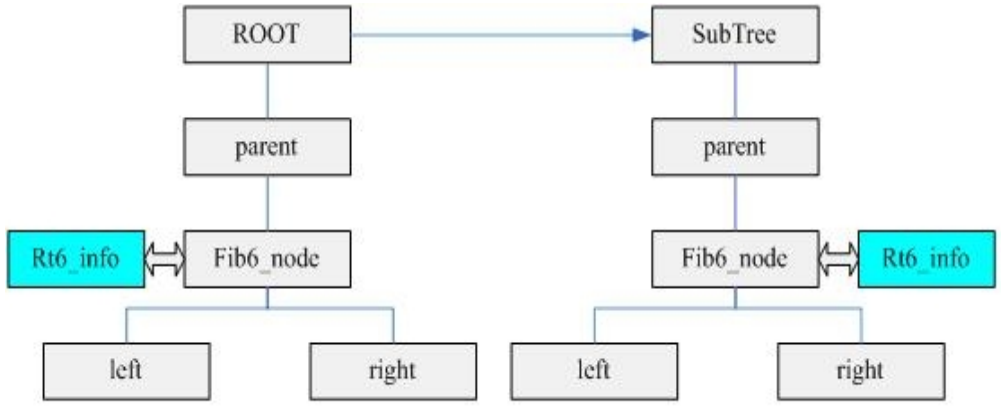
四.路由模块的处理

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	inet6_register_protosw	注册到系统的 struct inet_protosw 数组 inetsw6 中

	起来		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