第 13 章 网络：网络层、数据链路层

本章上接第 12 章，介绍因特网各层协议（IPv4）、数据链路层协议在内核中的实现，以及网络设备驱动程序的实现，最后简要介绍 IPv6 在内核中的实现。

13.1 网络层协议实现

因特网网络层协议存在于端系统和路由器中，而传输层协议只存在于端系统中。网络中的端系统和路由器可以有多个网络接口，每个接口都可以接收和发送数据。那么，到达节点的数据包（或自己产生的数据包）该从哪个网络接口发送出去呢？网络层协议主要功能就是根据数据包中指示的目的 IP 地址，确定数据包从本机哪个网络接口发送或转发出去。

网络层协议主要包括控制平面和数据平面。数据平面中存在一个（或多个）路由选择表，选择表项中指示了数据包目的 IP 地址对应的转发网络接口。数据平面根据数据包报头中的目的 IP 地址查找路由选择表，确认转发接口，并执行数据包的转发（发送）。

控制平面通过路由选择协议确定数据平面中的路由选择表，数据平面根据路由选择表转发数据包，用户也可以手工设置路由选择表项。本书主要关注数据平面在 Linux 内核中的实现。

13.1.1 概述

在介绍网络层协议实现前，先简要介绍一下 IP 地址的分类，因为在发送/转发数据包时需要根据目的 IP 地址的类型执行不同的操作。

一般根据传输消息的特征将（目的）IP 地址分为单播、组播（又称多播）和广播地址。

●**单播（unicast）**：一个单播 IP 地址只能标识一台用户主机，一台用户主机只能识别一个单播 IP 地址。一份使用单播 IP 地址为目的地址的 IP 数据包，只能被一台用户主机接收。

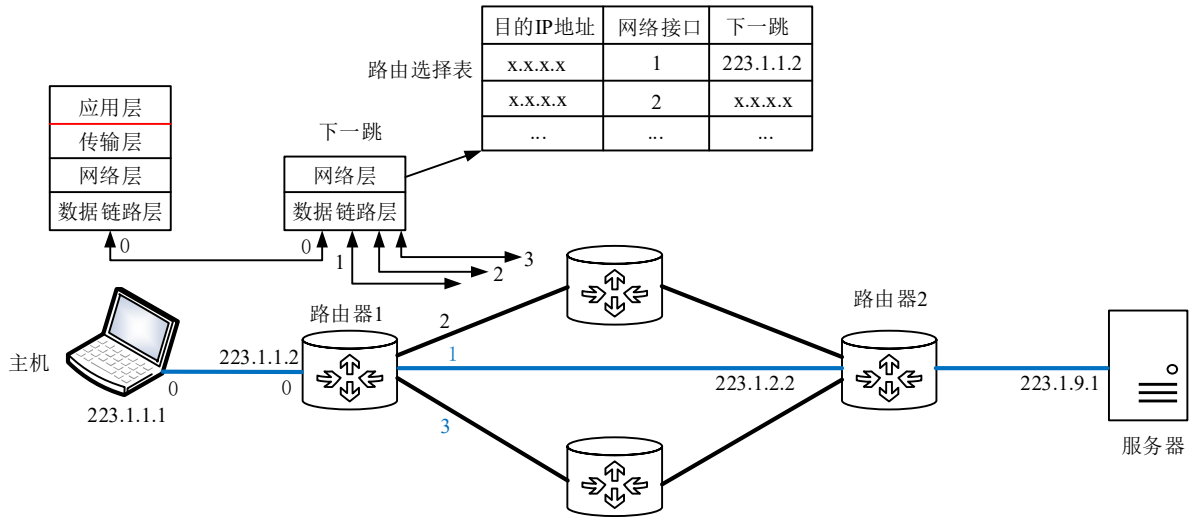
●**组播（multicast）**：一个组播 IP 地址能够标识网络不同位置的多个用户主机，一台用户主机可以同时识别多个组播 IP 地址。一份使用组播 IP 地址为目的地址的 IP 数据包，能够被网络不同位置的多个用户主机接收。组播地址范围为 224.0.0.0~239.255.255.255。

●**广播（broadcast）**：一个广播 IP 地址能够标识某确定网段内（子网）的所有用户主机，一份使用广播 IP 地址为目的地址的 IP 数据包，能够被该网段内的所有用户主机接收。IP 广播数据包不能跨网段传播。广播 IP 地址的主机部分全为 1，如：C 类网络 192.168.1.0 的默认子网掩码为 255.255.255.0，其广播地址为 192.168.1.255，其主机部分为十进制数 255，表示数据包广播到此 C 类网络内的所有主机。

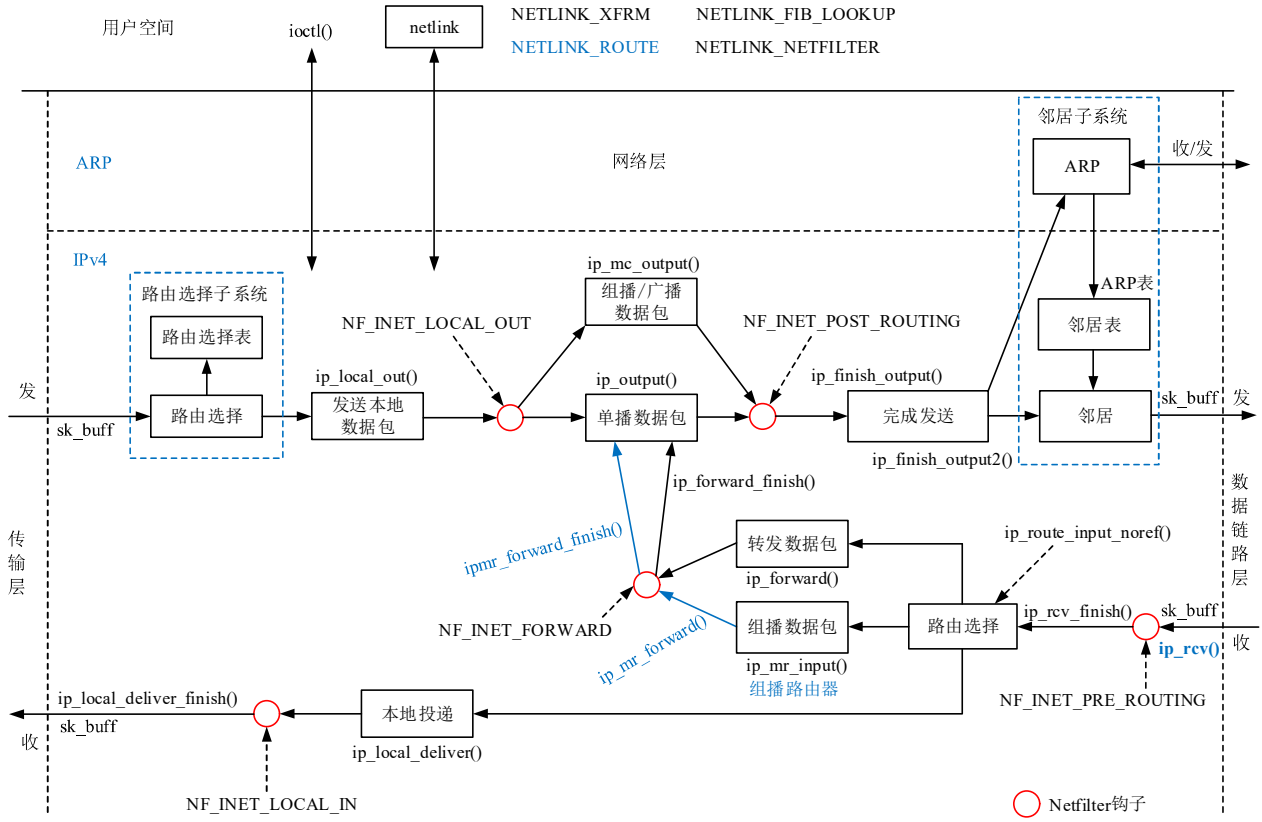
注：UDP 中有单播、组播和广播，而 TCP 是面向连接的点对点通信，只有单播，没有组播和广播。

1 协议实现框架

如下图所示，在端系统和路由器的网络层中存在一个或多个路由选择表，指示了发往某个地址（或地址范围）的数据包，从哪个网络接口发送出去。数据包在路由器之间接力传输，路由器通过路由选择表，为发送的数据包选择最优的路径。



IPv4 网络层协议发送、接收数据包流程如下图所示：



IPv4 网络层发送数据包的流程简述如下：

传输层协议（UDP、TCP 等）将数据包传递到网络层，网络层首先进行路由选择查找，在路由选择表中根据数据包目的 IP 地址查找匹配的表项，以确定数据包下一步的走向。外发需确定外发的网络接口和下一跳的 IP 地址，单播数据包由 ip_output()函数处理，组播和广播数据包由 ip_mc_output()函数处理。

ip_output()和 ip_mc_output()函数随后调用 ip_finish_output()->ip_finish_output2()函数，在邻居子系统中依据输出网络接口编号和下一跳 IP 地址在邻居表中查找表示下一跳（下一个节点网络接口）的邻居实例。邻居结构中缓存了邻居物理地址（和 L2 层报头），包含解析邻居物理地址的函数指针等信息。如果邻居中已缓存了 L2 层报头，则将报头写入数据包，将数据包发往数据链路层。如果邻居没有缓存 L2 层报头（或邻居物理地址），则调用解析邻居物理地址函数广播 ARP 请求（ARP 协议），接收邻居发回的 ARP 应答，从而解析出邻居物理地址，然后再生成 L2 层报头写入数据包，最后发送数据包。

IPv4 网络层接收数据包的流程简述如下：

数据链路层接收的数据包调用 IPv4 协议定义的 packet_type 实例 **ip_packet_type** 中的 **ip_rcv()**函数，将数据包传入网络层。ip_rcv()函数首先进行的也是路由选择，根据路由选择结果对数据包调用不同的处理函数。如果是传递给本机的数据包，则调用 ip_local_deliver()函数将数据包传递给本机传输层。如果是需要转发的数据包，调用 ip_forward()函数转发数据包。如果本机是组播路由器，对组播数据包调用 ip_mr_input()函数处理。转发（组播）数据包最终也是由 ip_output()函数发出。

IPv4 网络层协议中包含两个类型的表，分别是路由选择表和邻居表。路由选择表位于路由选择子系统，表项根据数据包的目的地址指示数据包下一步的走向。邻居表位于邻居子系统中，邻居表中管理的是邻居实例。邻居实例中包含了下一跳（邻居）的物理地址、L2 层报头、解析邻居物理地址的函数指针等信息。在 ip_finish_output2()中将查找或创建邻居实例，在发送数据包前将解析邻居物理地址，依此生成 L2 层报头。

用户空间可运行路由选择协议守护进程生成路由选择表项，用户进程也可以通过 netlink 套接字或 ioctl()系统调用添加、删除路由选择表项等，含设置本机网络接口的本地 IP 地址。另外，用户进程也可以通过 netlink 套接字或 ioctl()系统调用添加、删除邻居实例等。

2 Netfilter 子系统

在上图发送/接收数据包的流程中，有 5 个红色圆圈，它们表示 Netfilter 钩子（挂载点）。挂载点处可注册回调函数，当数据包到达挂载点时，先调用挂载点注册的回调函数。回调函数可对数据包进行各种处理，甚至可以将数据包丢弃，终止数据包的处理。如果回调函数返回 1，则表示发送/接收流程继续正常往下执行。

内核通过 Netfilter 子系统提供钩子挂载点、注册回调函数的接口、执行回调函数等机制。网络子系统中许多功能（子系统）都是通过向 Netfilter 钩子（挂载点）注册回调函数来实现，如 IPsec、NAT 等。

在 IPv4 网络层数据包发送/接收流程中存在 5 个 Netfilter 钩子（挂载点）。挂载点名称如下：

●**NF_INET_PRE_ROUTING**：在接收路径中，在执行路由选择前调用此挂载点注册的回调函数。

●**NF_INET_LOCAL_IN**：在接收路径中，在将数据包传递给本机传输层协议前调用此挂载点注册的回调函数。

●**NF_INET_FORWARD**：转发数据包时，在转发前调用此挂载点注册的回调函数。

●**NF_INET_LOCAL_OUT**：发送本地产生数据包时，在执行路由选择后调用此挂载点注册的回调函数。

●**NF_INET_POST_ROUTING**：发送本地数据包和转发数据包时，在完成最后发送前（到达邻居子系统）调用此挂载点注册的回调函数。

不止是 IPv4 网络层协议，其它的网络层协议，如 IPv6、ARP 等也设置了 Netfilter 钩子（挂载点）。

Netfilter 子系统用于在网络层协议数据包处理流程中插入钩子（挂载点），各钩子可以注册回调函数。数据包到达钩子处，先调用钩子处注册的回调函数，回调函数返回后（返回结果需为 1）再继续执行下面的流程。

Netfilter 子系统提供了一个框架，用于用户/内核向以上的 Netfilter 钩子注册回调函数，从而对数据包执行各种操作，如修改地址或端口号、过滤数据包、写入日志等。这些 Netfilter 挂接点为 Netfilter 内核模块提供了基础设施，让用户/内核能够通过注册回调函数来执行各种任务。

Netfilter 子系统公共层代码位于/net/netfilter/目录下，内核若需要支持 Netfilter 子系统需要选择 NETFILTER 配置选项。

下述网络功能（子系统）利用了 Netfilter 子系统提供的框架，通过向 Netfilter 钩子注册回调函数实现期望的功能：

●数据包选择（iptables）：防火墙。

●数据包过滤

●网络地址转换（NAT）：根据某些规则来转换网络层报头中的源地址和目的地址，实现子网内主机

对外具有相同的 IP 地址，主机在子网内具有各自的 IP 地址。

●数据包操作（修改报头等）

●连接跟踪

●网络统计信息收集

■注册回调函数

各类型网络层协议中定义的钩子名称由一个整数标识（编号），例如：IPv4 中钩子名称如下：

```
enum nf_inet_hooks {      /*/include/uapi/linux/netfilter.h*/
        NF_INET_PRE_ROUTING,      /*0*/
        NF_INET_LOCAL_IN,         /*1*/
        NF_INET_FORWARD,          /*2*/
        NF_INET_LOCAL_OUT,        /*3*/
        NF_INET_POST_ROUTING,     /*4*/
        NF_INET_NUMHOOKS          /*钩子总数为 5*/
};
```

钩子处每个注册的回调函数由 nf_hook_ops 结构体实例表示，其中包含钩子回调函数指针、优先级、适用的网络层协议类型、挂载点（钩子）名称等信息。

内核在/net/netfilter/core.c 文件内定义了全局二维双链表数组，用于管理内核所有网络层协议、所有钩子处注册的 nf_hook_ops 实例：

```
struct list_head nf_hooks[NFPROTO_NUMPROTO][NF_MAX_HOOKS] __read_mostly;
```

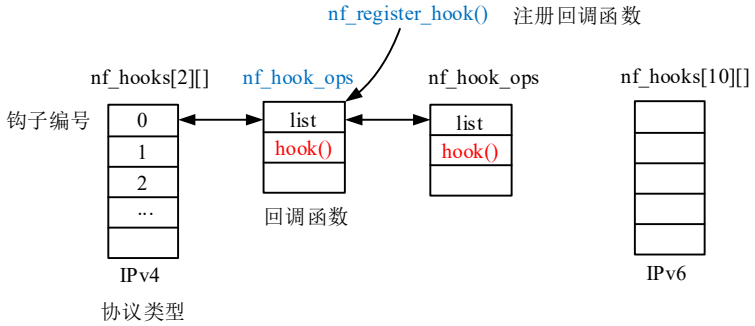
NFPROTO_NUMPROTO 表示使用钩子的网络层协议类型数量，定义在/include/uapi/linux/netfilter.h 头文件，也就是说每种协议对应一个双链表数组，每个钩子对应一个双链表数组项（管理 nf_hook_ops 实例）。

NF_MAX_HOOKS 表示最大钩子数量，目前为 8（/include/linux/netfilter_defs.h），因为 DECnet 网络协议中有 7 个挂接点。

使用 Netfilter 钩子的网络层协议类型定义如下（/include/uapi/linux/netfilter.h）：

```
enum {
    NFPROTO_UNSPEC =  0,
    NFPROTO_INET   =  1,
    NFPROTO_IPV4   =  2,      /*IPv4 协议*/
    NFPROTO_ARP    =  3,      /*ARP 协议*/
    NFPROTO_NETDEV =  5,
    NFPROTO_BRIDGE =  7,
    NFPROTO_IPV6   = 10,      /*IPv6 协议*/
    NFPROTO_DECNET = 12,
    NFPROTO_NUMPROTO,         /*协议类型最大数量*/
};
```

nf_hooks[][]二维数组结构如下图所示：



nf_hook_ops 结构体表示钩子处注册的一个回调函数，定义如下（/include/linux/netfilter.h）：

```
struct nf_hook_ops {
    struct list_head    list;     /*双链表成员，将实例链入到 nf_hooks[][]数组*/

    nf_hookfn           *hook;    /*钩子回调函数指针*/
    struct net_device   *dev;     /*网络设备*/
    struct module       *owner;
    void                *priv;    /*回调函数私有数据结构*/
    u_int8_t            pf;       /*网络层协议类型*/
    unsigned int        hooknum;  /*钩子名称（编号）*/
    int                 priority; /*回调函数优先级*/
};
```

nf_hook_ops 结构体主要成员简介如下：

●**list**：双链表成员，用于将 nf_hook_ops 实例链入到 nf_hooks[][]数组。

●**dev**：指向网络设备，在驱动程序中表示网络设备。

●**pf**：回调函数适用的网络层协议类型，见上文。

●**hooknum**：钩子名称（编号），由网络层协议定义。

●**priority**：优先级，一个挂接点可以注册多个回调函数，优先级数值越小的的回调函数越早被调用。IPv4 中回调函数优先级取值定义如下（/include/uapi/linux/netfilter_ipv4.h）：

```
enum nf_ip_hook_priorities {
    NF_IP_PRI_FIRST = INT_MIN,          /*最高优先级*/
    NF_IP_PRI_CONNTRACK_DEFRAG = -400,
    NF_IP_PRI_RAW = -300,
    NF_IP_PRI_SELINUX_FIRST = -225,
    NF_IP_PRI_CONNTRACK = -200,
    NF_IP_PRI_MANGLE = -150,
    NF_IP_PRI_NAT_DST = -100,
    NF_IP_PRI_FILTER = 0,
    NF_IP_PRI_SECURITY = 50,
    NF_IP_PRI_NAT_SRC = 100,
    NF_IP_PRI_SELINUX_LAST = 225,
    NF_IP_PRI_CONNTRACK_HELPER = 300,
    NF_IP_PRI_CONNTRACK_CONFIRM = INT_MAX,      /*优先级最低*/
    NF_IP_PRI_LAST = INT_MAX,           /*最低优先级*/
};
```

●**hook**：回调函数指针，函数原型如下（/include/linux/netfilter.h）：

```
typedef unsigned int nf_hookfn(const struct nf_hook_ops *ops,
                               struct sk_buff *skb,const struct nf_hook_state *state);
```

参数 state 指向的 nf_hook_state 结构体用于向回调函数传递信息，定义如下：

```
struct nf_hook_state {
    unsigned int hook;        /*挂接点编号*/
    int thresh;               /*优先级阀值，优先级数值小于 thresh（优先级更高）的回调函数不调用*/
    u_int8_t pf;              /*网络层协议类型*/
    struct net_device *in;    /*输入网络设备*/
    struct net_device *out;   /*输出网络设备*/
    struct sock *sk;          /*套接字*/
    struct list_head *hook_list;    /*钩子编号对应的 nf_hooks[][]双链表头*/
    int (*okfn)(struct sock *, struct sk_buff *);
                              /*回调函数返回 1（NF_ACCEPT）之后，继续调用的函数*/
};
```

钩子回调函数的返回值定义如下（/include/uapi/linux/netfilter.h）：

```
#define  NF_DROP      0  /*默默地丢弃数据包，不再执行后面的流程*/
#define  NF_ACCEPT    1  /*数据包正常地在网络层协议中传输，继续调用 okfn()函数*/
#define  NF_STOLEN    2  /*数据包不再继续传输（传输中断），由钩子回调函数处理*/
#define  NF_QUEUE     3  /*将数据包添加到一个队列，由用户空间处理*/
#define  NF_REPEAT    4  /*再次调用本回调函数*/
#define  NF_STOP      5   /*不再调用后面的回调函数了，数据包回到正常的处理流程*/
#define  NF_MAX_VERDICT  NF_STOP
```

注册 nf_hook_ops 实例的接口函数为 **nf_register_hook()**，定义如下（/net/netfilter/core.c）：

```
int nf_register_hook(struct nf_hook_ops *reg)
{
    struct list_head *nf_hook_list;
    struct nf_hook_ops *elem;

    mutex_lock(&nf_hook_mutex);
    switch (reg->pf) {        /*网络层协议类型*/
    case NFPROTO_NETDEV:
    ...
    default:
        nf_hook_list = &nf_hooks[reg->pf][reg->hooknum];   /*由协议类型和编号查找双链表头*/
        break;
    }

    list_for_each_entry(elem, nf_hook_list, list) {     /*按优先级在双链表中查找插入点*/
        if (reg->priority < elem->priority)          /*在双链表中按优先级数值从小到大排序*/
            break;
    }
    list_add_rcu(&reg->list, elem->list.prev);       /*将 nf_hook_ops 实例插入双链表*/
    mutex_unlock(&nf_hook_mutex);
    ...
```

```
    return 0;
}
```

nf_register_hook()函数根据 nf_hook_ops 实例中指示的网络层协议类型和钩子编号，查找 nf_hooks[][]二维数组中对应的项，即双链表表头，然后按优先级从高到低（优先级数值从小到大），在双链表中查找实例的插入位置，将实例添加到双链表中。

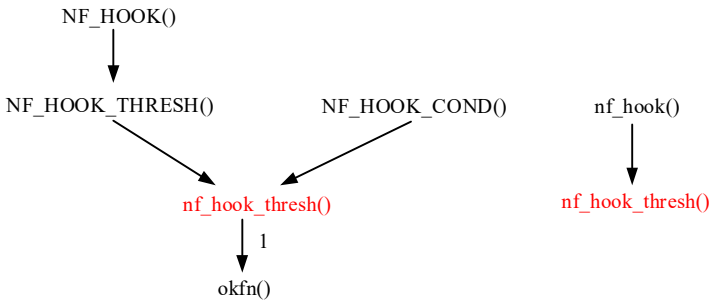
int **nf_register_hooks**(struct nf_hook_ops *reg, unsigned int n)：用于注册 nf_hook_ops 实例数组，reg 指向数组首成员，n 表示数组项数，函数内对每个数组项（nf_hook_ops 实例）调用 nf_register_hook()函数。

void **nf_unregister_hook**(struct nf_hook_ops *reg)：注销 nf_hook_ops 实例。

void **nf_unregister_hooks**(struct nf_hook_ops *reg, unsigned int n)：注销 nf_hook_ops 实例数组。

■调用回调函数

内核在/include/linux/netfilter.h 头文件中定义了调用钩子处注册回调函数的宏（函数），例如：**nf_hook()**、**NF_HOOK()**、**NF_HOOK_THRESH()**、**NF_HOOK_COND()**等。函数调用关系如下图所示：



以上所有调用钩子注册回调函数的接口最终都调用 nf_hook_thresh()函数，因此先看这个函数的定义：

```
static inline int nf_hook_thresh(u_int8_t pf, unsigned int hook,struct sock *sk,struct sk_buff *skb,
                                 struct net_device *indev,struct net_device *outdev,
                                 int (*okfn)(struct sock *, struct sk_buff *),int thresh)
/*thresh：优先级阀值，优先级数值小于此值（优先级更高）的回调函数不调用，
*okn()：调用完所有该调用的回调函数后，返回值为 1 时，继续调用的函数*/
{
    if (nf_hooks_active(pf, hook)) {       /*钩子处注册了回调函数*/
        struct nf_hook_state state;       /*nf_hook_state 实例*/

        nf_hook_state_init(&state, &nf_hooks[pf][hook], hook, thresh,pf, indev, outdev, sk, okfn);
                                                        /*由函数参数设置 nf_hook_state 实例*/
        return nf_hook_slow(skb, &state);       /*依次调用注册的回调函数，/net/netfilter/core.c*/
    }
    return 1;        /*钩子处没有注册回调函数，直接返回 1*/
}
```

如果 hook 钩子处没有注册回调函数，nf_hook_thresh()函数直接返回 1，如果注册了回调函数将调用函数 nf_hook_slow()依次调用钩子处的回调函数。

nf_hook_slow()函数遍历 hook 钩子对应双链表中优先级数值大于等于 thresh 的 nf_hook_ops 实例（优先级从高到低），调用其中的回调函数。如果某一回调函数返回值不为 NF_ACCEPT，将不再调用此回调函数之后的回调函数，nf_hook_slow()函数依此回调函数的返回值确认返回值。如果最后一个回调函数之前的所有回调函数返回值都是 NF_ACCEPT，nf_hook_slow()函数将依据最后一个回调函数的返回值确认返回

值。nf_hook_slow()函数返回 1（回调函数返回值为 NF_ACCEPT 或 NF_STOP）表示数据包继续执行正常的处理流程。

nf_hook_thresh()函数返回 1，表示调用者需要调用 okfn()函数继续处理数据包。注意在 nf_hook_thresh()函数中并没有调用 okfn()函数。

下面看一下调用钩子注册回调函数的各接口函数定义：

（1）**nf_hook()**：只调用回调函数，定义如下：

```
static inline int nf_hook(u_int8_t pf, unsigned int hook, struct sock *sk,
                          struct sk_buff *skb, struct net_device *indev,struct net_device *outdev,
                          int (*okfn)(struct sock *, struct sk_buff *))
{
    return nf_hook_thresh(pf, hook, sk, skb, indev, outdev, okfn, INT_MIN);
}
```

（2）**NF_HOOK_THRESH()**：调用优先级数值大于等于 **thresh** 的所有回调函数，最后返回 1 后，再调用 **okfn()**函数，定义如下：

```
static inline int NF_HOOK_THRESH(uint8_t pf, unsigned int hook, struct sock *sk,
                                 struct sk_buff *skb, struct net_device *in,struct net_device *out,
                                 int (*okfn)(struct sock *, struct sk_buff *), int thresh)
/*pf：协议类型，hook：钩子编号，okfn：回调函数最终返回 1（NF_ACCEPT）后调用的函数*/
{
    int ret = nf_hook_thresh(pf, hook, sk, skb, in, out, okfn, thresh);
    if (ret == 1)
        ret = okfn(sk, skb);        /*调用 okfn()函数*/
    return ret;
}
```

（3）**NF_HOOK()**：调用钩子注册的所有回调函数，最后返回 1 后，再调用 **okfn()**函数，定义如下：

```
static inline int NF_HOOK(uint8_t pf, unsigned int hook, struct sock *sk, struct sk_buff *skb,
    struct net_device *in, struct net_device *out,int (*okfn)(struct sock *, struct sk_buff *))
{
    return NF_HOOK_THRESH(pf, hook, sk, skb, in, out, okfn, INT_MIN);
                                        /*优先级阀值设为最小值，调用钩子注册的所有回调函数*/
}
```

（4）**NF_HOOK_COND()**：指定条件为假将不调用回调函数，直接调用 okfn()函数；指定条件为真，将调用回调函数，最后返回 1 后，将调用 okfn()函数，定义如下：

```
static inline int NF_HOOK_COND(uint8_t pf, unsigned int hook, struct sock *sk,
                               struct sk_buff *skb, struct net_device *in, struct net_device *out,
                               int (*okfn)(struct sock *, struct sk_buff *), bool cond)
/*cond：指定条件*/
{
    int ret;

    if (!cond ||((ret = nf_hook_thresh(pf, hook, sk, skb, in, out, okfn, INT_MIN)) == 1))
        ret = okfn(sk, skb);
```

```
    return ret;
}
```

内核支持 Netfilter 子系统需选择 NETFILTER 配置选项，如果没有选择此选项，nf_hook()函数返回 1，NF_HOOK()、NF_HOOK_COND()等宏直接调用 okfn()函数。

3 网络层用户接口

用户可通过用户空间工具配置、获取网络层、网络接口参数。有两个用于控制 TCP/IP 联网和处理网络设备的包：net-tools 和 iproute2。

iproute2 包包含如下命令：

●**ip**：用于管理网络路由选择表和网络接口等。

●**tc**：用于流量控制管理。

●**ss**：用于转储套接字统计信息。

●**lnstat**：用于转储 Linux 网络统计信息。

●**bridge**：用于管理网桥地址和设备。

iproute2 包主要基于通过 netlink 套接字从用户空间向内核发送请求并获取应答，但也存在一些使用 ioctl()系统调用的例外情况。

net-tool 包基于 ioctl()系统调用，包含以下著名的命令：

●**ifconfig**

●**arp**

●**route**

●**netstat**

●**hostname**

●**rarp**

iproute2 包的一些高级功能在 net-tool 包中并没有。

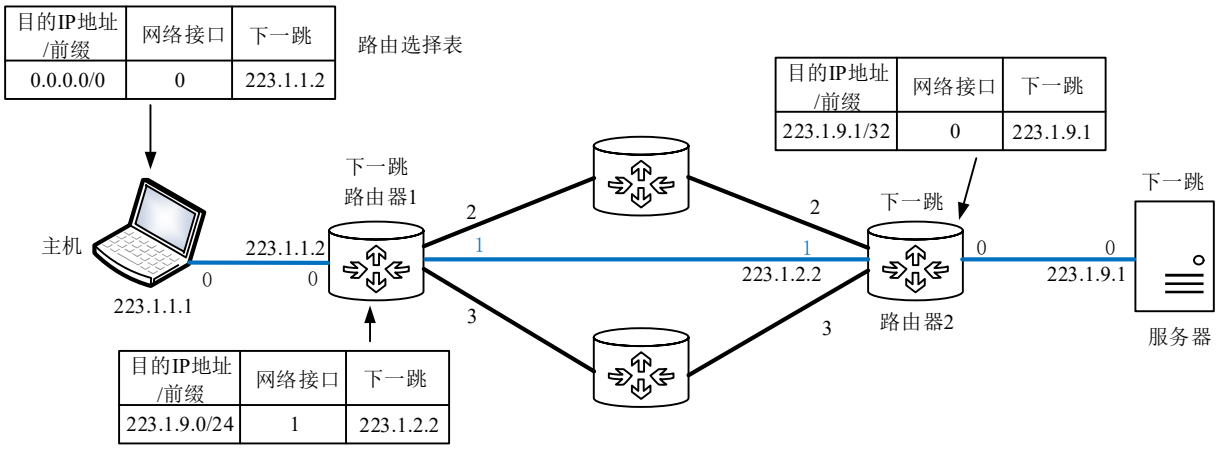
13.1.2 路由选择子系统

在网络层协议发送和接收数据包流程中，首先要进行的都是路由选择。路由选择根据目的 IP 地址确定数据包下一步的走向，如是投递给本机，外发（或转发），是单播还是组播等。路由选择结果中包含数据包下一步的处理函数，发送和接收数据包流程执行完路由选择后，调用路由选择结果中的处理函数处理数据包。

1 概述

因特网中的主机和路由器具有一个或多个网络接口，路由器之间相互连接，组成一个互联的网络。主机和路由器中的每个网络接口具有至少一个 IP 地址，路由器需要根据数据包中的目的 IP 地址将数据包接力传递，最终送达至目的主机。

如下图所示，假设主机（笔记本电脑）需要向服务器发送数据包，主机到服务器具有多条路径，经路由器 1 和路由器 2 到达服务器是最短的路径。那么，网络中的路由器如何根据数据包目的 IP 地址选择最短的路径呢？



网络中各节点网络层协议中维护着一个或多个路由选择表，表项指示了发往某个目的地址（地址范围）的数据包由节点哪个网络接口发出。网络层协议通过路由选择表来引导数据包的走向。

路由选择表项中至少需要包含以下信息：

●**目的 IP 地址**：32 位的目的 IP 地址。

●**掩码**：32 位字段，前若干个连续的比特位为 1，剩下比特位为 0，掩码与目的 IP 地址配合，用于确定一个目的地址范围。

●**网络接口**：目的 IP 地址与此表项匹配的数据包，通过此网络接口发送。

●**下一跳 IP 地址**：与接口指示网络接口相连（通信）的下一个网络节点中网络接口的 IP 地址。

掩码和目的 IP 地址按位相与的结果确定了一个目的地址范围，假设某个表项中目的 IP 地址为 223.1.9.2，掩码为 255.255.255.0，相与的结果为 223.1.9.0，所确定的地址范围为 223.1.9.0-223.1.9.255，即在此范围内的目的 IP 地址都与此表项匹配。发往此地址范围的数据包，由表项中指定的网络接口发出。

地址范围 223.1.9.0-223.1.9.255 可写成 223.1.9.0/24，24 表示 IP 地址前 24 位是固定的，称为前缀。

如上图中所示，主机路由选择表中存在一个表项，表项中目的 IP 地址为 0.0.0.0/0，网络接口为 0，下一跳 IP 地址为 223.1.1.2（路由器 1 网络接口 0），这是一个默认表项，可匹配任意 IP 地址。因为主机只有一个网络接口，外发的数据包只能通过此网络接口发出。

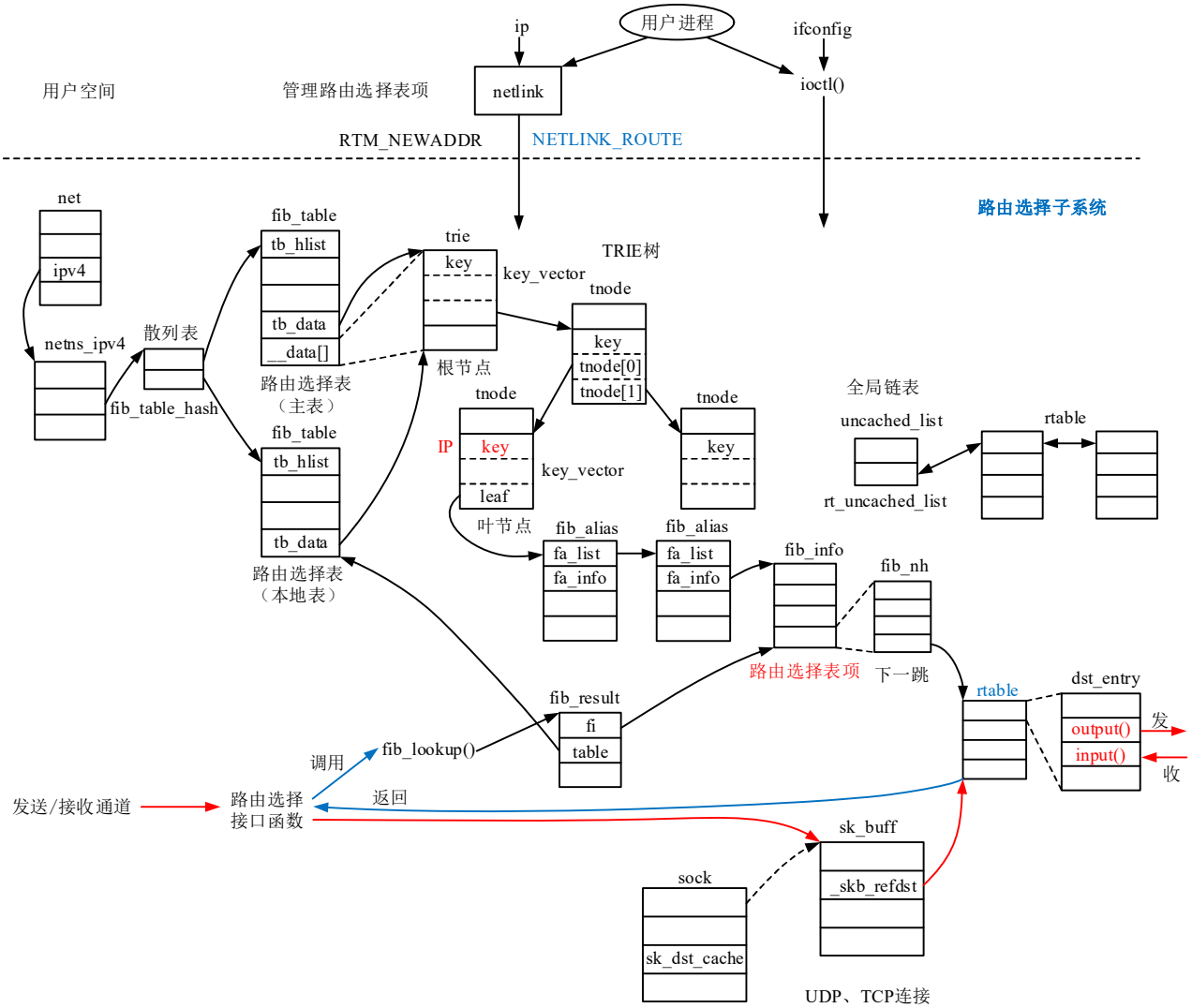
路由器 1 路由选择表中存在一个表项，将目的 IP 地址为 223.1.9.0/24 的数据包导向网络接口 1。路由器 2 路由选择表中存在一个表项，将目的 IP 地址为 223.1.9.1 的数据包导向网络接口 0，通过两个路由器的接力，数据包最终到达服务器。

由上可知，网络各节点中的路由选择表，决定了数据包所经过的网络路径。

网络节点中的路由选择表在网络协议初始化过程中创建，也可以由用户指令创建（需支持策略路由）。用户可通过 ip、ifconfig 命令等，添加、删除、获取路由选择表项。也可以在用户空间运行路由选择协议守护进程，由用户进程通过某种算法设置路由选择表，本书暂不涉及路由选择协议。

网络层协议在发送/接收数据包时，以数据包目的 IP 地址执行路由选择，在路由选择表中查找匹配的表项，以确定数据包的走向，对于外发的数据包需确定由哪个网络接口发出，以及下一跳的 IP 地址。

IPv4 网络层协议中路由选择子系统如下图所示：



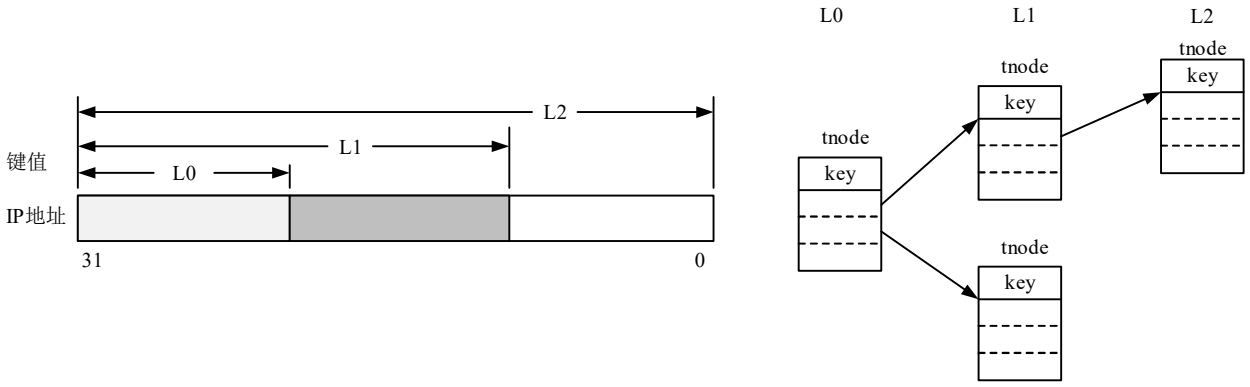
在 IPv4 路由选择子系统中，路由选择表由 fib_table 结构体表示，路由选择表属于网络命名空间的资源。内核在初始化阶段为初始网络命名空间创建了两个路由选择表，主表和本地表（共用路由选择表项）。新建网络命名空间时，也会为其创建初始的主表和本地表。

路由选择表项由 fib_info 结构体表示，包含表项的许多具体信息。fib_info 结构体中嵌入 fib_nh 结构体（数组）表示下一跳信息，包含输出网络接口编号、下一跳 IP 地址等。

路由选择表项 fib_info 实例由 TRIE 树管理，树中节点由 tnode 结构体表示。fib_info 实例由 TRIE 树叶节点管理，叶节点中 leaf 链表链接的是 fib_alias 实例，此实例关联到 fib_info 实例。fib_alias 结构体表示路由选择表项别名，目的 IP 地址、输出网络接口、下一跳 IP 地址等参数相同，只是个别参数不同的路由选择表项，可共用 fib_info 实例，而将不同的参数保存在 fib_alias 实例中。

路由选择表项中的目的 IP 地址保存在叶节点键值 key 成员中。叶节点管理目的 IP 地址相同的所有路由选择表项，目的 IP 地址前缀（掩码）的信息保存在 fib_alias 实例中。TRIE 树中其它节点键值 key 保存的是目的 IP 地址的部分值（分段）。

TRIE 树可视为基数树的变种，树中节点组成一个树状的层次结构。树中节点键值 key 是目的 IP 地址的部分或全部值，如下图所示。假设树中共 3 级（从高到低为 L0-L2），低一级节点（子节点）中键值是其上级节点键值的累积。以目的 IP 地址查找匹配表项时，在树中从高到低依次比较各级节点键值与 IP 地址值（只比较键值部分），找到键值匹配长度最长的叶节点中的路由表项，此表项为目的 IP 地址匹配的表项。TRIE 树依此实现路由选择查找中最长前缀匹配算法。



路由选择子系统返回给调用者的路由选择结果由 rtable 结构体表示，其内嵌的 dst_entry 结构体成员指针将赋予套接字 sock 实例或数据包 sk_buff 实例。

在执行套接字连接操作（connect()系统调用）时，将执行路由选择。路由选择接口函数调用 fib_lookup()函数在路由选择表中查找匹配的表项，fib_lookup()函数返回结果由 fib_result 结构体表示，包含查找到的路由选择表、表项等信息。路由选择接口函数然后依 fib_result 结果查找表项中下一跳缓存的 rtable 实例，或创建并设置新 rtable 实例。新实例缓存到下一跳 fib_nh 实例中，或缓存到全局双链表。路由选择接口函数将 rtable 实例返回给调用者。在连接操作中，路由选择结果 rtable 实例 dst_entry 结构体成员，将赋予 sock 实例。

在发送数据包路径中，如果套接字已经建立连接，sock 实例关联的 dst_entry 实例将赋予 sk_buff 实例，如果没有建立连接，将执行路由选择，返回结果 dst_entry 实例成员将赋予 sk_buff 实例。

在接收数据包路径中，在 IPv4 协议接收数据包 ip_rcv()函数中，将执行路由选择，返回结果 dst_entry 实例成员将赋予 sk_buff 实例。

在发送路径中将调用 sk_buff 实例关联 dst_entry 实例中的 **output()**函数发送数据包，在接收路径中将调用 sk_buff 实例关联 dst_entry 实例中的 **input()**函数接收数据包。

在路由选择接口函数中将对 dst_entry 实例中 output()和 input()函数指针赋值。发送路径路由选择中，dst_entry 实例 ouput()函数赋值如下：

dst_entry.output=**ip_output()**　　/*发送单播数据包*/

dst_entry.output=**ip_mc_output()**　/*发送组播或广播数据包*/

接收路径路由选择中，dst_entry 实例 input()函数赋值如下：

dst_entry.input=**ip_local_deliver()**　　/*所有投递到本机的数据包，含单播、组播、广播数据包*/

dst_entry.input=**ip_forward()**　　/*接收转发数据包*/

dst_entry.input=ip_mr_input()　　/*组播路由器转发组播数据包*/

对于需要外发的数据包，ip_output()、ip_forward()和 ip_mc_output()函数内最终调用 ip_finish_output()函数发送数据包。ip_finish_output()函数将根据路由选择结果中指示的输出网络接口编号和下一跳 IP 地址，在邻居子系统中查找邻居 neighbour 实例（不存在时创建），如果邻居中已缓存了邻居物理地址（或 L2 层报头），则对数据包写入 L2 层报头，发往数据链路层。如果邻居物理地址尚未解析将调用邻居实例中的 output()函数，先解析邻居物理地址，然后再发送数据包。邻居子系统将在下一节中介绍。

对于投递到本机的数据包，ip_local_deliver()函数调用传输层协议注册的 net_protocol 实例中的处理函数，将数据包交由传输层协议处理。

内核在初始化阶段创建的路由选择表是空表，需要用户通过 ip、ifconfig 等命令添加（删除、查询）路由选择表项。ip 命令通过 NETLINK_ROUTE 类型 netlink 套接字添加路由选择表项，ifconfig 命令通过 ioctl()系统调用添加路由选择表项。另外，用户还需要设置本机网络接口的本地 IP 地址，本地 IP 地址也会以表项形式添加到路由选择表中。

IPv4 路由选择子系统实现代码位于/net/ipv4/目录下，主要包含以下文件：

●**route.c**：主要实现路由选择子系统对网络层协议的接口、路由选择子系统初始化等。

●**fib_frontend**：实现路由选择表的初始化，实现与用户的接口等。

●**fib_semantics.c**：实现路由选择表项 fib_info 实例的管理、操作等。

●**fib_tries.c**：实现路由选择表最长匹配前缀查找算法（TRIE 树），实现路由选择表项的管理和查找等。

●**devinet.c**：实现网络设备在网络层协议中的表示，本机 IP 地址管理等。

●**fib_rules.c**：用于实现策略路由选择（本小节暂不涉及）。

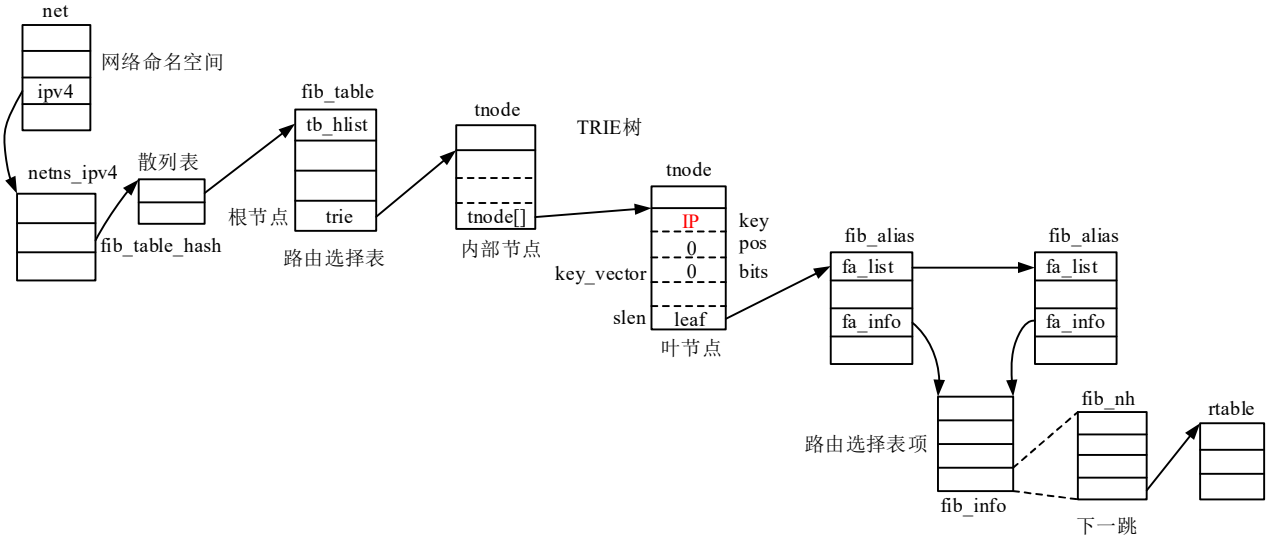
2 路由选择表

路由选择表是属于网络命名空间下的资源，也就是说每个网络命名空间有自己的路由选择表。在内核初始化阶段将为初始网络命名空间创建本地表和主表，新建网络命名空间时，将为其创建主表和本地表。

■路由选择表结构

路由选择表由 fib_table 结构体表示，路由选择表项（条目、路由）由 fib_info 结构体表示，fib_alias 表示路由选择表项别名，关联到路由选择表项。路由选择表 fib_table 实例由网络命名空间中的散列表管理，也就是说路由选择表属于特定的网络命名空间。

fib_table 路由选择表中嵌入 TRIE 树结构用于管理路由选择表项。TRIE 树叶节点通过散列链表管理表项别名 fib_alias 实例，fib_alias 实例关联到 fib_info 实例。一个 TRIE 树叶节点管理目的 IP 地址相同，但参数不同的路由选择表项。参数不同的路由选择表项，某些有区别的参数保存在 fib_alias 实例中，而共用相同的 fib_info 实例，如下图所示。TRIE 树用于实现了路由选择查找中的最长前缀匹配算法，下文将详细介绍。



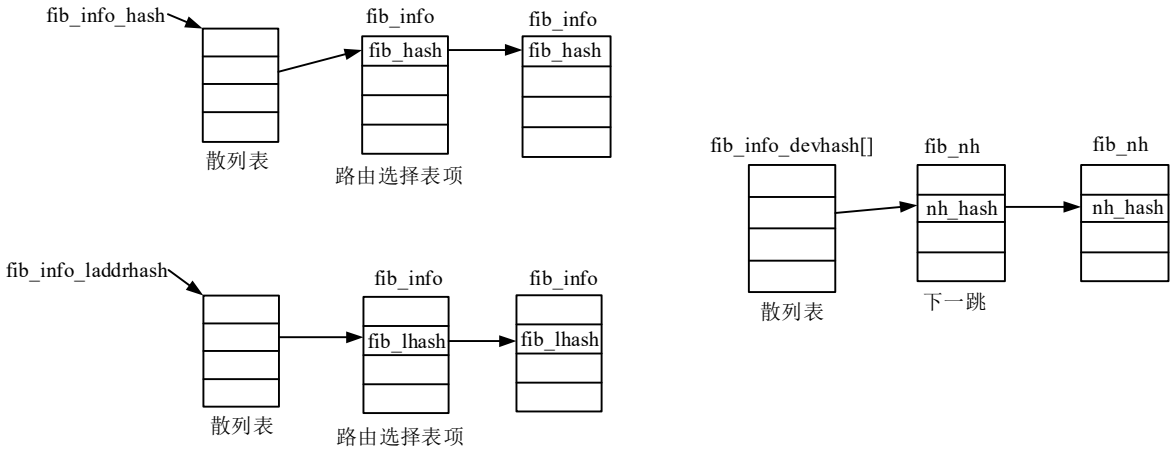
fib_info 结构体记录了路由选择表项的大部分信息，fib_info 结构体中嵌入 fib_nh 结构体成员，表示下一跳的信息。fib_nh 结构体保存的是从本机网络接口到一下跳网络接口的通道信息。如果内核配置了多路径路由选择，fib_info 实例最后嵌入的是 fib_nh 实例数组（需选择 IP_ROUTE_MULTIPATH 配置选项）。

内核在/net/ipv4/fib_semantics.c 文件内定义了全局散列表 fib_info_hash 和 fib_info_laddrhash 用于管理 fib_info 实例，定义了全局散列表 fib_info_devhash[]用于管理 fib_nh 实例，如下图所示：

static struct hlist_head ***fib_info_hash**;　　/*管理所有 fib_info 实例*/

static struct hlist_head ***fib_info_laddrhash**;　　/*管理设置了 fi->fib_prefsrc 成员的 fib_info 实例*/

static struct hlist_head **fib_info_devhash**[DEVINDEX_HASHSIZE];　　/*管理所有 fib_nh 实例，256 项*/



在创建表示路由选择表项 fib_info 实例的 fib_create_info()函数中，会将 fib_info 和 fib_nh 实例插入以上散列表。fib_info 实例插入 fib_info_hash 散列表的散列函数为 fib_info_hashfn(fi)，插入到 fib_info_laddrhash 散列表的散列函数为 fib_laddr_hashfn(fi->fib_prefsrc)（由 fib_prefsrc 计算散列值）。

fib_nh 实例插入 **fib_info_devhash**[]散列表的散列函数为 fib_devindex_hashfn(fib_nh->nh_dev->ifindex)，由输出网络设备编号计算散列值。

■数据结构

下面介绍表示路由选择表、表项、下一跳的数据结构。

●fib_table

fib_table 结构体表示路由选择表，结构体定义如下（/include/net/ip_fib.h）：

```
struct fib_table {
    struct hlist_node    tb_hlist;    /*散列链表节点成员，将表插入网络命名空间中散列表*/
    u32                  tb_id;       /*路由选择表标识（编号），内核中可有多个路由选择表*/
    int                  tb_num_default;    /*表示包含的默认路由选择表项*/
    struct rcu_head      rcu;
    unsigned long        *tb_data;    /*指向 TRIE 树根节点，用于管理路由表项*/
    unsigned long        __data[0];   /*为 TRIE 树根节点预留空间*/
};
```

fib_table 结构体主要成员简介如下：

●tb_hlist：散列链表节点成员，网络命名空间 net 结构体中 netns_ipv4 结构体成员包含散列表，用于管理网络命名空间内的路由选择表，此节点成员用于将实例添加到管理散列表。

●tb_id：路由选择表标识，编号，主表编号为 254，本地表编号为 255。

●tb_data：指向管理路由选择表项 fib_info 实例的结构，即 TRIE 树根节点，此处为__data[]成员。

●__data[0]：为路由选择表项 fib_info 实例管理结构预留空间，在分配路由选择表时，一并为管理数据结构分配空间（TRIE 树根节点），如果路由选择表与其它表共用 TRIE 树，可不分配空间，而令 tb_data 成员指向其它路由选择表中__data[0]成员（TRIE 树根节点）。

在内核初始化阶段将为初始网络命名空间创建初始的路由选择表，即主表和本地表。在创建新网络命名空间时，将会为新网络命名空间创建初始的路由选择表（主表和本地表），详见下文。

●fib_info

路由选择表项由 fib_info 结构体表示，用户可通过 ip 等用户工具添加路由选择表项，用户设置的路由选择表项信息将保存在 fib_info 实例中。

fib_info 结构体定义如下（/include/net/ip_fib.h）：

```
struct fib_info {
    struct hlist_node    fib_hash;    /*将实例链入 fib_info_hash 散列表*/
    struct hlist_node    fib_lhash;
                         /*将实例链入 fib_info_laddrhash 散列表（实例指定了 fib_prefsrc 值）*/
    struct net       *fib_net;       /*指向所属网络命名空间*/
    int              fib_treeref;    /*引用计数*/
    atomic_t         fib_clntref;    /*引用计数*/
    unsigned int         fib_flags;    /*标志位*/
    unsigned char        fib_dead;     /*标志，指示是否可将 fib_info 对象释放*/
    unsigned char        fib_protocol;    /*路由选择协议标识符，如 RTPROT_BOOT*/
    unsigned char        fib_scope;    /*目标地址范围*/
    unsigned char        fib_type;     /*路由选择表项类型（目的地址类型）*/
    __be32               fib_prefsrc;    /*以源 IP 地址作为键值执行查找，设置此值*/
    u32              fib_priority;     /*优先级，默认为 0 表示最高，值越大优先级越低*/
    u32              *fib_metrics;    /*指向一个数组，存储了各种参数，很多参数与 TCP 相关*/
#define fib_mtu    fib_metrics[RTAX_MTU-1]          /*MTU 值*/
#define fib_window    fib_metrics[RTAX_WINDOW-1]
#define fib_rtt    fib_metrics[RTAX_RTT-1]         /*RTT 值*/
#define fib_advmss    fib_metrics[RTAX_ADVMSS-1]
    int              fib_nhs;    /*下一跳的数量，选择了多路径路由时此值大于 1，否则为 1*/
#ifdef CONFIG_IP_ROUTE_MULTIPATH    /*多路径路由选择，可有多个下一跳*/
    int              fib_power;
#endif
    struct rcu_head          rcu;
    struct fib_nh        fib_nh[0];      /*下一跳（数组）*/
#define fib_dev          fib_nh[0].nh_dev      /*将数据包传输到（第一个）下一跳的网络设备*/
};
```

fib_info 结构体主要成员简介如下：

●**fib_protocol**：路由选择协议标识符，取值定义在/include/uapi/linux/rtnetlink.h 头文件。从用户空间添加路由选择规则时，如果没有指定路由选择协议 ID，则默认设为 RTPROT_BOOT。管理员添加路由时，可能会使用修饰符 proto static，指出路由表项是由管理员添加的，此时设为 RTPROT_STATIC。

●**fib_scope**：指出了到达目的 IP 地址距离的类型，取值定义在/include/uapi/linux/rtnetlink.h 头文件：

```
enum rt_scope_t {
    RT_SCOPE_UNIVERSE=0,    /*地址可用于任何地方，最常见的情形*/
    RT_SCOPE_SITE=200,      /*仅用于 IPv6*/
    RT_SCOPE_LINK=253,      /*地址只能从直连主机访问*/
    RT_SCOPE_HOST=254,      /*目的地址是本机地址，如环回地址、本机地址*/
    RT_SCOPE_NOWHERE=255    /*目的地址不存在*/
};
```

●**fib_type**：路由选择表项（路由）类型，以确定数据包下一步的走向，取值定义如下：

```
enum {                  /*/include/uapi/linux/rtnetlink.h*/
    RTN_UNSPEC,
    RTN_UNICAST,        /* Gateway or direct route，网关或直连路由*/
    RTN_LOCAL,          /*本地路由，表示本机地址*/
    RTN_BROADCAST,    /* Accept locally as broadcast,send as broadcast，广播地址*/
```

```
    RTN_ANYCAST,        /* Accept locally as broadcast,but send as unicast */
    RTN_MULTICAST,          /* Multicast route，组播路由*/
    RTN_BLACKHOLE,          /* Drop，黑洞*/
    RTN_UNREACHABLE,    /*目的地址不可达*/
    RTN_PROHIBIT,       /*禁止特定的流量通过*/
    RTN_THROW,          /* Not in this table  */
    RTN_NAT,            /*Translate this address，转换地址*/
    RTN_XRESOLVE,       /* Use external resolver  */
    __RTN_MAX
};
```

●**fib_priority**：优先级，值越大优先级越低，默认为 0，表示优先级最高。

●**fib_metrics**：指向一个数组，保存各种参数，许多是用于 TCP 的参数。参数名称，即数组项索引值标识定义在/include/uapi/linux/rtnetlink.h 头文件：

```
enum {
    RTAX_UNSPEC,
    RTAX_LOCK,
    RTAX_MTU,       /*数组项保存 MTU 值*/
    RTAX_WINDOW,
    RTAX_RTT,
    RTAX_RTTVAR,
    RTAX_SSTHRESH,
    RTAX_CWND,      /*拥塞窗口值*/
    RTAX_ADVMSS,
    RTAX_REORDERING,
    RTAX_HOPLIMIT,
    RTAX_INITCWND,      /*初始拥塞窗口值*/
    RTAX_FEATURES,
    RTAX_RTO_MIN,
    RTAX_INITRWND,
    RTAX_QUICKACK,
    RTAX_CC_ALGO,
    __RTAX_MAX
};
```

在创建 fib_info 实例时 fib_metrics 初始化为指向 dst_default_metrics（空数组）。

●**fib_nh[0]**：fib_nh 结构体成员（数组），表示下一跳的信息，详见下文。使用多路径路由选择时，可在一条路由中指定多个下一跳。多路径路由是指发往表项中指示的目的地址（子网）的数据包，在本机有多个输出网络接口，每个网络接口对应一个下一跳。

●**fib_alias**

fib_alias 表示路由选择表项别名，结构体定义在/net/ipv4/fib_lookup.h 头文件内：

```
struct fib_alias {
    struct hlist_node   fa_list;    /*散列链表节点，将实例链接到 TRIE 树叶子节点*/
    struct fib_info     *fa_info;   /*指向 fa_info 实例*/
    u8              fa_tos;     /*服务类型*/
    u8              fa_type;    /*路由类型*/
    u8              fa_state;   /*状态标志*/
```

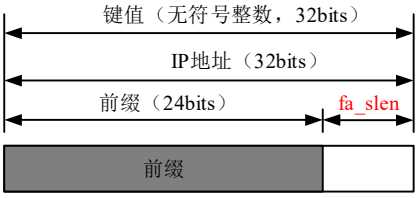
```
    u8              fa_slen;      /*键值比特位数减掩码中 1（前缀）的位数，即空闲不使用位数*/
    u32             tb_id;        /*路由选择表编号*/
    s16             fa_default;
    struct rcu_head     rcu;      /*释放 fa_alias 实例时的回调函数，回调函数中释放 fa_alias 实例*/
};
```

在有些情况下，会针对同一个目标地址或子网创建多个路由选择表项。这些路由选择表项可能唯一的区别就是其中的 TOS 不同。在这种情况下，将为每条路由创建一个 fib_alias 实例，而不是 fib_info 实例。fib_alias 实例用于存储前往同一个子网但参数不同的路由。多个 fib_alias 实例的 fa_info 指针可指向同一个 fib_info 实例，即共享同一个 fib_info 实例。

fib_alias 实例添加到 TRIE 树叶节点，查找路由选择表项时，先查找到 fib_alias 实例，然后查找其 fa_info 成员指向的 fib_info 实例。

这里需要说明一下的是 fa_slen 成员。在 TRIE 树的节点中键值 key 表示目的 IP 地址，它由无符号整数表示，其比特位数固定为 32 位，不管是 32 位系统还是 64 位系统。在内核中此常数由 KEYLENGTH 表示。fa_slen 成员值为键值 key 比特位数减 IP 地前缀比特位数（掩码中 1 的位数），也就是目地 IP 地址中除匹配前缀之外的比特位数量（表示主机号的比特位数）。

例如，如下图所示，无符号整型数长度为 32bits，IP 地址前缀为 24bits，则 fa_slen 值为 32-24=8。



在初始化路由选择表的 ip_fib_init()函数中将调用 **fib_trie_init()**函数为 fib_alias 结构体和 TRIE 树叶节点结构创建 slab 缓存。

●fib_nh

fib_nh 结构体表示下一跳的信息，表征的是本机某个网络接口到下一个节点（端系统或路由器）某个网络接口的通信通道，包含本机外出网络设备（接口）、网络设备编号、范围、下一跳 IP 地址等。

fib_nh 结构体定义如下（/include/net/ip_fib.h）：

```
struct fib_nh {
    struct net_device   *nh_dev;    /*外出网络设备 net_device 实例*/
    struct hlist_node   nh_hash;    /*散列表节点成员，将实例链入 fib_info_devhash 散列表*/
    struct fib_info     *nh_parent;  /*指向路由表项*/
    unsigned int        nh_flags;   /*标志*/
    unsigned char       nh_scope;   /*范围*/
#ifdef CONFIG_IP_ROUTE_MULTIPATH    /*支持多路径路由选择*/
    int             nh_weight;
    int             nh_power;
#endif
#ifdef CONFIG_IP_ROUTE_CLASSID
    __u32               nh_tclassid;
#endif
    int             nh_oif;     /*外出网络设备索引*/
    __be32          nh_gw;      /*网关 IP 地址，下一跳 IP 地址*/
    __be32          nh_saddr;   /*本地 IP 地址*
```

```
    int             nh_saddr_genid;     /*随机数*/
    struct rtable __rcu * __percpu *nh_pcpu_rth_output;
                        /*指向 rtable 指针数组，缓存发送通道路由选择中 rtable 实例*/
    struct rtable __rcu *nh_rth_input;      /*指向 rtable 实例，缓存接收通道 rtable 实例*/
    struct fnhe_hash_bucket __rcu *nh_exceptions;       /*下一跳例处*/
};
```

fib_nh 结构体主要成员简介如下：

●**nh_dev**：指向传输数据包到下一跳的本机网络设备 net_device 实例。

●**nh_oif**：外出网络设备编号。

●**nh_gw**：下一跳 IP 地址。

●**nh_flags**：标志，取值定义如下（/include/uapi/linux/rtnetlink.h）：

```
#define    RTNH_F_DEAD          1    /*下一跳不可用(used by multipath)    */
#define    RTNH_F_PERVASIVE     2    /* Do recursive gateway lookup    */
#define    RTNH_F_ONLINK        4    /* Gateway is forced on link*/
#define    RTNH_F_OFFLOAD       8    /* offloaded route */
#define    RTNH_F_LINKDOWN      16   /* carrier-down on nexthop */
```

●**nh_rth_input**：指向 rtable 实例。

●**nh_exceptions**：指向 fnhe_hash_bucket 实例，表示下一跳例外。

■ TRIE 树

路由选择表中通过 TRIE 树管理路由选择表项，TRIE 树中叶节点关联 fib_alias 实例，fib_alias 实例关联表示路由选择表项的 fib_info 实例。

在介绍 TRIE 树前，先简要介绍一下路由选择表项所包含的信息，以及表项匹配的规则。

假设路由器有 4 条链路（网络接口），编号从 0 至 3，数据包转发端口如下表所示：

目的 IP 地址范围	网络接口
11001000 00010111 00010000 00000000 至 11001000 00010111 00010111 11111111	0
11001000 00010111 00011000 00000000 至 11001000 00010111 00011000 11111111	1
11001000 00010111 00011000 00000000 至 11001000 00010111 00011111 11111111	2
其它	3

上表中每个表项指示了发往某段目的地址范围内（子网）的数据包从路由器中哪个网络接口转发出去。例如，发往 11001000 00010111 00011111 00111111 地址的数据包由网络接口 2 转发（匹配第 3 个表项）。

在这个路由器路由选择表中仅需设置以下 4 个表项：

（匹配前缀）目的地址	掩码	端口	下一跳 IP
11001000 00010111 00010...	11111111 11111111 11111...	0	x.x.x.x
11001000 00010111 00011000...	11111111 11111111 11111111...	1	x.x.x.x
11001000 00010111 00011...	11111111 11111111 11111...	2	x.x.x.x
0.0.0.0	255.255.255.255	3	x.x.x.x

上表每个表项中目的地址匹配前缀字段位数与掩码中置 1 的位数相同，省略的位都为 0。在查找数据包目的 IP 地址的匹配表项时，将目的 IP 地址与掩码按位相与，如果结果与表项中目的地址相等，则表示目的 IP 地址与此表项匹配，数据包将由表项中指示的网络接口发出。由此可知，所谓匹配就是目的 IP 地址的前若干位必须与表项中的匹配前缀（prefix）相同。

在上表中第 2 个表项表示的地址范围（11001000 00010111 00011000...）包含在第 3 个表项表示的地址

范围内（11001000 00010111 00011...），也就是说表项 2 是表项 3 的子集。若数据包目的 IP 地址在表项 2 所示的地址范围内，它将能同时匹配表项 2 和表项 3。在路由选择查找中，将返回匹配前缀最长的匹配项，也就是说对于在表项 2 范围内的目的 IP 地址将返回匹配表项 2，而不是表项 3，这称为**最长前缀匹配规则**。

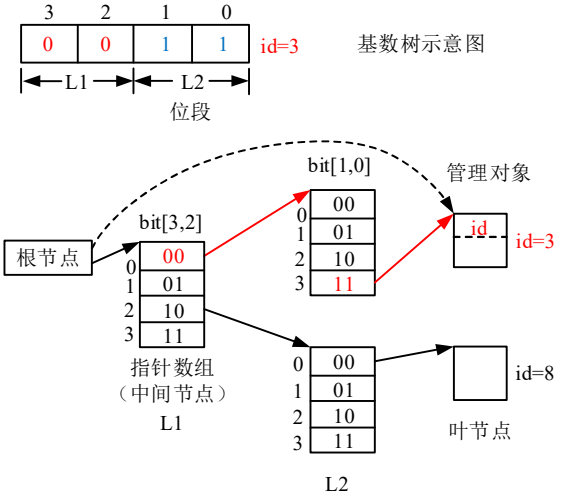
●引言

TRIE 是一种特殊的树（字典树），它代替了之前内核使用的 FIB 散列表和缓存，用于实现最长前缀匹配算法。TRIE 与第 1 章介绍过的基数树类似，我们先回顾一下基数树的原理。

基数树为它管理的每个对象分配一个由无符号整数表示的 id 值，用户可以通过此 id 值快速检索到 id 值代表的对象，基数树采用多级指针数组的结构来管理对象，对象只能位于叶节点。

下图示意了由 4 个比特位表示 id 值的基数树的结构，图中将 id 值的 4 个比特分成 2 个位段，分别是 bit[1，0]和 bit[3,2]，因此基数树有 2 级（L1~L2），每一级的节点为指针数组，指针数组项数为 2^{bits}，其中 bits 表示本级对应位段中比特位数量，即 L1 和 L2 级各节点数组项数都为 4。该级位段内二进制比特位表示的数值为数组的索引值。

基数树有一个根节点，根节点中包含一个指针成员，指向 L1 级的节点（指针数组）。L1 级的指针数组项指向 L2 级的节点（指针数组），L2 级（最低级）的指针数组项指向管理的对象（叶节点）。



基数树创建时只有根节点，当向其中添加管理对象时将创建从根节点至叶节点路径中的所有中间节点。例如，当向新创建的基数树添加 id 为 3 的对象时，将首先创建 L1 级的指针数组，以 bit[3,2]=0x00（0）为索引值，检查指针数组项是否关联了 L2 级指针数组，如果没有则创建，最后将对象关联到 L2 指针数组项，项数由 id 值 L2 位段表示的数确定，即 bit[1,0]=0x11（3）。

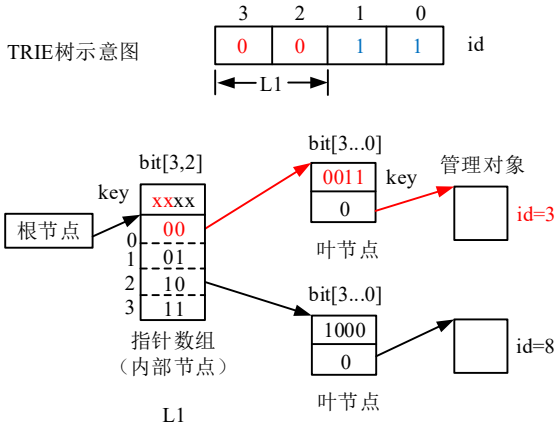
查找 id 值表示的对象时，将 id 值以按位段进行分段，以各位段表示的数值为索引值，从高到低依次检索树中指针数组，最低级的指针数组项指向的即是要查找的对象。

在基数树中，id 值位段的划分是固定的，并且位段的划分必须是连续的，不能有空洞。在基数树中每个对象有固定的位置，位置即代表了 id 值，查找到对象后不再需要检查对象的 id 值是否正确。

基数树也有缺点，如果对象 id 值比较稀疏（id 值不是按顺序分配的），id 值位段过细的划分会导致需要创建许多的中间节点，从而浪费内存并降低查找的速度。例如，假设在上面的基数树中只有一个 id 值为 3 的对象，也需要创建两个中间节点（L1、L2 级各 1 个）。如果不创建中间节点，直接将根节点指向 id 值为 3 的对象，则可以省略两个中间节点。在查找对象时，只要检查一下对象内包含的 id 值是否为要查找的 id 值即可。TRIE 树正是基于这种思想建立的。

TRIE 树可以认为是基数树的变种，TRIE 树中对 id 值位段的划分是不固定的（按需进行），位段的长度是不固定的，位段之间也可以不连续，可以有空洞。

TRIE 树中也有根节点，管理对象由叶节点管理，叶节点中只有一个指针成员，即指向管理对象。TRIE 树每个节点中增加了一个键值字段（key），用于检查时比对键值（id 值）。例如，下图是上面的基数树用 TRIE 树的表示：

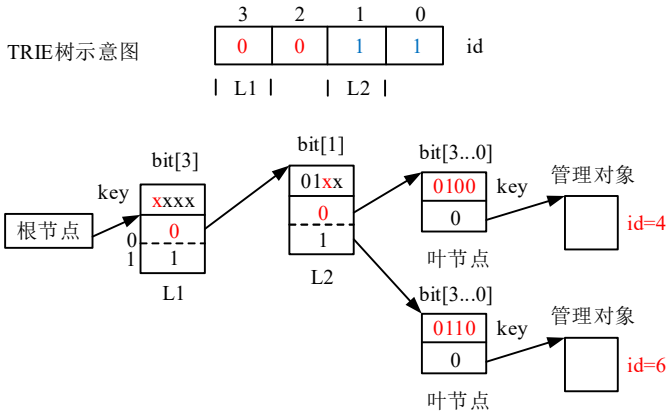


在上面的基数树中 L2 级指针数组中每个数组只有一个数组项有数据，因此可将此指针数组省略，将 L1 级中对应的指针数组项直接指向叶节点。如上图中所示，L1 级数组项 0 指向管理 id 为 3 的对象的叶节点，第 2 个数组项指管理 id 为 8 的对象的叶节点。

上图中 L1 级的位段依然为 bit[3,2]，但是 bit[1,0]位段没有使用（与基数树相同）。叶节点就不一样了，它没有使用位段，而是将管理对象整个 id 值作为键值，用于查找操作中比对 id 值。

查找对象时，首先用 id 值 bit[3,2]位段值检索 L1 级中指针数组项，指针项指向的是叶节点，再将 id 值与叶节点中键值（key 成员）比对，如果相等则其管理对象为需要查找的对象，如果不相等则说明查找的对象不存在。

TRIE 树中 id 值位段可以是不连续的，如下图所示：

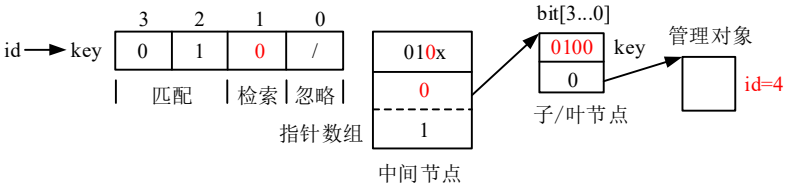


图中 L1 级位段为 bit[3]，L2 级位段为 bit[1]，而 bit[2]并未使用，因为现有对象中 id 值都是以 0x01 开头（前缀），而没有 0x00 开头的 id 值，因此可以不需要考虑 bit[2]（现有对象 bit[2]都为 0）。在查找对象时，到达 L2 级时，将 id 值与其节点键值比对，如果 id 值开头不为 0x01 则可以直接返回，因为不存在此 id 值的对象，如果 id 值开头为 0x01，再进行下一级的比对（查找）。

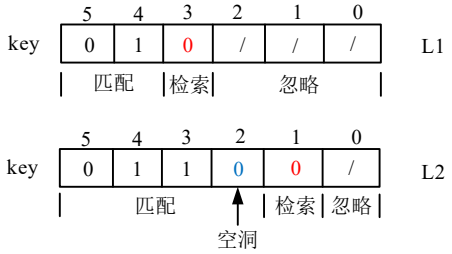
TRIE 树每个节点中保存的键值 key 可划分成 3 个段，分别是匹配段、检索段、忽略段，如下图所示。检索段就是基数树中的位段，用于检索本节点中的指针数组项，数组项指向下一级节点。在节点键值 key 中检索段值为 0，在查找操作中将 id 值的检索段与键值 key 中的检索段执行异或操作，得出的结果即是节点中指针数组项索引，用于查找下一个子节点。

忽略段是本级查找过程中忽略的段，不比对。匹配段是在查找操作中需要与 id 值匹配的位段，各级节点之间的匹配段是累积的，本级的匹配字段加上检索段（指针数组项索引值）将作为下一级（子节点）的匹配字段。如果下级检索字段与本级检索字段之间存在空洞，则空洞处值也将作为下一级的匹配段。

如下图所示，中间节点键值 bit[3,2]为匹配段，bit1 为检索段，bit0 为忽略段。查找操作中，到达此中间节点时，将 id 值与 key 值进行按位异或操作，异或结果右移忽略段的位数（即将忽略段的值去掉）。如果 id 与 key 值的匹配段异或结果为 0，表示 id 与 key 值匹配。检索段异或出的结果即用于检索指针数组项，查找下一级子节点。



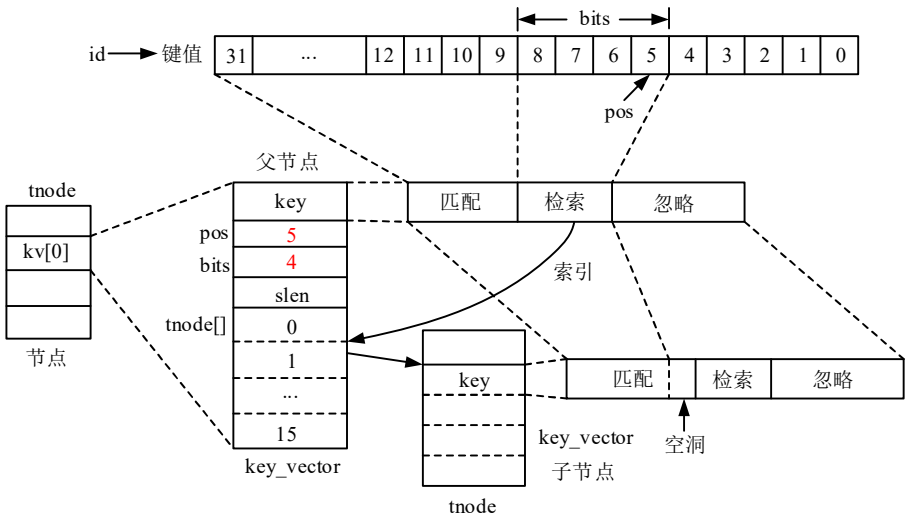
上面讲到相邻两级间的检索段之间可以有空洞。如下图所示，L1 级的检索段为 bit3，而 L2 级的检索段为 bit1，bit2 为空洞区（此处固定为 0），L1 级的匹配段为 bit[5...4]，L2 级的匹配段为 bit[5...2]。



TRIE 树中的节点，以及每个节点键值（含匹配段、检索段、忽略段）都是动态的，在向树中添加和删除对象时会动态改变。

●数据结构

下面看一下 Linux 内核中实现 TRIE 树的数据结构，相关数据结构都定义在/net/ipv4/fib_tries.c 文件内。下图示意了表示 TRIE 树节点的数据结构：



TRIE 树中根节点由 trie 结构体表示，结构体定义如下：

```
struct trie {            /*根节点，根节点下只有一个子节点*/
    struct key_vector kv[1];      /*key_vector 结构体成员*/
  #ifdef CONFIG_IP_FIB_TRIE_STATS
    struct trie_use_stats __percpu *stats;      /*统计信息*/
  #endif
};
```

TRIE 树中间节点和叶节点由 tnode 结构体表示，结构体定义如下：

```
struct tnode {
    struct rcu_head rcu;        /*链接需要释放的节点*/
    t_key empty_children;         /*数组项指针为 NULL 的数组项数量（子节点为 NULL）*/
```

```
    t_key full_children;              /* KEYLENGTH bits needed */
    struct key_vector __rcu *parent;      /*父节点（上一级节点）*/
    struct key_vector kv[1];      /*key_vector 结构体成员*/
    #define tn_bits    kv[0].bits      /*检索段比特位数*/
};
```

tnode 结构体主要成员简介如下：

●**parent**：指向父节点。

●**kv[1]**：key_vector 结构体成员，见下文。

由以上数据结构定义可知，TRIE 树所有节点中都包含一个 **key_vector** 结构体成员，它保存了节点的主要信息。

key_vector 结构体定义如下：

```
struct key_vector {
    t_key key;          /*节点键值，32 位无符号整型数*/
    unsigned char pos;        /*检索段起始比特位位置*/
    unsigned char bits;       /*检索段比特位数量*/
    unsigned char slen;       /*leaf 链接 fib_alias 实例（或子节点）中 fa_slen 值的最大值*/
    union {           /*指针数组，联合体*/
        struct hlist_head leaf;       /*叶子节点，包含一个散列链表头，管理键值相同的对象*/
        struct key_vector __rcu *tnode[0];  /*内部节点，子节点指针数组，数组项数 2bits*/
    };
};
```

key_vector 结构体主要成员简介如下：

●**key**：t_key 数据结构实例，即无符号正整数，表示节点键值。TRIE 树用于管理路由选择表项时，IP 地址将作为 id 值和节点键值。

●**pos、bits**：表示本节点键值检索段的起始比特位置和长度。假设某一节点键值检索段起始位置为 bit5，长度为 4 比特，则 pos 值为 5，bits 值为 4，子节点指针数组项数为 2^4=16。

内部节点 bits 值为非 0（大于 0），叶节点 bits 成员值为 0（pos 不一定为 0），表示没有子节点了，key_vector 结构体最后成员解释为一个散列链表头，用于链接管理对象。

●**tnode[0]**：节点不是叶节点时，tnode[]是一个指针数组，指向子节点 key_vector 实例，指针数组项数为 2^{bits}，在分配节点 tnode 结构体实例时将为指针数组分配合适的空间。

●**leaf**：节点为叶节点时，leaf 表示散列链表头，用于链接管理对象，如 fib_alias 实例。

●**slen**：如果节点为叶节点，leaf 链接的是 fib_alias 实例，slen 保存的是所有 fib_alias 实例中 fa_slen 成员值的最大值。fib_alias 实例在 leaf 链表中以 fa_slen 成员值升序（从小到大）排列。如果节点为内部节点，slen 值为其下所有子节点中 slen 值中的最大值。

目的 IP 地址相同，但匹配前缀长度不同的表项，由同一个 TRIE 树叶节点管理，表项 fib_info 实例通过 fib_alias 实例链接到 leaf 链表，fib_alias 实例以匹配前缀从大到小在 leaf 链表中排列，在查找匹配表项时，返回匹配前缀最长的表项。

●节点操作

路由选择表中管理路由表项的 TRIE 树根节点嵌入到 fib_table 实例中，在创建路由选择表时，将对内嵌的根节点初始化，主要是将 trie.kv[0].pos 和 trie.kv[0].slen 成员都设为 KEYLENGTH（32），trie.kv[0].bits 为 0。

下面列出几个主要的 TRIE 树节点操作函数，这些函数都定义在/net/ipv4/fib_tries.c 文件内：

●static inline unsigned long **get_index**(t_key key, struct key_vector *kv)：返回（key ^ kv->key）（异或）右移 kv->pos 位的结果。如果 key 与 kv 节点中键值匹配，返回值将小于 $2^{(kv->bits)}$，否则大于或等于此值。

也就是说 get_index()函数可检测 key 与 kv 节点中键值是否匹配，如果匹配返回值指针数组索引值，指向下一节点的。如是 kv 是叶节点，其 kv->pos 和 kv->bits 都为 0，key 需与 kv->key 完全相同才算匹配。

●static struct key_vector ***fib_find_node**(struct trie *t,struct key_vector **tp, u32 key)：在 TRIE 树中查找与键值 key 最长匹配的节点，t 指向树的根节点。如果最长匹配节点为叶节点，则返回叶节点中 key_vector 实例指针，参数*tp 指向此叶节点的父节点。如果最长匹配节点为内部节点，则函数返回 NULL，*tp 指向最长匹配节点中 key_vector 实例，在向 TRIE 树中添加新叶节点时，需要用到此指针。

●static int **fib_insert_node**(struct trie *t, struct key_vector *tp,struct fib_alias *new, t_key **key**)：如果 TRIE 树中尚不存在键值为 key 的叶节点，调用此函数创建并插入键值为 key 的新叶节点，new 指向叶节点管理对象，*tp 指向 fib_find_node()函数中查找到的 key 值最长匹配内部节点。

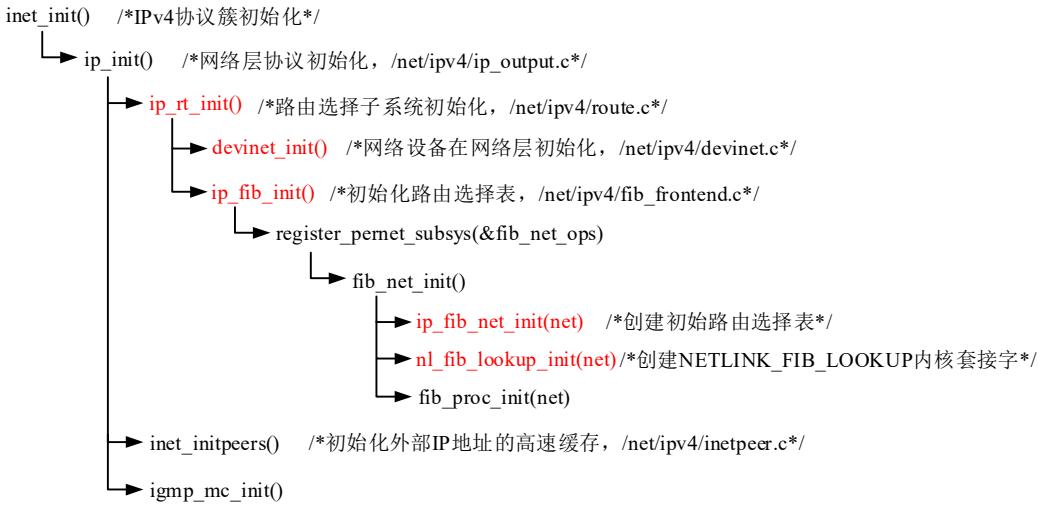
在插入新叶节点的过程中，可能需要创建新的内部节点。新内部节点作为*tp 指向节点的子节点，新叶节点又作为此新内部节点的子节点。如果不需要创建新内部节点，新叶节点将作为*tp 指向节点的子节点。最后，fib_insert_node()还需要调用 trie_rebalance()函数对 TRIE 树进行平衡操作，即合并或删除不必要的内部节点。

在添加路由选择表项时，需指定目的 IP 地址和匹配前缀长度，目的 IP 地址将作为 TRIE 节点的键值，创建 fib_alias 实例，调用 fib_insert_node()函数将 fib_alias 实例添加到 TRIE 树叶节点。

以上函数源代码请读者结合 TRIE 树原理，自行阅读。

3 路由选择子系统初始化

路由选择子系统初始化函数为 **ip_rt_init()**，在 IPv4 网络层协议初始化函数 ip_init()中被调用，函数调用关系如下图所示：



ip_init()函数内调用 **ip_rt_init()**函数完成路由选择子系统的初始化（详见下文），调用函数 inet_initpeers()初始化外部 IP 地址（下一跳）的高速缓存。

■路由初始化函数

下面介绍 ip_rt_init()函数的实现。ip_rt_init()定义在/net/ipv4/route.c 文件内，代码简列如下：

```
int __init ip_rt_init(void)
{
    int rc = 0;
    int cpu;

    ip_idents = kmalloc(IP_IDENTS_SZ * sizeof(*ip_idents), GFP_KERNEL);   /*分配原子变量数组*/
```

```
    ...
    prandom_bytes(ip_idents, IP_IDENTS_SZ * sizeof(*ip_idents));

    ip_tstamps = kcalloc(IP_IDENTS_SZ, sizeof(*ip_tstamps), GFP_KERNEL);   /*分配 u32 数组*/
    ...

    for_each_possible_cpu(cpu) {        /*为每个 CPU 创建 uncached_list 链表，缓存 rtable 实例*/
        struct uncached_list *ul = &per_cpu(rt_uncached_list, cpu);

        INIT_LIST_HEAD(&ul->head);
        spin_lock_init(&ul->lock);
    }
#ifdef CONFIG_IP_ROUTE_CLASSID
    ip_rt_acct = __alloc_percpu(256 * sizeof(struct ip_rt_acct), __alignof__(struct ip_rt_acct));
    if (!ip_rt_acct)
        panic("IP: failed to allocate ip_rt_acct\n");
#endif

    ipv4_dst_ops.kmem_cachep =kmem_cache_create("ip_dst_cache", sizeof(struct rtable), 0,
                                        SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
                                /*创建 rtable 结构体缓存，赋予 dst_ops 实例 ipv4_dst_ops*/

    ipv4_dst_blackhole_ops.kmem_cachep = ipv4_dst_ops.kmem_cachep;

    if (dst_entries_init(&ipv4_dst_ops) < 0)   /*初始化 ipv4_dst_ops 实例，/include/net/dst_ops.h*/
        panic("IP: failed to allocate ipv4_dst_ops counter\n");

    if (dst_entries_init(&ipv4_dst_blackhole_ops) < 0)     /*初始化 ipv4_dst_blackhole_ops 实例*/
        panic("IP: failed to allocate ipv4_dst_blackhole_ops counter\n");

    ipv4_dst_ops.gc_thresh = ~0;
    ip_rt_max_size = INT_MAX;

    devinet_init();   /*网络设备在网络层协议中的初始化，/net/ipv4/devinet.c*/
    ip_fib_init();   /*路由选择表初始化，/net/ipv4/fib_frontend.c*/

    if (ip_rt_proc_init())      /*路由选择子系统在 procfs 中的初始化，/net/ipv4/route.c*/
        pr_err("Unable to create route proc files\n");
#ifdef CONFIG_XFRM
    xfrm_init();
    xfrm4_init();
#endif
    rtnl_register(PF_INET, RTM_GETROUTE, inet_rtm_getroute, NULL, NULL);
                                        /*注册获取路由信息的 netlink 消息处理函数*/
#ifdef CONFIG_SYSCTL
    register_pernet_subsys(&sysctl_route_ops);
```

```
#endif
    register_pernet_subsys(&rt_genid_ops);
                                /*初始化 net->ipv4.dev_addr_genid 成员等，/net/ipv4/route.c*/
    register_pernet_subsys(&ipv4_inetpeer_ops);    /*初始化 net->ipv4.peers 成员，/net/ipv4/route.c*/
    return rc;
}
```

ip_rt_init()函数主要工作如下：

（1）为每个 CPU 创建 uncached_list 链表，用于缓存 rtable 实例。

（2）初始化 dst_ops 结构体实例 ipv4_dst_ops 和 ipv4_dst_blackhole_ops 实例，dst_ops 结构体是 dst_entry 结构体的操作函数，后面将介绍。

（3）调用 **devinet_init()**函数完成网络设备在网络层协议中的初始化，后面将介绍。

（3）调用 **ip_fib_init()**函数创建初始的路由选择表，详见下文。

（4）调用 ip_rt_proc_init()函数完成路由选择子系统在 procfs 中的初始化（创建文件）。

■初始化路由选择表

ip_rt_init()函数调用 **ip_fib_init()**函数初始化路由选择表，函数定义如下（/net/ipv4/fib_frontend.c）：

```
void __init ip_fib_init(void)
{
    /*注册添加、删除、获取路由消息的处理函数*/
    rtnl_register(PF_INET, RTM_NEWROUTE, inet_rtm_newroute, NULL, NULL);
    rtnl_register(PF_INET, RTM_DELROUTE, inet_rtm_delroute, NULL, NULL);
    rtnl_register(PF_INET, RTM_GETROUTE, NULL, inet_dump_fib, NULL);

    register_pernet_subsys(&fib_net_ops);    /*fib_net_ops 实例初始化函数中创建初始路由选择表*/
    register_netdevice_notifier(&fib_netdev_notifier);    /*向 netdev_chain 通知链注册通知*/
    register_inetaddr_notifier(&fib_inetaddr_notifier);
                                /*向 inetaddr_chain 通知链注册通知，/net/ipv4/devinet.c*/

    fib_trie_init();
        /*为 fib_alias 和 TRIE 树叶节点结构创建 slab 缓存，/net/ipv4/fib_trie.c*/
}
```

ip_fib_init()函数内完成的主要工作有：

●注册 NETLINK_ROUTE 类型 netlink 套接字添加、删除、获取路由消息的处理函数。

●注册 pernet_operations 结构体实例 **devinet_ops**，实例初始化函数中将创建初始的路由选择表，见下文。

●向原始通知链 **netdev_chain** 注册通知 fib_netdev_notifier 实例，通知回调函数完成发生设备事件时需要在路由选择子系统中执行的工作。如网络设备激活时，根据网络设备关联的 in_ifaddr 实例，添加本地路由选择表项。

●向 **inetaddr_chain** 通知链注册通知 fib_inetaddr_notifier 实例（/net/ipv4/fib_frontend.c），此通知回调函数主要是在路由选择表中为设备添加/删除本地地址路由选择表项。**inetaddr_chain** 通知链中通知表示在设备地址发生变化时需要执行的工作。

●完成 TRIE 树初始化工作，即为 fib_alias 和 TRIE 树叶节点结构创建 slab 缓存。

下面来看一下 fib_net_ops 实例的定义，其初始化函数中将为网络命名空间创建初始的路由选择表：

```
static struct pernet_operations fib_net_ops = {
```

```
    .init = fib_net_init,        /*初始化函数*/
    .exit = fib_net_exit,
};
```

fib_net_ops 实例初始化函数为 fib_net_init()，定义如下（/net/ipv4/fib_frontend.c）：

```
static int __net_init fib_net_init(struct net *net)
{
    int error;

#ifdef CONFIG_IP_ROUTE_CLASSID
    net->ipv4.fib_num_tclassid_users = 0;
#endif
    error = ip_fib_net_init(net);     /*创建初始路由选择表，/net/ipv4/fib_frontend.c*/
    ...
    error = nl_fib_lookup_init(net); /*创建 NETLINK_FIB_LOOKUP 套接字，/net/ipv4/fib_frontend.c*/
    ...
    error = fib_proc_init(net);
                        /*在 procfs 文件系统中创建文件，用于导出路由信息，/net/ipv4/fib_trie.c*/
    ...
out:
    return error;
    ...
}
```

fib_net_init()函数内调用 **ip_fib_net_init()**函数为网络命名空间创建初始的路由选择表，调用函数 nl_fib_lookup_init()为网络命名空间创建 NETLINK_FIB_LOOKUP 套接字，调用 fib_proc_init(net)函数在 procfs 文件系统中创建文件，用于导出路由信息。

●创建初始路由选择表

ip_fib_net_init()函数用于创建初始路由选择性表，代码如下：

```
static int __net_init ip_fib_net_init(struct net *net)
{
    int err;
    size_t size = sizeof(struct hlist_head) * FIB_TABLE_HASHSZ;
    /*FIB_TABLE_HASHSZ 为 256 或 2，取决于是否支持策略路由选择，/include/net/ip_fib.h*/

    size = max_t(size_t, size, L1_CACHE_BYTES);     /*缓存行对齐*/

    net->ipv4.fib_table_hash = kzalloc(size, GFP_KERNEL);
                                    /*为网络命名空间创建路由选择表散列表*/
    ...
    err = fib4_rules_init(net);     /*创建路由选择表，/net/ipv4/fib_frontend.c*/
    ...
    return 0;
    ...
}
```

ip_fib_net_init()函数首先为网络命名空间创建管理路由选择表的散列表，然后调用 fib4_rules_init()函

数创建初始路由选择表。

如果没有配置支持策略路由（没有选择 IP_MULTIPLE_TABLES 配置选项），则 fib4_rules_init()函数定义如下：

```
static int __net_init fib4_rules_init(struct net *net)
{
    struct fib_table *local_table, *main_table;     /*路由选择表指针*/

    main_table = fib_trie_table(RT_TABLE_MAIN, NULL); /*创建主路由选择表，/net/ipv4/fib_trie.c*/
    ...

    local_table = fib_trie_table(RT_TABLE_LOCAL, main_table);
                                /*创建本地路由选择表，指向主路由选择表 TRIE 树*/
    ...
    /*将主、本地路由选择表分别插入到主、本地路由选择表散列链表*/
    hlist_add_head_rcu(&local_table->tb_hlist,
                &net->ipv4.fib_table_hash[TABLE_LOCAL_INDEX]);
    hlist_add_head_rcu(&main_table->tb_hlist,
                &net->ipv4.fib_table_hash[TABLE_MAIN_INDEX]);
    return 0;
    ...
}
```

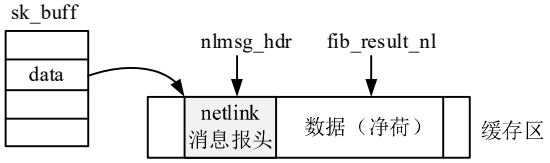
在没有选择支持策略路由时，内核创建两个路由选择表，一个是主表，一个是本地表，主表编号（id）为 254，本地表编号为 255（/include/uapi/linux/rtnetlink.h）。

fib_trie_table()函数用于创建路由选择表（创建 fib_table 实例），如果第二个参数为 NULL，则为路由选择表创建并初始化 TRIE 树实例，如果第二个参数不为 NULL，则其 tb_data 成员指向第二个参数表示的路由选择表中的 TRIE 树。新创建的路由选择表为空，没有路由选择表项。

由上面函数调用可知，内核创建了初始的主路由选择表和本地路由选择表，并且本地路由选择表复用主路由选择表的 TRIE 树，即共用主路由选择表项。

●NETLINK_FIB_LOOKUP 套接字

NETLINK_FIB_LOOKUP 协议类型的 netlink 套接字，用于用户查询路由选择表的表项信息。用户与内核套接字通过 fib_result_nl 结构体来传递信息，结构体实例嵌入到 netlink 消息净荷区（数据区），内核查找到信息后填充至实例，返回给用户套接字。消息格式如下图所示：



fib_result_nl 结构体定义如下（/include/net/ip_fib.h）：

```
struct fib_result_nl {
    /*以下是用户提供的参数*/
    __be32    fl_addr;     /* To be looked up*/
    u32       fl_mark;
    unsigned char     fl_tos;
    unsigned char     fl_scope;
    unsigned char     tb_id_in;
```

```
    /*以下是需要查找的参数（返回的参数）*/
    unsigned char     tb_id;          /*路由选择表编号*/
    unsigned char  prefixlen;       /*前缀长度*/
    unsigned char  nh_sel;          /*下一跳*/
    unsigned char  type;            /*路由类型*/
    unsigned char  scope;       /*范围*/
    int                err;          /*错误码*/
};
```

fib_net_init()函数内除了创建初始路由选择表外，还调用 nl_fib_lookup_init(net)函数创建协议类型为 NETLINK_FIB_LOOKUP 的 neltink 内核套接字，函数定义如下（/net/ipv4/fib_frontend.c）：

```
static int __net_init nl_fib_lookup_init(struct net *net)
{
    struct sock *sk;
    struct netlink_kernel_cfg cfg = {
        .input      = nl_fib_input,   /*用户消息处理函数*/
    };

    sk = netlink_kernel_create(net, NETLINK_FIB_LOOKUP, &cfg);   /*创建内核套接字*/
    if (!sk)
        return -EAFNOSUPPORT;
    net->ipv4.fibnl = sk;
    return 0;
}
```

用户消息处理函数为 nl_fib_input()定义如下（/net/ipv4/fib_frontend.c）：

```
static void nl_fib_input(struct sk_buff *skb)
{
    struct net *net;
    struct fib_result_nl *frn;     /*/include/net/ip_fib.h*/
    struct nlmsghdr *nlh;
    u32 portid;

    net = sock_net(skb->sk);
    nlh = nlmsg_hdr(skb);     /*netlink 消息报头*/
    if (skb->len < NLMSG_HDRLEN || skb->len < nlh->nlmsg_len ||nlmsg_len(nlh) < sizeof(*frn))
        return;

    skb = netlink_skb_clone(skb, GFP_KERNEL);
    if (!skb)
        return;
    nlh = nlmsg_hdr(skb);

    frn = (struct fib_result_nl *) nlmsg_data(nlh);
    nl_fib_lookup(net, frn);
                    /*查找路由选择表项消息，填充至 fib_result_nl 实例，/net/ipv4/fib_frontend.c*/
```

```
        portid = NETLINK_CB(skb).portid;          /* netlink portid */
        NETLINK_CB(skb).portid = 0;               /* from kernel */
        NETLINK_CB(skb).dst_group = 0;    /* unicast */
        netlink_unicast(net->ipv4.fibnl, skb, portid, MSG_DONTWAIT);    /*将 sk_buff 实例发还给用户*/
    }
```

nl_fib_lookup()函数负责查找路由选择表项，并将查找到信息填充消息中 fib_result_nl 实例，最后内核将查找结果发还给用户套接字。读者可学习完后面的路由查找操作，然后再去阅读 nl_fib_lookup()函数源代码。

4 管理路由选项表项

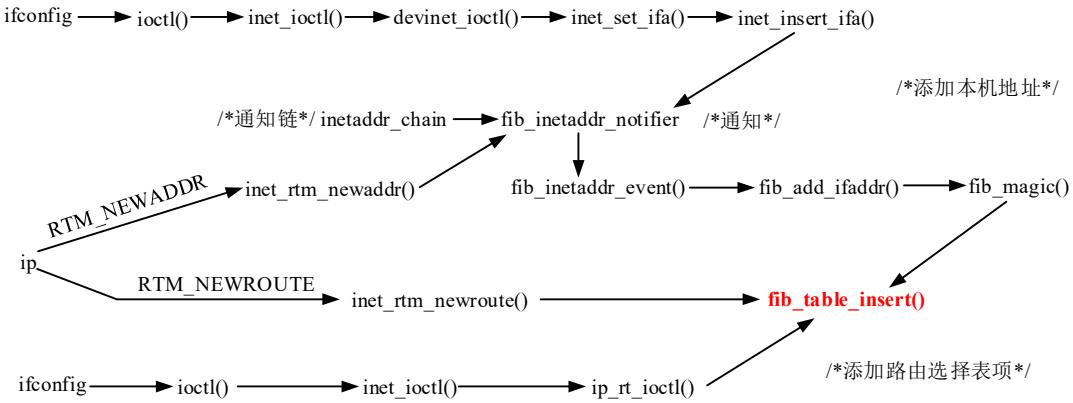
在前面介绍的初始化函数中，内核在初始化阶段创建了主表和本地表，这两个表都是空表，没有路由选择表项。

用户可以通过 ip、ifconfig 等用户空间工具设置、删除路由选择表项。

ip 命令通过 NETLINK_ROUTE 类型 netlink 套接字的 RTM_NEWROUTE 类型消息向内核传递添加路由的信息，通过 RTM_DELROUTE 消息删除路由消息，通过 RTM_GETROUTE 消息获取路由消息。在向网络设备配置本机地址时，也会向路由选择表添加表示本机地址的路由选择表项（环回地址）。

ifconfig 命令通过 ioctl()系统调用与内核通信，可用于管理路由选择表项、本机地址等。ip 命令比 ifconfig 命令更强大，并将取代 ifconfig 命令。

下图示意了添加路由选择表项的 4 个途径及函数调用关系（通过 ip、ifconfig 添加路由和本机地址）：



ip、ifconfig 命令可以直接向路由选择表添加路由选择表项。

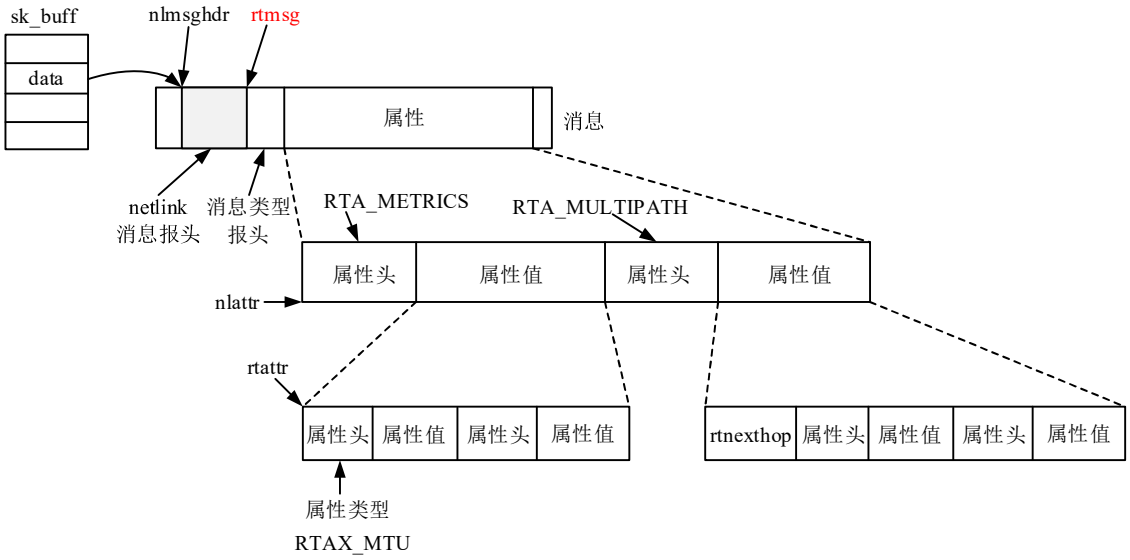
在 ip、ifconfig 命令添加本机 IP 地址时，在消息处理函数中将执行 inetaddr_chain 通知链中通知，其中 fib_inetaddr_notifier 通知的回调函数将会向路由选择表添加表项。

以上路径最终都通过 **fib_table_insert()**接口函数向路由选择表添加表项。

下面将以 RTM_NEWROUTE 消息添加路由选择表项为例，说明添加路由表项的实现，后面还将介绍 RTM_NEWADDR 消息中添加路由选择表项的实现。ifconfig 命令中添加路由表项的实现函数请读者自行阅读源代码。删除、查询路由选择表项的实现代码也请读者自行阅读。

■RTM_NEWROUTE 消息

RTM_NEWROUTE 消息通过 NETLINK_ROUTE 类型 netlink 套接字发送，套接字实现请参考 12.3.5 小节。下面直接看一下 RTM_NEWROUTE 消息格式，如下图所示。RTM_NEWROUTE 消息类型值保存在 netlink 消息报头中（**nlmsghdr.nlmsg_type**），RTM_NEWROUTE 消息类型报头由 rtmsg 结构体表示，在 netlink 消息报头之后。



●rtmsg

rtmsg 结构体表示 RTM_NEWROUTE 消息类型报头，定义如下（/include/uapi/linux/rtnetlink.h）：

```
struct rtmsg {
    unsigned char    rtm_family;    /*协议簇，必须是第一个成员*/
    unsigned char    rtm_dst_len;   /*掩码中 1 的位数，匹配前缀长度*/
    unsigned char    rtm_src_len;   /*源地址（本机端口地址）长度*/
    unsigned char    rtm_tos;       /*服务类型*/

    unsigned char    rtm_table;     /*路由选择表 id 值，若表不存在则创建*/
    unsigned char    rtm_protocol;  /*安装路由协议，路由由谁安装*/
    unsigned char    rtm_scope;     /*路由范围*/
    unsigned char    rtm_type;      /*路由类型（目的地址类型）*/

    unsigned         rtm_flags;     /*标记*/
};
```

rtmsg 结构体中部分成员同 fib_info 结构体中的成员，也就是说由用户设置路由选择表项参数：

◎rtm_type：路由（表项）类型，表示目的地址类型，取值由 fib_info.fib_type 成员。

◎rtm_protocol：安装路由（表项）协议类型，表示路由选择表项由谁安装的，取值同 fib_info.protocol 成员。

◎rtm_scope：范围，表示到目标地址的距离，取值同 fib_info.fib_scope 成员。

◎rtm_flags：标志，取值定义如下：

```
#define  RTM_F_NOTIFY      0x100    /* Notify user of route change*/
#define  RTM_F_CLONED      0x200    /* This route is cloned*/
#define  RTM_F_EQUALIZE    0x400    /* Multipath equalizer: NI*/
#define  RTM_F_PREFIX      0x800    /* Prefix addresses*/
```

●属性

RTM_NEWROUTE 消息属性区域有多个属性（通用的 netlink 属性）组成，属性类型定义如下：

```
enum rtattr_type_t {              /*/include/uapi/linux/rtnetlink.h*/
    RTA_UNSPEC,
    RTA_DST,        /*目的地址*/
```

```
    RTA_SRC,        /*本机地址（源地址）*/
    RTA_IIF,        /*输入网络设备编号*/
    RTA_OIF,        /*输出网络设备编号*/
    RTA_GATEWAY,    /*网关地址*/
    RTA_PRIORITY,   /*优先级*/
    RTA_PREFSRC,    /*路由表项中源 IP 地址*/
    RTA_METRICS,    /*参数数组*/
    RTA_MULTIPATH,  /*多路径路由选择*/
    RTA_PROTOINFO,  /*不再使用用*/
    RTA_FLOW,
    RTA_CACHEINFO,
    RTA_SESSION,    /*不再使用*/
    RTA_MP_ALGO,    /*不再使用*/
    RTA_TABLE,      /*路由选择表编号*/
    RTA_MARK,
    RTA_MFC_STATS,
    RTA_VIA,
    RTA_NEWDST,
    RTA_PREF,
    __RTA_MAX
};
```

以上属性的属性值里面可能不是简单的整数，而是由某个数据结构或子属性数组组成。例如，RTA_METRICS 属性的属性值里面是一串子属性。

RTA_METRICS 属性的子属性的属性头由 rtattr 结构体表示，定义如下：

```
struct rtattr {         /*/include/uapi/linux/rtnetlink.h*/
    unsigned short  rta_len;    /*子属性长度*/
    unsigned short  rta_type;   /*子属性类型*/
};
```

子属性 rta_type 属性类型定义如下（与 fib_info 结构体中 fib_metrics 成员指向的数组对应）：

```
enum {
    RTAX_UNSPEC,
    RTAX_LOCK,
    RTAX_MTU,
    RTAX_WINDOW,
    RTAX_RTT,
    RTAX_RTTVAR,
    RTAX_SSTHRESH,
    RTAX_CWND,
    RTAX_ADVMSS,
    RTAX_REORDERING,
    RTAX_HOPLIMIT,
    RTAX_INITCWND,      /*初始拥塞窗口*/
    RTAX_FEATURES,
    RTAX_RTO_MIN,
    RTAX_INITRWND,
```

```
    RTAX_QUICKACK,
    RTAX_CC_ALGO,
    __RTAX_MAX
};
```

又如，RTA_MULTIPATH 属性表示多路径路由（IP_ROUTE_MULTIPATH）信息，即下一跳信息。其属性值由一个队列组成，队列中的成员由 rtnexthop 结构体实例后接若干个属性组成，上图中只画出一个 rtnexthop 实例及其属性，一个 rtnexthop 结构体实例表示一个下一跳信息。在添加路由时，此属性值将用于设置 fib_info 实例中下一跳 fib_nh 数组实例。

rtnexthop 结构体定义在头文件/include/uapi/linux/rtnetlink.h，其中属性类型由枚举类型 rtattr_type_t 表示。

在设置路由表项时，假设目的地址为 10.0.0.0/25 网络，则目地 IP 地址为 10.0.0.0，匹配前缀长度为 25。

●消息处理函数

在路由选择表初始化函数 ip_fib_init()中注册了 RTM_NEWROUTE 消息的处理函数：

```
void __init ip_fib_init(void)
{
    rtnl_register(PF_INET, RTM_NEWROUTE, inet_rtm_newroute, NULL, NULL);
    ...
}
```

RTM_NEWROUTE 消息处理函数为 inet_rtm_newroute()，函数定义如下（/net/ipv4/fib_frontend.c）：

```
static int inet_rtm_newroute(struct sk_buff *skb, struct nlmsghdr *nlh)
/*skb：指向 sk_buff 实例，nlh：指向 netlink 消息报头*/
{
    struct net *net = sock_net(skb->sk);
    struct fib_config cfg;      /*fib_config 结构体实例*/
    struct fib_table *tb;
    int err;

    err = rtm_to_fib_config(net, skb, nlh, &cfg);
                        /*由消息填充 fib_config 实例，/net/ipv4/fib_frontend.c*/
    ...
    tb = fib_new_table(net, cfg.fc_table);     /*查找（或创建）路由选择表，/include/net/ip_fib.h*/
    ...
    err = fib_table_insert(tb, &cfg);     /*添加路由选择表项，/net/ipv4/fib_trie.c*/
errout:
    return err;
}
```

fib_config 结构体用于传递创建路由选择表项时所需的消息，结构体定义见下文。

inet_rtm_newroute()函数首先调用 rtm_to_fib_config()函数将消息转换成 fib_config 实例，然后调用函数 fib_new_table()查找或创建路由选择表（fib_config 实例中保存了路由选择表 id 值），在没有配置策略路由选项时（IP_MULTIPLE_TABLES），只查找本地路由选择表或主表，不创建新表。最后调用 **fib_table_insert()** 函数创建（或查找）fib_info 实例并添加到路由选择表和管理结构。

rtm_to_fib_config()与 fib_new_table()函数源代码请读者自行阅读，下面介绍 fib_table_insert()函数的实现。

■添加路由表项

在接口函数 fib_table_insert()中将通过 fib_config 结构体传递要创建路由表项的信息，此处 fib_config 实例中的信息来自 RTM_NEWROUTE 消息。

fib_config 结构体定义如下（/include/net/ip_fib.h）：

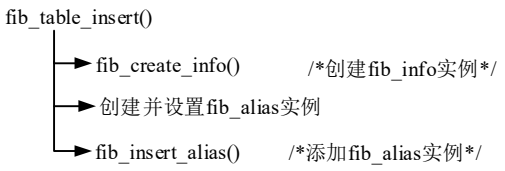
```
struct fib_config {
    u8          fc_dst_len;    /*目的地址前缀长度（掩码中 1 的位数），来处 rtmsg.rtm_dst_len*/
    u8          fc_tos;        /*服务类型，来自 rtmsg.rtm_tos*/
    u8          fc_protocol;   /*安装路由协议（路由表项由谁添加的），来自 rtmsg.rtm_protocol*/
    u8          fc_scope;      /*路由范围，来自 rtmsg.rtm_scope*/
    u8          fc_type;       /*路由类型，来自 rtmsg.rtm_type*/
    /* 3 bytes unused */
    u32         fc_table;      /*路由选择表编号，来自 RTA_TABLE 属性或 rtmsg 实例*/
    __be32      fc_dst;        /*目标 IP 地址，网络字节序，来自 RTA_DST 属性*/
    __be32      fc_gw;         /*下一跳 IP 地址，网络字节序，来自 RTA_GATEWAY 属性*/
    int         fc_oif;        /*外出网络设备编号，来自 RTA_OIF 属性*/
    u32         fc_flags;      /*路由标志，来自 rtmsg.rtm_flags*/
    u32         fc_priority;   /*优先级，来自 RTA_PRIORITY 属性*/
    __be32      fc_prefsrc;    /*本机源 IP 地址，网络字节序，来自 RTA_PREFSRC 属性*/
    struct nlattr       *fc_mx;     /*指向 RTA_METRICS 属性值中的第一个属性头实例*/
    struct rtnexthop    *fc_mp;     /*指向 RTA_MULTIPATH 属性值中第一个 rtnexthop 实例*/
    int         fc_mx_len;     /*RTA_METRICS 属性类型属性值的长度*/
    int         fc_mp_len;     /*RTA_MULTIPATH 属性类型属性值的长度*/
    u32         fc_flow;       /*来自 RTA_FLOW 属性*/
    u32         fc_nlflags;    /*netlink 标志，来处 netlink 消息报头 nlmsg_flags 成员*/
    struct nl_info      fc_nlinfo;     /*表示 netlink 消息信息*/
};
```

如果是通过 ip 命令添加路由选择表项，fib_config 结构体成员值提取自 RTM_NEWROUTE 消息中的 rtmsg 结构体实例（消息类型报头）或消息属性值。

fib_config 结构体最后的 fc_nlinfo 成员为 nl_info 结构体实例。nl_info 结构体保存 RTM_NEWROUTE 消息的相关信息，结构体定义在/include/net/netlink.h 头文件：

```
struct nl_info {
    struct nlmsghdr     *nlh;    /*指向 netlink 消息报头*/
    struct net      *nl_net;     /*指向网络命名空间*/
    u32             portid;      /*发送消息的用户套接字端口号*/
};
```

RTM_NEWROUTE 消息处理 inet_rtm_newroute()函数，在构建 fib_config 实例，并查找到添加表项的路由选择表后，调用 **fib_table_insert()**函数向路由选择表添加表项，函数调用关系简列如下图所示。



fib_table_insert()函数的主要工作是调用 **fib_create_info()**函数创建（或查找）路由选择表项 fib_info 实

例，然后将 fib_info 实例通过 fib_alias 实例添加到路由选择表 TRIE 树叶节点。

fib_create_info()函数根据 fib_config 参数创建 fib_info 实例，在 fib_info_hash 散列表查找是否存在相同参数的 fib_info 实例，如果存在则释放新创建的 fib_info 实例，返回原实例指针，否则返回新创建实例指针。**fib_insert_alias()**函数用于将创建的 fib_alias 实例（关联了 fib_info 实例）添加到 TRIE 树叶节点。

如果已存在相同的路由选择表项 fib_info 实例，可能需要替换原实例，或保留原实例，详见下面函数实现。

●fa_alias 实例管理

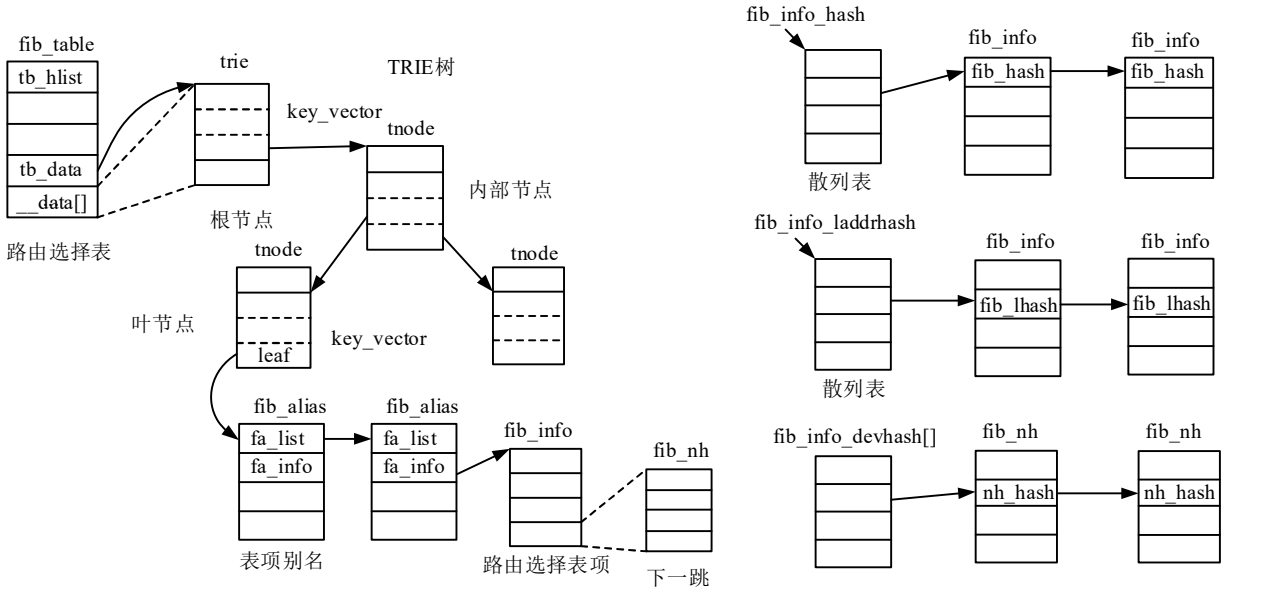
在讲解 fib_table_insert()函数代码前，我们先看一下 TRIE 树叶节点对 fa_alias 实例的管理。因为在插入路由选择表项前，需要查找是否已存在匹配的 fa_alias 实例。

fib_table_insert()函数在创建 fib_info 实例时，需要将实例插入管理散列表，并将其中下一跳实例也插入管理散列表。

fib_table_insert()函数在创建 fib_info 实例后，需要在 TRIE 树中查找是否已有匹配的 fib_alias 实例的存在，要根据查找结果确定是增加新 fib_alias 实例，还是替换原 fib_alias 实例，又或者是保留原匹配的实例。

每个 TRIE 叶节点管理目的 IP 地址相同的路由选择表项，表项按匹配前缀长度从大到小排列。也就是说，对于相同目的 IP 地址可以定义匹配前缀长度不同的表项。

下图示意了路由选择表的结构，以及 fib_info、fib_nh 实例管理散列表：



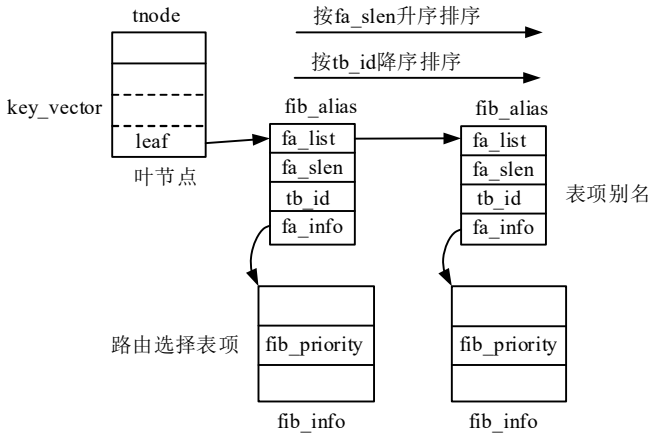
TRIE 树叶节点管理的 fib_alias 实例是 fib_info 别名，目的是为了让某些个别参数不同的表项可以共用 fib_info 实例，不同的参数保存在 fi_alias 实例中。

向 TRIE 树插入 fib_alias 实例的 **fib_insert_alias()**函数定义如下（/net/ipv4/fib_trie.c）：

```
static int fib_insert_alias(struct trie *t, struct key_vector *tp,struct key_vector *l, struct fib_alias *new,
                                                    struct fib_alias *fa, t_key key)
/*t：指向 TRIE 树根节点，new：需要添加的 fib_alias 实例，key：新叶节点键值（目的地址），
*tp、l：来自 fib_find_node()函数查找结果，l 不为 NULL 时，指向 key 匹配的叶节点，
*新 fib_alias 实例实例添加到此叶节点，tp 指向 l 的父节点；
*l 为 NULL 时，tp 指向键值 key 最长匹配的内部节点，没有匹配的叶节点。
*fa：若 l 为 NULL，则 fa 为 NULL；若 l 不为 NULL，fa 指向 l 叶节点 leaf 链表中某个 fib_alias 实例，
*新实例添加到 fa 前面。
*/
{
```

```
    if (!l)        /*l 为 NULL 表示最长匹配节点是内部节点，不是叶节点*/
        return fib_insert_node(t, tp, new, key);     /*/net/ipv4/fib_semantics.c*/
                /*创建新叶节点，使其关联 fib_alias 实例，并插入 TRIE 树*/

    /*存在与 key 匹配的叶节点*/
    if (fa) {        /*新实例插入到 fa 实例前面*/
        hlist_add_before_rcu(&new->fa_list, &fa->fa_list);
    } else {      /*fa 为 NULL，查找合适的位置，将新实例插入 l 叶节点 leaf 散列链表*/
        struct fib_alias *last;

        hlist_for_each_entry(last, &l->leaf, fa_list) {      /*遍历 leaf 链表，查找插入点*/
            if (new->fa_slen < last->fa_slen)        /*fib_alias 实例按 fa_slen 升序排列*/
                break;
            if ((new->fa_slen == last->fa_slen) &&(new->tb_id > last->tb_id))
                break;                                  /*fa_slen 相同，按 tb_id 降序排列*/
            fa = last;        /*本次遍历的 fa_alias 实例*/
        }

        if (fa)
            hlist_add_behind_rcu(&new->fa_list, &fa->fa_list);     /*添加到 fa 之后*/
        else
            hlist_add_head_rcu(&new->fa_list, &l->leaf);       /*添加到 leaf 链表头部*/
    }

    if (l->slen < new->fa_slen) {       /*slen：保存 fa_alias 实例中 fa_slen 成员最大值*/
        l->slen = new->fa_slen;
        leaf_push_suffix(tp, l);     /*更新 tp->slen 值（所有子节点中 slen 最大值），/net/ipv4/fib_trie.c*/
    }
    return 0;
}
```

由以上函数可知，fib_alias 实例在叶节点 leaf 散列链表中的排序规则如下：



leaf 散列链表 fib_alias 实例中首先按照 fa_slen 值从小到大（升序）排列（匹配前缀长度从大到小），相同 fa_slen 值的实例按 tb_id 值从大至小（降序）排列。

另外，若 fa_slen 和 tb_id 值相同，则按 fa_tos 降序排列，fa_slen、tb_id 以及 fa_tos 相同，则按 fib_priority 数值升序排列（优先级从高到低）。

●**接口函数**

向路由选择表插入表项的插口函数为 **fib_table_insert()**，定义如下（/net/ipv4/fib_trie.c）：

```
int fib_table_insert(struct fib_table *tb, struct fib_config *cfg)
/*tb：指向路由选择表，cfg：指向 fib_config 实例，传递路由表项参数*/
{
    struct trie *t = (struct trie *)tb->tb_data;     /*路由选择表关联的 TRIE 树根节点*/
    struct fib_alias *fa, *new_fa;
    struct key_vector *l, *tp;
    unsigned int nlflags = 0;
    struct fib_info *fi;        /*指向 fib_info*/
    u8 plen = cfg->fc_dst_len;          /*匹配前缀长度*/
    u8 slen = KEYLENGTH - plen;     /*TRIE 节点键值长度（比特位数）减 IP 地址前缀长度*/
    u8 tos = cfg->fc_tos;     /*服务类型*/
    u32 key;
    int err;
    ...
    key = ntohl(cfg->fc_dst);     /*目的 IP 地址作为键值，用于查找或建立 TRIE 树节点*/
    ...
    fi = fib_create_info(cfg); /*根据 fib_config 实例查找或创建 fib_info 实例，/net/ipv4/fib_semantics.c*/
    ...

    l = fib_find_node(t, &tp, key);       /*在 TRIE 树中查找最长匹配节点，/net/ipv4/fib_trie.c*/
    fa = l ? fib_find_alias(&l->leaf, slen, tos, fi->fib_priority,tb->tb_id) : NULL;     /*/net/ipv4/fib_trie.c*/
                    /*l 为 NULL 则 fa 为 NULL，
                    *l 不为 NULL 表示查找到了最长匹配 key 值的叶节点，然后调用 fib_find_alias()
                    *在此叶节点 leaf 链表中查找第一个同时满足下列条件的 fib_alias 实例，赋予 fa：
                    *（1）fa_slen 值等于 slen，并且 tb_id 值等于 tb->tb_id；
                    *（2）fa_info->fib_priority 大于等于 fi->fib_priority（且 fa->fa_tos 小于等于 tos），
                    *或者 fa->fa_tos 小于 tos。
                    *如果没有找到满足条件的实例，则 fa 设为 NULL。*/

    /*如果查找到满足上述条件的 fib_alias 实例*/
    if (fa && fa->fa_tos == tos &&fa->fa_info->fib_priority == fi->fib_priority) {     /*大的 if 语句开始*/
        /*fa 指向的 fib_alias 实例中以下参数都与 cfg 中指定的参数相同：
        *1.slen 值，2.路由选择表 tb_id 值，3.服务类型 fa_tos 值，4.优先级 fib_priority。
        *即目的地址匹配前缀（掩码）、路由选择表、服务类型、优先级相同
        *（只剩路由类型等参数不同）。*/

        struct fib_alias *fa_first, *fa_match;
        err = -EEXIST;
        if (cfg->fc_nlflags & NLM_F_EXCL)     /*netlink 消息报头 nlmsg_flags 成员值*/
            goto out;          /*保留现有 fib_alias 实例，释放新建 fib_info 实例，函数返回*/

        /*下面是查找匹配 fib_info 实例的 fib_alias 实例，替换或留用它*/
        fa_match = NULL;       /*指向找到的匹配 fib_alias 实例*/
        fa_first = fa;
```

```
/*遍历 leaf 散列链表中 fa 及其之后的 fib_alias 实例，查找与 fib_info 匹配的 fib_alias 实例，
*匹配条件是：fib_alias 实例中 fa_slen、tb_id、fa_tos、fa_info->fib_priority、fa_type 成员值
*都须与 cfg 参数指定的参数相同，fa->fa_info 成员必须指向查找到的 fib_info 实例。
*/
hlist_for_each_entry_from(fa, fa_list) {
    if ((fa->fa_slen != slen) || (fa->tb_id != tb->tb_id) ||(fa->fa_tos != tos))
        break;
    if (fa->fa_info->fib_priority != fi->fib_priority)
        break;
    if (fa->fa_type == cfg->fc_type &&fa->fa_info == fi) {
        fa_match = fa;          /*找到匹配的 fib_alias 实例*/
        break;
    }
}    /*遍历 fa 及其之后的 fib_alias 实例结束，fa_match 指向匹配实例，或为 NULL*/

/*下面是用新实例代替 fib_find_alias()函数中查找到的 fib_alias 实例，
*如果找到了匹配的 fib_alias 实例，则不替换，函数返回。*/
if (cfg->fc_nlflags & NLM_F_REPLACE) {    /*指定了需要替换老的 fib_alias 实例*/
    struct fib_info *fi_drop;
    u8 state;

    fa = fa_first;    /*fa 指向 fib_find_alias()函数中查找到的 fib_alias 实例*/
    if (fa_match) {
        if (fa == fa_match)    /*fa 是匹配的 fib_alias 实例，释放 fib_info 实例，函数返回*/
            err = 0;
        goto out;
    }
    err = -ENOBUFS;
    new_fa = kmem_cache_alloc(fn_alias_kmem, GFP_KERNEL);    /*创建 fib_alias 实例*/
    ...

    /*下面是设置新 fib_alias 实例*/
    fi_drop = fa->fa_info;          /*fa 关联的 fib_info 实例*/
    new_fa->fa_tos = fa->fa_tos;
    new_fa->fa_info = fi;
    new_fa->fa_type = cfg->fc_type;
    state = fa->fa_state;
    new_fa->fa_state = state & ~FA_S_ACCESSED;
    new_fa->fa_slen = fa->fa_slen;
    new_fa->tb_id = tb->tb_id;    /*路由选择表 id*/
    new_fa->fa_default = -1;

    err = switchdev_fib_ipv4_add(key, plen, fi,new_fa->fa_tos,
                                 cfg->fc_type,cfg->fc_nlflags,tb->tb_id);
                                /*没有选择 NET_SWITCHDEV 配置选项为空操作*/
```

```
            ...
            hlist_replace_rcu(&fa->fa_list, &new_fa->fa_list);   /*新 fa_alias 实例替换老 fa 实例*/
            alias_free_mem_rcu(fa);     /*释放原 fa_alias 实例*/
            fib_release_info(fi_drop);     /*释放原 fa_alias 实例关联的 fib_info 实例*/
            if (state & FA_S_ACCESSED)
                rt_cache_flush(cfg->fc_nlinfo.nl_net);
                                        /*net->ipv4.rt_genid 计数值加 1，/net/ipv4/route.c*/
            rtmsg_fib(RTM_NEWROUTE, htonl(key), new_fa, plen,
                        tb->tb_id, &cfg->fc_nlinfo, NLM_F_REPLACE);
                                            /*向用户套接字发送消息*/
            goto succeeded;        /*替换成功，函数返回*/
        }    /*需要替换老 fib_alias 实例结束*/

        /*cfg->fc_nlflags 没有指定需要替换老的 fa_alias 实例*/
        if (fa_match)
            goto out;   /*找到了匹配的 fib_alias 实例，不需要替换，释放 fib_info 实例，函数返回*/

        /*下面是需要在现有叶节点 leaf 散列链表增加新 fib_alias 实例*/
        if (cfg->fc_nlflags & NLM_F_APPEND)
            nlflags = NLM_F_APPEND; /*新 fib_alias 实例添加到 fa_slen 成员值相同实例的最后面*/
        else
            fa = fa_first;       /*fa 指向 fib_find_alias()函数中查找到的 fib_alias 实例*/
    }   /*大的 if 语句结束*/

    /*fib_find_node()函数查找到的最长匹配节点不是叶节点，需要创建新叶节点，添加 fib_alias 实例；
    *或者需要在现有叶节点 leaf 散列链表添加新 fib_alias 实例*/
    err = -ENOENT;
    if (!(cfg->fc_nlflags & NLM_F_CREATE))
                        /*没有指定创建新的 fib_alias 实例，释放 fib_info 实例，函数返回*/
        goto out;

    err = -ENOBUFS;
    new_fa = kmem_cache_alloc(fn_alias_kmem, GFP_KERNEL);   /*创建 fib_alias 实例*/
    ...
    /*设置新 fib_alias 实例*/
    new_fa->fa_info = fi;       /*指向 fib_info 实例*/
    new_fa->fa_tos = tos;       /*服务类型*/
    new_fa->fa_type = cfg->fc_type;     /*路由类型*/
    new_fa->fa_state = 0;
    new_fa->fa_slen = slen;       /*赋值 fa_slen 成员*/
    new_fa->tb_id = tb->tb_id;       /*路由选择表 id 值*/
    new_fa->fa_default = -1;

    err = switchdev_fib_ipv4_add(key, plen, fi, tos, cfg->fc_type,cfg->fc_nlflags, tb->tb_id);
    ...
    err = fib_insert_alias(t, tp, l, new_fa, fa, key);     /*插入新 fib_alias 实例，/net/ipv4/fib_trie.c*/
```

```
                                        /*fa 为 NULL，不为 NULL 时，新节点插入到 fa 前面，
                                          *l 为 NULL，或新实例需要插入的 TRIE 叶节点*/
    ...

    if (!plen)        /*plen 为 0，默认路由选择表项*/
        tb->tb_num_default++;      /*默认路由选择表项数量加 1*/

    rt_cache_flush(cfg->fc_nlinfo.nl_net);      /*net->ipv4.rt_genid 计数值加 1，/net/ipv4/route.c*/
    rtmsg_fib(RTM_NEWROUTE, htonl(key), new_fa, plen, new_fa->tb_id,&cfg->fc_nlinfo, nlflags);
                                        /*向用户套接字发送应答消息，/net/ipv4/fib_semantics.c*/
succeeded:
    return 0;
    ...
}
```

fib_table_insert()函数主要执行以下工作：

（1）调用 fib_create_info()函数依据 fib_config 实例创建 fib_info 实例，如果已存在相同参数的实例，则释放新创建的实例，返回已有实例指针，详见下文。

（2）依据键值 key 调用 fib_find_node(t, &tp, key)函数在 TRIE 树中查找最长匹配的节点，若匹配节点为叶节点则 fib_find_node()函数返回此节点指针，tp 指向此叶节点的父节点。如果匹配节点不是叶节点，则 fib_find_node()函数返回 NULL，tp 指向最长匹配节点（内部节点）。

（3）如果最长匹配节点是叶节点，则执行以下操作，否则跳至步骤（4）：

如果叶节点中存在完全匹配的 fa_alias 实例，释放 **fib_create_info()**函数返回的 fib_info 实例，函数返回。

如果叶节点中存在除表项类型 fa_type 外，其它参数都相同的 fa_alias 实例，且消息指定了替换老的实例，则创建新 fa_alias 实例替换叶节点中 fa_alias 实例（第一个除表项类型 fa_type 外，其它参数都相同的 fa_alias 实例），函数返回。

如果叶节点中不存在除表项类型 fa_type 外，其它参数都相同的 fa_alias 实例，或者没有指定替换老的实例，则执行步骤（4）。

（4）创建并设置 fib_alias 实例，调用 **fib_insert_alias()**函数将实例插入 TRIE 树，此函数内可能是需要在 TRIE 树中创建新的叶节点，关联 fib_alias 实例，也可能是将 fib_alias 实例插入 fib_find_node()函数查找到的叶节点 leaf 散列链表。

（5）调用 rt_cache_flush()函数将 net->ipv4.rt_genid 计数值加 1，调用 rtmsg_fib()函数向用户套接字发送消息。

向 TRIE 树插入 fa_alias 实例的 fib_insert_alias()函数前面介绍过了，下面看一下创建或查找 fib_info 实例 fib_create_info()函数的实现。

●创建 fib_info 实例

在介绍 fib_create_info()函数前，先看一下 fib_prop 结构体的定义（/net/ipv4/fib_lookup.h）：

```
struct fib_prop {
    int    error;          /*错误码，0 表示无错误*/
    u8    scope;           /*地址范围*/
};
```

内核在/net/ipv4/fib_semantics.c 文件内，定义了 fib_prop 结构体数组，每种路由（表项）类型对应一个数组项，表示路由类型对应的错误码和范围（如果创建某路由类型的范围超过了 socpe 值，将返错误码 error）：

```
const struct fib_prop fib_props[RTN_MAX + 1] = {
```

```
    [RTN_UNSPEC] = {
        .error      = 0,
        .scope      = RT_SCOPE_NOWHERE,
    },
    [RTN_UNICAST] = {
        .error      = 0,
        .scope      = RT_SCOPE_UNIVERSE,
    },
    [RTN_LOCAL] = {
        .error      = 0,
        .scope      = RT_SCOPE_HOST,
    },
    [RTN_BROADCAST] = {
        .error      = 0,
        .scope      = RT_SCOPE_LINK,
    },
    ...
};
```

fib_create_info(cfg)函数依据 fib_config 实例创建（或查找）fib_info 实例，函数定义如下：

```
struct fib_info *fib_create_info(struct fib_config *cfg)     /*/net/ipv4/fib_semantics.c*/
{
    int err;
    struct fib_info *fi = NULL;
    struct fib_info *ofi;
    int nhs = 1;       /*下一跳数量，没有配置多路径路由选择时此值为 1*/
    struct net *net = cfg->fc_nlinfo.nl_net;     /*网络命名空间*/

    if (cfg->fc_type > RTN_MAX)       /*路由类型超过最大值，返回*/
        goto err_inval;

    if (fib_props[cfg->fc_type].scope > cfg->fc_scope)
                                    /*路由类型范围超过 fib_props[cfg->fc_type].scope 值，返回*/
        goto err_inval;

  #ifdef CONFIG_IP_ROUTE_MULTIPATH        /*支持多路径路由选择*/
      ...
  #endif

    err = -ENOBUFS;
    /*fib_info_cnt 表示 fib_info 实例数量，fib_info_hash_size 表示散列表数组项数*/
    if (fib_info_cnt >= fib_info_hash_size) {
        /*fib_info 实例大于等于散列表项数时，
        *扩展并迁移 fib_info_hash 和 fib_info_laddrhash 散列表
        */
       ...
```

```
}

fi = kzalloc(sizeof(*fi)+nhs*sizeof(struct fib_nh), GFP_KERNEL);     /*创建 fib_info 实例*/
...             /*错误处理*/
fib_info_cnt++;       /*fib_info 数量值加 1*/
if (cfg->fc_mx) {     /*如果消息传递了 RTA_METRICS 属性，为 fib_metrics 成员分配空间*/
      fi->fib_metrics = kzalloc(sizeof(u32) * RTAX_MAX, GFP_KERNEL);
      ...
} else                  /*如果消息没有传递 RTA_METRICS 属性，使用默认值，/net/core/dst.c*/
      fi->fib_metrics = (u32 *) dst_default_metrics;       /*空数组*/

/*设置 fib_info 实例*/
fi->fib_net = net;
fi->fib_protocol = cfg->fc_protocol;
fi->fib_scope = cfg->fc_scope;
fi->fib_flags = cfg->fc_flags;
fi->fib_priority = cfg->fc_priority;
fi->fib_prefsrc = cfg->fc_prefsrc;
fi->fib_type = cfg->fc_type;

fi->fib_nhs = nhs;     /*下一跳数量*/
change_nexthops(fi) {       /*设置 fib_nh 实例，为其分配 rtable 实例指针数组（percpu 变量）*/
      nexthop_nh->nh_parent = fi;       /*指向 fib_info 实例*/
      nexthop_nh->nh_pcpu_rth_output = alloc_percpu(struct rtable __rcu *); /*rtable 指针数组*/
      ...       /*错误处理*/
} endfor_nexthops(fi)

if (cfg->fc_mx) {       /*解析 RTA_METRICS 属性，填充 fi->fib_metrics[]数组*/
      struct nlattr *nla;
      int remaining;

      nla_for_each_attr(nla, cfg->fc_mx, cfg->fc_mx_len, remaining) {    /*遍历子属性*/
            int type = nla_type(nla);
            if (type) {
                  u32 val;
                  if (type > RTAX_MAX)
                        goto err_inval;
                  if (type == RTAX_CC_ALGO) {     /*拥塞控制算法属性*/
                        char tmp[TCP_CA_NAME_MAX];
                        nla_strlcpy(tmp, nla, sizeof(tmp));
                        val = tcp_ca_get_key_by_name(tmp); /*由名称查找拥塞算法中 ca->key 键值*/
                        if (val == TCP_CA_UNSPEC)
                              goto err_inval;
                  } else {
                        val = nla_get_u32(nla);
                  }
```

```
                if (type == RTAX_ADVMSS && val > 65535 - 40)
                    val = 65535 - 40;
                if (type == RTAX_MTU && val > 65535 - 15)
                    val = 65535 - 15;
                fi->fib_metrics[type - 1] = val;        /*设置参数到 fib_metrics[]数组项*/
            }
        }       /*遍历 RTA_METRICS 属性下的子属性结束*/
    }    /*解析 RTA_METRICS 属性结束*/

    if (cfg->fc_mp) {     /*如果指定了下一跳，RTA_MULTIPATH 属性值*/
    #ifdef CONFIG_IP_ROUTE_MULTIPATH
        ...
    #endif
    } else {       /*没有指定表示下一跳信息的 RTA_MULTIPATH 属性*/
        struct fib_nh *nh = fi->fib_nh;

        nh->nh_oif = cfg->fc_oif;       /*输出网络设备*/
        nh->nh_gw = cfg->fc_gw;       /*下一跳 IP 地址*/
        nh->nh_flags = cfg->fc_flags;       /*标志*/
    #ifdef CONFIG_IP_ROUTE_CLASSID
        ...
    #endif
    #ifdef CONFIG_IP_ROUTE_MULTIPATH
        ...
    #endif
    }

    if (fib_props[cfg->fc_type].error) {    /*路由类型错误码非 0*/
        if (cfg->fc_gw || cfg->fc_oif || cfg->fc_mp)
            goto err_inval;
        goto link_it;        /*在散列表中查找 fib_info 实例*/
    } else {
        switch (cfg->fc_type) {
        case RTN_UNICAST:
        case RTN_LOCAL:
        case RTN_BROADCAST:
        case RTN_ANYCAST:
        case RTN_MULTICAST:       /*有效的路由类型*/
            break;
        default:
            goto err_inval;
        }
    }
    if (cfg->fc_scope > RT_SCOPE_HOST)         /**/
        goto err_inval;
```

```
    if (cfg->fc_scope == RT_SCOPE_HOST) {     /*目的地址是本地，不需要外发*/
        struct fib_nh *nh = fi->fib_nh;

        if (nhs != 1 || nh->nh_gw)
            goto err_inval;
        nh->nh_scope = RT_SCOPE_NOWHERE;     /*设置路由类型，不需要外发*/
        nh->nh_dev = dev_get_by_index(net, fi->fib_nh->nh_oif);     /*由 id 查找 net_device 实例*/
        ...        /*错误处理（未找到指定网络设备）*/
    } else {        /*路由范围为其它值*/
        int linkdown = 0;

        change_nexthops(fi) {    /*遍历下一跳*/
            err = fib_check_nh(cfg, fi, nexthop_nh);     /*检查下一跳有效性*/
            ...      /*错误处理*/
            if (nexthop_nh->nh_flags & RTNH_F_LINKDOWN)
                linkdown++;
        } endfor_nexthops(fi)
        if (linkdown == fi->fib_nhs)
            fi->fib_flags |= RTNH_F_LINKDOWN;
    }

    if (fi->fib_prefsrc) {     /*消息指定了本地地址*/
        if (cfg->fc_type != RTN_LOCAL || !cfg->fc_dst ||fi->fib_prefsrc != cfg->fc_dst)
            if (inet_addr_type(net, fi->fib_prefsrc) != RTN_LOCAL)
                                                /*目的地址不是本地类型，返回错误码*/
                goto err_inval;
    }

    change_nexthops(fi) {      /*遍历下一跳，设置 fib_nh->nh_saddr 成员（本地地址）*/
        fib_info_update_nh_saddr(net, nexthop_nh);    /*/net/ipv4/fib_semantics.c*/
    } endfor_nexthops(fi)

link_it:
    ofi = fib_find_info(fi);
      /*在 fib_info_hash 散列表中查找是否存在相同的 fib_info 实例，/net/ipv4/fib_semantics.c*/
    if (ofi) {          /*存在则释放分配的实例，函数返回查找到的现有实例*/
        fi->fib_dead = 1;
        free_fib_info(fi);       /*释放 fib_info 实例*/
        ofi->fib_treeref++;       /*现有实例增加引用计数*/
        return ofi;               /*返回现有实例*/
    }

    /*没有找到相同的 fib_info 实例*/
    fi->fib_treeref++;
    atomic_inc(&fi->fib_clntref);
    spin_lock_bh(&fib_info_lock);
```

```
        hlist_add_head(&fi->fib_hash,&fib_info_hash[fib_info_hashfn(fi)]);
                                                        /*将实例插入 fib_info_hash 散列表*/
        if (fi->fib_prefsrc) {              /*将实例插入 fib_info_laddrhash 散列表*/
            struct hlist_head *head;
            head = &fib_info_laddrhash[fib_laddr_hashfn(fi->fib_prefsrc)];
            hlist_add_head(&fi->fib_lhash, head);
        }

        change_nexthops(fi) {       /*遍历下一跳，将 fib_nh 实例插入散列表*/
            struct hlist_head *head;
            unsigned int hash;
            if (!nexthop_nh->nh_dev)
                continue;
            hash = fib_devindex_hashfn(nexthop_nh->nh_dev->ifindex);
                                                /*将 fib_nh 实例插入 fib_info_devhash 散列表*/
            head = &fib_info_devhash[hash];
            hlist_add_head(&nexthop_nh->nh_hash, head);
        } endfor_nexthops(fi)
        spin_unlock_bh(&fib_info_lock);
        return fi;      /*返回新 fib_info 实例*/
        ...
    }
```

fib_create_info()函数虽然比较长，但比较容易理解，函数完成的主要工作如下：

●检查是否要扩展并迁移 fib_info_hash 和 fib_info_laddrhash 散列表，需要则扩展并迁移散列表。

●创建并设置 fib_info 实例，及设置其中的下一跳 fib_nh 实例。

●对于目的地址不是本机的路由，调用 **fib_check_nh()**函数检查其下一跳的有效性，详见下文。

●在 fib_info_hash 散列表中查找是否存在相同参数的 fib_info 实例，如果存在则使用旧实例，释放新创建的实例，函数返回，如果没有相同参数的 fib_info 实例，则继续往下执行。

●将新 fib_info 实例及其下一跳 fib_nh 实例插入管理散列表，返回 fib_info 实例指针。

fib_check_nh()函数用于检查外发路由 fib_info 实例中下一跳 fib_nh 实例的有效性，主要是判断下一跳 IP 地址是否有效，以及输出网络设备是否存在且已打开（可用），函数定义如下（/net/ipv4/fib_semantics.c）：

```
    static int fib_check_nh(struct fib_config *cfg, struct fib_info *fi,struct fib_nh *nh)
    {
        int err;
        struct net *net;
        struct net_device *dev;

        net = cfg->fc_nlinfo.nl_net;     /*指向网络空间*/
        if (nh->nh_gw) {       /*如果指定了下一跳 IP 地址*/
            struct fib_result res;        /*保存查找路由选择表项结果*/
            if (nh->nh_flags & RTNH_F_ONLINK) {     /*设置了 RTNH_F_ONLINK 标志*/
                if (cfg->fc_scope >= RT_SCOPE_LINK)
                    return -EINVAL;
                if (inet_addr_type(net, nh->nh_gw) != RTN_UNICAST)    /*网关地址类型*/
                    return -EINVAL;          /*非单播地址为无效地址*/
```

```
            dev = __dev_get_by_index(net, nh->nh_oif);        /*由编号查找网络设备*/
            if (!dev)           /*网络设备不存在*/
                return -ENODEV;
            if (!(dev->flags & IFF_UP))       /*网络设备未找开*/
                return -ENETDOWN;
            if (!netif_carrier_ok(dev))   /*测试 dev->state 成员__LINK_STATE_NOCARRIE 标记*/
                nh->nh_flags |= RTNH_F_LINKDOWN;
            nh->nh_dev = dev;
            dev_hold(dev);        /*增加引用计数*/
            nh->nh_scope = RT_SCOPE_LINK;
            return 0;
        }
        /*指定了下一跳 IP 地址，但 nh->nh_flags 没有设置 RTNH_F_ONLINK 标志*/
        rcu_read_lock();
        {    /*通过路由选择查找检查下一跳地址是否属于单播或本地地址*/
            struct flowi4 fl4 = {
                .daddr = nh->nh_gw,
                .flowi4_scope = cfg->fc_scope + 1,
                .flowi4_oif = nh->nh_oif,
                .flowi4_iif = LOOPBACK_IFINDEX,
            };

            if (fl4.flowi4_scope < RT_SCOPE_LINK)
                fl4.flowi4_scope = RT_SCOPE_LINK;
            err = fib_lookup(net, &fl4, &res,FIB_LOOKUP_IGNORE_LINKSTATE);
                                    /*查找路由选择表项，确定下一跳地址类型等，见下文*/
            ...        /*错误处理*/
        }
        err = -EINVAL;
        if (res.type != RTN_UNICAST && res.type != RTN_LOCAL)
            goto out;                    /*非单播、本地地址，返回错误码*/
        nh->nh_scope = res.scope;         /*范围*/
        nh->nh_oif = FIB_RES_OIF(res);       /*网络设备编号*/
        nh->nh_dev = dev = FIB_RES_DEV(res);       /*网络设备*/
        if (!dev)
            goto out;
        dev_hold(dev);
        if (!netif_carrier_ok(dev))
            nh->nh_flags |= RTNH_F_LINKDOWN;
        err = (dev->flags & IFF_UP) ? 0 : -ENETDOWN;         /*网络设备未打开，返回错误码*/
    }
    else {       /*如果没有指定下一跳 IP 地址*/
        struct in_device *in_dev;

        if (nh->nh_flags & (RTNH_F_PERVASIVE | RTNH_F_ONLINK))
            return -EINVAL;
```

```
            rcu_read_lock();
            err = -ENODEV;
            in_dev = inetdev_by_index(net, nh->nh_oif);     /*查找输出网络设备*/
            if (!in_dev)
                goto out;
            err = -ENETDOWN;
            if (!(in_dev->dev->flags & IFF_UP))     /*网络设备是否打开*/
                goto out;
            nh->nh_dev = in_dev->dev;       /*关联网络设备*/
            dev_hold(nh->nh_dev);
            nh->nh_scope = RT_SCOPE_HOST;
            if (!netif_carrier_ok(nh->nh_dev))
                nh->nh_flags |= RTNH_F_LINKDOWN;
            err = 0;
        }
    out:
        rcu_read_unlock();
        return err;
    }
```

5 本机地址管理

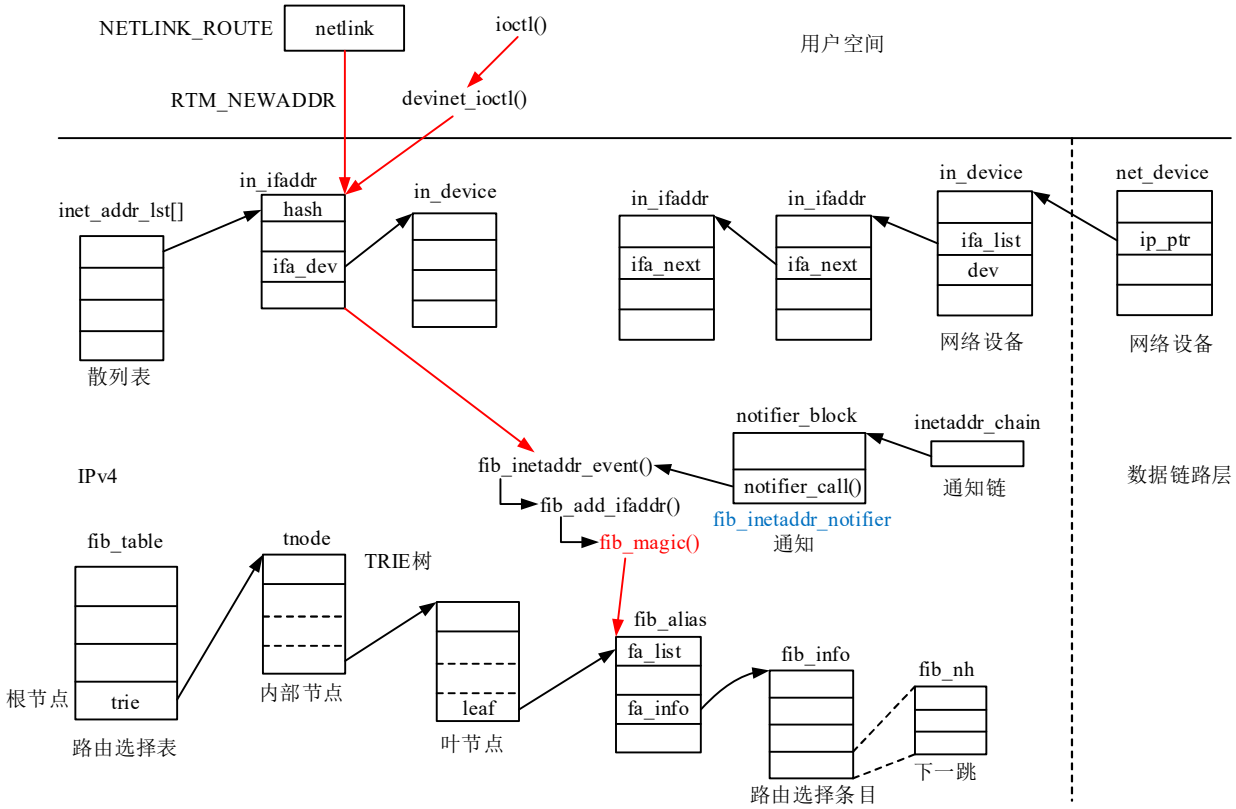
主机和路由器的每个接口（网络设备）至少需要分配一个 IP 地址，甚至可以有多个 IP 地址。用户可以通过 NETLINK_ROUTE 协议类型的 netlink 套接字 RTM_NEWADDR 类型消息设置接口 IP 地址，也可以通过 ioctl()系统调用（ifconfig）设置 IP 地址。

网络设备在驱动程序中由 net_device 实例表示，其关联了用户设置的 IP 地址信息，添加的本机接口 IP 地下还会在路由选择表中增加路由选择表项。

■概述

网络设备（接口）在驱动程序中（数据链路层）由 net_device 结构体表示，在 IPv4 网络层网络设备由 in_device 结构体表示，net_device 结构体中具有指向 in_device 结构体的指针成员。net_device 是数据链路层表示设备的通用数据结构，可适配所有的网络层协议，in_device 结构体只在 IPv4 网络层协议中表示网络设备，如下图所示。

网络设备 IP 地址在 IPv4 网络层由 in_ifaddr 结构体表示，结构体实例由全局散列表管理。一个网络设备可以有多个地址，因此 in_device 结构体中具有指向 in_ifaddr 实例单链表的指针成员，in_ifaddr 结构体具有指向对应网络设备 in_device 实例的指针成员。



用户可通过向内核发送 RTM_NEWADDR 类型消息、ioctl()系统调用等设置网络设备 IP 地址，消息处理函数内将创建表示地址的 in_ifaddr 结构体实例，并关联到 in_device 实例和插入全局散列表，最后执行 inetaddr_chain 通知链中的通知。

在路由选择表初始化函数 ip_fib_init()中，向 inetaddr_chain 通知链注册了 **fib_inetaddr_notifier** 通知实例，此通知执行函数内将依据 in_ifaddr 实例向路由选择表添加表示本机地址的路由选择表项。

IPv4 网络层协议中网络设备（IP 地址）相关代码位于/net/ipv4/devinet.c 文件内。

■数据结构

在 IPv4 网络层协议中，网络设备（接口）由 in_device 结构体表示，网络设备驱动中 net_device 结构体中 ip_ptr 成员指向 in_device 结构体实例（网络设备在 IP 网络层中的表示）。

in_device 结构体定义如下（/include/linux/inetdevice.h）：

```
struct in_device {
    struct net_device    *dev;      /*指向网络设备 net_device 实例*/
    atomic_t             refcnt;    /*引用计数*/
    int                  dead;      /*设备是否关闭*/
    struct in_ifaddr     *ifa_list;     /*指向 in_ifaddr 实例单链表，表示网络设备 IP 地址信息*/

    struct ip_mc_list __rcu   *mc_list;     /**/
    struct ip_mc_list __rcu   * __rcu *mc_hash;

    int                  mc_count;      /* Number of installed mcasts*/
    spinlock_t           mc_tomb_lock;
    struct ip_mc_list    *mc_tomb;
    unsigned long        mr_v1_seen;
    unsigned long        mr_v2_seen;
```

```
    unsigned long       mr_maxdelay;
    unsigned char       mr_qrv;
    unsigned char       mr_gq_running;
    unsigned char       mr_ifc_count;
    struct timer_list   mr_gq_timer;  /* general query timer */
    struct timer_list   mr_ifc_timer; /* interface change timer */

    struct neigh_parms *arp_parms;     /*指向邻居参数，见下一小节*/
    struct ipv4_devconf    cnf;        /*网络设备配置参数，来自于网络命名空间*/
    struct rcu_head     rcu_head;
};
```

in_device 结构体中 ifa_list 成员管理的是 in_ifaddr 实例单链表，表示的是网络设备的 IP 地址（接口）信息。

in_ifaddr 结构体定义如下（/include/linux/inetdevice.h）：

```
struct in_ifaddr {
    struct hlist_node   hash;         /*散列链表节点，将实例添加到全局散列表 inet_addr_lst[]*/
    struct in_ifaddr    *ifa_next;    /*指向下一个 in_ifaddr 实例，添加到 in_device 中单链表*/
    struct in_device    *ifa_dev;     /*指向 in_device 实例*/
    struct rcu_head     rcu_head;     /*rcu 队列头*/
    __be32              ifa_local;    /*本地地址*/
    __be32              ifa_address;  /*目的地址*/
    __be32              ifa_mask;     /*子网掩码*/
    __be32              ifa_broadcast;/*广播地址*/
    unsigned char       ifa_scope;    /*范围*/
    unsigned char       ifa_prefixlen;/*子网掩码中 1 的个数*/
    __u32               ifa_flags;    /*标志，/include/uapi/linux/if_addr.h*/
    char            ifa_label[IFNAMSIZ];  /*兼容旧版本的结构*/

    /* In seconds, relative to tstamp. Expiry is at tstamp + HZ * lft. */
    __u32               ifa_valid_lft;    /*地址有效时间*/
    __u32               ifa_preferred_lft;
    unsigned long       ifa_cstamp;  /*创建时间戳*/
    unsigned long       ifa_tstamp;  /*更新时间戳*/
};
```

内核在/net/ipv4/devinet.c 文件内定义了全局散列表 **inet_addr_lst**[IN4_ADDR_HSIZE]，用于管理 in_ifaddr 实例。

■网络设备初始化

网络命名空间中需要保存网络设备在网络层中的配置参数，配置参数由 ipv4_devconf 结构体表示。每个网络命名空间具有自身的 ipv4_devconf 实例，IPv4 网络资源 netns_ipv4 结构体中具有指向 ipv4_devconf 实例的指针成员，如下所示：

```
struct netns_ipv4 {
    ...
```

```
    struct ipv4_devconf  *devconf_all;     /*网络设备配置参数*/
    struct ipv4_devconf  *devconf_dflt;    /*网络设备配置参数*/
    ...
}
```

ipv4_devconf 结构体定义如下（/include/linux/inetdevice.h）：

```
struct ipv4_devconf {
    void *sysctl;        /*系统控制项*/
    int   data[IPV4_DEVCONF_MAX];        /*配置参数数组，整数数组*/
    DECLARE_BITMAP(state, IPV4_DEVCONF_MAX);     /*位图*/
};
```

ipv4_devconf 结构体中 data[]数组表示网络设备配置参数，数组项定义如下（/include/uapi/linux/ip.h）：

```
enum   {
    IPV4_DEVCONF_FORWARDING=1,
    IPV4_DEVCONF_MC_FORWARDING,
    IPV4_DEVCONF_PROXY_ARP,
    IPV4_DEVCONF_ACCEPT_REDIRECTS,
    IPV4_DEVCONF_SECURE_REDIRECTS,
    IPV4_DEVCONF_SEND_REDIRECTS,
    IPV4_DEVCONF_SHARED_MEDIA,
    IPV4_DEVCONF_RP_FILTER,
    IPV4_DEVCONF_ACCEPT_SOURCE_ROUTE,
    IPV4_DEVCONF_BOOTP_RELAY,
    IPV4_DEVCONF_LOG_MARTIANS,
    IPV4_DEVCONF_TAG,
    IPV4_DEVCONF_ARPFILTER,
    IPV4_DEVCONF_MEDIUM_ID,
    IPV4_DEVCONF_NOXFRM,
    IPV4_DEVCONF_NOPOLICY,
    IPV4_DEVCONF_FORCE_IGMP_VERSION,
    IPV4_DEVCONF_ARP_ANNOUNCE,
    IPV4_DEVCONF_ARP_IGNORE,
    IPV4_DEVCONF_PROMOTE_SECONDARIES,
    IPV4_DEVCONF_ARP_ACCEPT,
    IPV4_DEVCONF_ARP_NOTIFY,
    IPV4_DEVCONF_ACCEPT_LOCAL,
    IPV4_DEVCONF_SRC_VMARK,
    IPV4_DEVCONF_PROXY_ARP_PVLAN,
    IPV4_DEVCONF_ROUTE_LOCALNET,
    IPV4_DEVCONF_IGMPV2_UNSOLICITED_REPORT_INTERVAL,
    IPV4_DEVCONF_IGMPV3_UNSOLICITED_REPORT_INTERVAL,
    IPV4_DEVCONF_IGNORE_ROUTES_WITH_LINKDOWN,
    __IPV4_DEVCONF_MAX
};
```

内核在/net/ipv4/devinet.c 文件内为初始网络命名空间定义了 ipv4_devconf 实例 **ipv4_devconf** 和 **ipv4_devconf_dflt**，表示初始的网络设备配置参数。

网络设备在 IPv4 网络层初始化工作由 **devinet_init()**函数完成，由前面介绍的 ip_rt_init()函数调用，函数定义如下（/net/ipv4/devinet.c）：

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

    for (i = 0; i < IN4_ADDR_HSIZE; i++)
        INIT_HLIST_HEAD(&inet_addr_lst[i]);     /*初始化管理 in_ifaddr 实例的散列表*/

    register_pernet_subsys(&devinet_ops);   /*devinet_ops 实例初始化函数为 devinet_init_net()*/

    register_gifconf(PF_INET, inet_gifconf);     /*注册 SIOCGIF 命令的处理函数，/net/core/dev_ioctl.c*/
    register_netdevice_notifier(&ip_netdev_notifier);
                                /*向原始通知链 netdev_chain 注册通知，/net/core/dev.c*/

    queue_delayed_work(system_power_efficient_wq, &check_lifetime_work, 0);     /*延时工作*/

    rtnl_af_register(&inet_af_ops);
                        /*注册地址簇操作结果，用于处理 NETLINK_ROUTE 套接字消息*/
    rtnl_register(PF_INET, RTM_NEWADDR, inet_rtm_newaddr, NULL, NULL);
                                                    /*注册设置新地址消息处理函数*/
    rtnl_register(PF_INET, RTM_DELADDR, inet_rtm_deladdr, NULL, NULL);
                                                    /*注册删除地址消息处理函数*/
    rtnl_register(PF_INET, RTM_GETADDR, NULL, inet_dump_ifaddr, NULL);
                                                   /*注册获取地址消息处理函数*/
    rtnl_register(PF_INET, RTM_GETNETCONF, inet_netconf_get_devconf,
                                                       inet_netconf_dump_devconf, NULL);
                                                    /*注册获取配置参数消息的处理函数*/
}
```

devinet_init()函数内完成以下主要工作：

（1）初始化管理 in_ifaddr 实例的全局散列表 inet_addr_lst[]。

（2）注册 pernet_operations 结构体实例 **devinet_ops**，实例初始化函数为 **devinet_init_net()**，函数内判断若是初始网络命名空间，则将 net->ipv4.devconf_all 和 net->ipv4.devconf_dflt 成员分别指 **ipv4_devconf** 和 **ipv4_devconf_dflt** 实例。如果不是初始网络命名空间，则为其创建新实例。

（3）向 gifconf_list[]函数指针数组注册 IPv4 协议簇 SIOCGIF 命令的处理函数 **inet_gifconf()**。内核在文件/net/core/dev_ioctl.c 文件内定义了 gifconf_list[]函数指针数组，每个协议簇对应一个数组项，数组项是函数指针，表示协议簇对 SIOCGIF（通用配置接口）命令的处理函数。

（4）向原始通知链 **netdev_chain** 注册通知 **ip_netdev_notifier** 实例。内核在/net/core/dev.c 文件定义了原始通知链 **netdev_chain**，register_netdevice_notifier()函数用于向通知链注册通知，在网络设备有事件发送时，将会执行通知链中的通知。**ip_netdev_notifier** 实例中的处理函数用于处理设备事件（/net/ipv4/devinet.c），如果是注册网络设备时（net_device）执行此通知，将为设备创建并初始化 in_device 实例，也就是说网络接口对应的 in_device 实例在注册网络设备时创建，详见下文。

（5）注册 IPv4 地址簇操作结构 **inet_af_ops** 实例（结构体定义见 12.3 节），实例定义在/net/ipv4/devinet.c

文件内，用于处理 NETLINK_ROUTE 套接字消息，处理接口参数等。

（6）注册 NETLINK_ROUTE 套接字 RTM_NEWADDR、RTM_DELADDR、RTM_GETADDR 和 RTM_GETNETCONF 消息的处理函数。

●创建 in_device 实例

devinet_init()函数中向 netdev_chain 通知链注册的 ip_netdev_notifier 通知，定义如下（/net/ipv4/devinet.c）：

```
static struct notifier_block ip_netdev_notifier = {
    .notifier_call = inetdev_event,     /*通知回调函数，/net/ipv4/devinet.c*/
};
```

通知回调函数为 inetdev_event()。网络设备驱动程序在注册 net_device 实例时，将执行 netdev_chain 通知链中通知，设备事件为 NETDEV_REGISTER。ip_netdev_notifier 通知回调函数 inetdev_event()中此时将调用 inetdev_init(dev)函数为网络设备创建 in_device 实例（如果 in_device 实例尚不存在）。

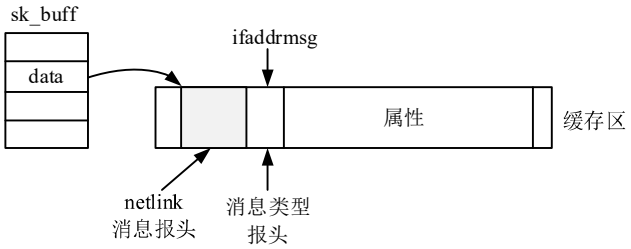
inetdev_init(dev)函数定义如下（/net/ipv4/devinet.c）：

```
static struct in_device *inetdev_init(struct net_device *dev)
{
    struct in_device *in_dev;
    int err = -ENOMEM;

    ASSERT_RTNL();

    in_dev = kzalloc(sizeof(*in_dev), GFP_KERNEL);       /*分配 in_device 实例*/
    ...      /*错误处理*/
    memcpy(&in_dev->cnf, dev_net(dev)->ipv4.devconf_dflt,sizeof(in_dev->cnf));
                                                    /*复制网络命名空间中的默认配置参数*/
    in_dev->cnf.sysctl = NULL;
    in_dev->dev = dev;
    in_dev->arp_parms = neigh_parms_alloc(dev, &arp_tbl);   /*/net/core/neighbour.c*/
                            /*创建邻居参数 neigh_parms 实例，并添加到 arp_tbl 邻居表中的参数链表*/
    ...   /*错误处理*/
    if (IPV4_DEVCONF(in_dev->cnf, FORWARDING))
        dev_disable_lro(dev);

    dev_hold(dev);
    in_dev_hold(in_dev);

    err = devinet_sysctl_register(in_dev);       /*注册系统控制参数*/
    ...     /*错误处理*/
    ip_mc_init_dev(in_dev);       /*本机为组播路由器时，执行初始化，/net/ipv4/igmp.c*/
    if (dev->flags & IFF_UP)
        ip_mc_up(in_dev);       /*启动设备（组播路由器），/net/ipv4/igmp.c*/

    rcu_assign_pointer(dev->ip_ptr, in_dev);       /*net_device->ip_ptr 指向 in_device 实例*/
out:
    return in_dev ?: ERR_PTR(err);         /*返回 in_device 实例指针*/
    ...
```

}

■设置本机地址

网络接口地址参数保存在 in_ifaddr 结构体中，用户可通过 ip、ifconfig 等命令设置网络接口 IP 地址。其中 ip 命令通过 NETLINK_ROUTE 类型套接字 RTM_NEWADDR 类型消息设置网络接口 IP 地址，消息结构如下图所示：



消息类型参数 RTM_NEWADDR 保存在 netlink 消息报头 **nlmsg_type** 字段中，在 netlink 消息报头后面是 RTM_NEWADDR 消息类型报头，由 ifaddrmsg 结构体表示，消息类型报头后面是属性。

RTM_NEWADDR 消息报头 ifaddrmsg 结构体定义在/include/uapi/linux/if_addr.h 头文件：

```
struct ifaddrmsg {
    __u8        ifa_family;     /*协议簇标识，必须是第一个成员*/
    __u8        ifa_prefixlen;  /*匹配前缀长度（掩码中 1 的位数）*/
    __u8        ifa_flags;      /*标记*/
    __u8        ifa_scope;      /*地址范围*/
    __u32       ifa_index;      /*网络接口编号（索引值）*/
};
```

RTM_NEWADDR 消息类型中属性类型定义如下（/include/uapi/linux/if_addr.h）：

```
enum {
    IFA_UNSPEC,
    IFA_ADDRESS,
    IFA_LOCAL,      /*本地地址*/
    IFA_LABEL,
    IFA_BROADCAST,    /*广播地址*/
    IFA_ANYCAST,
    IFA_CACHEINFO,      /*/include/uapi/linux/if_addr.h*/
            /*缓存信息，由 ifa_cacheinfo 结构体实例表示，可设置更详细的信息，如有效期等*/
    IFA_MULTICAST,
    IFA_FLAGS,
    __IFA_MAX,      /*属性类型最大值*/
};
```

RTM_NEWADDR 类型消息由 **inet_rtm_newaddr()**函数处理（见 devinet_init()函数），函数定义如下：

```
static int inet_rtm_newaddr(struct sk_buff *skb, struct nlmsghdr *nlh)     /*/net/ipv4/devinet.c*/
/*skb：指向接收到的数据包，nlh：指向 netlink 消息报头*/
{
    struct net *net = sock_net(skb->sk);
    struct in_ifaddr *ifa;
```

```
    struct in_ifaddr *ifa_existing;
    __u32 valid_lft = INFINITY_LIFE_TIME;       /*最大值 0xFFFFFFFF*/
    __u32 prefered_lft = INFINITY_LIFE_TIME;

    ASSERT_RTNL();

    ifa = rtm_to_ifaddr(net, nlh, &valid_lft, &prefered_lft);
                        /*将消息转换成 in_ifaddr 结构体实例，关联 in_device 实例，/net/ipv4/devinet.c*/
    ...
    ifa_existing = find_matching_ifa(ifa);     /*查找是否已存在匹配的 in_ifaddr，没有则返回 NULL*/
    if (!ifa_existing) {        /*没有匹配实例*/
        set_ifa_lifetime(ifa, valid_lft, prefered_lft);       /*设置地址有效时间*/
        if (ifa->ifa_flags & IFA_F_MCAUTOJOIN) {
            int ret = ip_mc_config(net->ipv4.mc_autojoin_sk,true, ifa);
            ...
        }
        return __inet_insert_ifa(ifa, nlh, NETLINK_CB(skb).portid);
                        /*将 in_ifaddr 实例插入散列表，添加本地路由等，/net/ipv4/devinet.c*/
    } else {        /*如果已有匹配的 in_ifaddr 实例*/
        inet_free_ifa(ifa);       /*释放已分配的 in_ifaddr 实例*/

        if (nlh->nlmsg_flags & NLM_F_EXCL ||!(nlh->nlmsg_flags & NLM_F_REPLACE))
            return -EEXIST;
        ifa = ifa_existing;
        set_ifa_lifetime(ifa, valid_lft, prefered_lft);
        cancel_delayed_work(&check_lifetime_work);
        queue_delayed_work(system_power_efficient_wq,&check_lifetime_work, 0);
        rtmsg_ifa(RTM_NEWADDR, ifa, nlh, NETLINK_CB(skb).portid);   /*向用户套接字发送消息*/
    }
    return 0;
}
```

inet_rtm_newaddr()函数中首先将 RTM_NEWADDR 消息转换成 in_ifaddr 实例，然后在网络设备对应的 in_device 实例 ifa_list 单链表中查找是否已存在匹配的 in_ifaddr 实例，如果存在则释放新创建的实例，并向用户套接字发送应答消息等。如果没有匹配的 in_ifaddr 实例，则将新实例插入 in_device 实例中 ifa_list 单链表以及全局散列表，并向用户套接字发送应答消息等。

在__inet_insert_ifa()函数中还将执行通知链 **inetaddr_chain** 中的通知，其中 fib_inetaddr_notifier 通知的回调函数将向路由选择表添加本机地址的路由选择表项。

下面将详细介绍 inet_rtm_newaddr()函数中调用函数的实现。

●消息转 in_ifaddr 实例

rtm_to_ifaddr()函数用于将 RTM_NEWADDR 类型消息转换成 in_ifaddr 实例，函数定义如下：

```
static struct in_ifaddr *rtm_to_ifaddr(struct net *net, struct nlmsghdr *nlh,
                                       __u32 *pvalid_lft, __u32 *pprefered_lft)
{
    struct nlattr *tb[IFA_MAX+1];       /*属性头指针数组*/
    struct in_ifaddr *ifa;          /*指向 in_ifaddr 实例*/
```

```
struct ifaddrmsg *ifm;        /*指向消息类型报头*/
struct net_device *dev;       /*指向网络设备（驱动程序中的表示）*/
struct in_device *in_dev;     /*指向网络设备在网络层协议中的表示*/
int err;

err = nlmsg_parse(nlh, sizeof(*ifm), tb, IFA_MAX, ifa_ipv4_policy);     /*解析属性*/
...
ifm = nlmsg_data(nlh);     /*从消息中获取消息类型报头，ifaddrmsg 实例指针*/
err = -EINVAL;
if (ifm->ifa_prefixlen > 32 || !tb[IFA_LOCAL])
        goto errout;

dev = __dev_get_by_index(net, ifm->ifa_index);     /*由索引值查找网络设备，见下一节*/
...
in_dev = __in_dev_get_rtnl(dev);
                /*获取 net_device 指向的 in_device 实例，注册 net_device 实例时创建*/
...
ifa = inet_alloc_ifa();     /*分配 in_ifaddr 实例（清零）*/
...
ipv4_devconf_setall(in_dev);   /*填充 in_dev->cnf.state 位图，/include/linux/inetdevice.h*/
neigh_parms_data_state_setall(in_dev->arp_parms);     /*设置邻居参数*/
in_dev_hold(in_dev);       /*引用计数 refcnt 加 1*/

if (!tb[IFA_ADDRESS])     /*如果消息没有设置 IFA_ADDRESS 属性*/
        tb[IFA_ADDRESS] = tb[IFA_LOCAL];        /*设置与本地地址相同*/

/*设置 in_ifaddr 实例*/
INIT_HLIST_NODE(&ifa->hash);
ifa->ifa_prefixlen = ifm->ifa_prefixlen;       /*匹配前缀长度*/
ifa->ifa_mask = inet_make_mask(ifm->ifa_prefixlen);     /*设置子网掩码*/
ifa->ifa_flags = tb[IFA_FLAGS] ? nla_get_u32(tb[IFA_FLAGS]) :ifm->ifa_flags;
ifa->ifa_scope = ifm->ifa_scope;
ifa->ifa_dev = in_dev;       /*in_device 实例*/

ifa->ifa_local = nla_get_in_addr(tb[IFA_LOCAL]);       /*本地地址（IP 地址）*/
ifa->ifa_address = nla_get_in_addr(tb[IFA_ADDRESS]);

if (tb[IFA_BROADCAST])
        ifa->ifa_broadcast = nla_get_in_addr(tb[IFA_BROADCAST]);     /*设置广播地址*/

if (tb[IFA_LABEL])
        nla_strlcpy(ifa->ifa_label, tb[IFA_LABEL], IFNAMSIZ);
else
        memcpy(ifa->ifa_label, dev->name, IFNAMSIZ);     /*复制标签*/

if (tb[IFA_CACHEINFO]) {
```

```
        struct ifa_cacheinfo *ci;

        ci = nla_data(tb[IFA_CACHEINFO]);     /*缓存信息，由 ifa_cacheinfo 结构体表示*/
        if (!ci->ifa_valid || ci->ifa_prefered > ci->ifa_valid) {
            err = -EINVAL;
            goto errout_free;
        }
        *pvalid_lft = ci->ifa_valid;
        *pprefered_lft = ci->ifa_prefered;
    }
    return ifa;     /*返回 in_ifaddr 实例指针*/
    ...
}
```

●插入 in_ifaddr 实例

在创建并设置 in_ifaddr 实例后，inet_rtm_newaddr()函数调用**__inet_insert_ifa()**函数将 in_ifaddr 实例插入 in_device 实例中的单链表以及全局散列表，并执行 inetaddr_chain 通知链中的通知。

__inet_insert_ifa()函数定义如下（/net/ipv4/devinet.c）：

```
static int __inet_insert_ifa(struct in_ifaddr *ifa, struct nlmsghdr *nlh,u32 portid)
/*ifa：指向 in_ifaddr 实例，nlh：指向 netlink 消息报头，portid：用户套接字端口号*/
{
    struct in_device *in_dev = ifa->ifa_dev;
    struct in_ifaddr *ifa1, **ifap, **last_primary;

    ASSERT_RTNL();

    if (!ifa->ifa_local) {          /*必须有本地地址*/
        inet_free_ifa(ifa);
        return 0;
    }

    ifa->ifa_flags &= ~IFA_F_SECONDARY;
    last_primary = &in_dev->ifa_list;     /*in_ifaddr 实例链表*/

    /*遍历 in_device 实例中 in_ifaddr 实例单链表，寻找插入位置*/
    for (ifap = &in_dev->ifa_list; (ifa1 = *ifap) != NULL;ifap = &ifa1->ifa_next) {
        if (!(ifa1->ifa_flags & IFA_F_SECONDARY) &&ifa->ifa_scope <= ifa1->ifa_scope)
            last_primary = &ifa1->ifa_next;
        if (ifa1->ifa_mask == ifa->ifa_mask &&inet_ifa_match(ifa1->ifa_address, ifa)) {
            if (ifa1->ifa_local == ifa->ifa_local) {
                inet_free_ifa(ifa);
                return -EEXIST;
            }
            if (ifa1->ifa_scope != ifa->ifa_scope) {
                inet_free_ifa(ifa);
                return -EINVAL;
```

```
                }
                ifa->ifa_flags |= IFA_F_SECONDARY;
            }
        }

        if (!(ifa->ifa_flags & IFA_F_SECONDARY)) {
            prandom_seed((__force u32) ifa->ifa_local);
            ifap = last_primary;
        }

        /*新实例插入到*ifap 指向的实例前面*/
        ifa->ifa_next = *ifap;      /*将 in_ifaddr 实例插入 in_device 实例中单链表*/
        *ifap = ifa;

        inet_hash_insert(dev_net(in_dev->dev), ifa);     /*将 in_ifaddr 实例插入散列表*/

        cancel_delayed_work(&check_lifetime_work);
        queue_delayed_work(system_power_efficient_wq, &check_lifetime_work, 0);

        rtmsg_ifa(RTM_NEWADDR, ifa, nlh, portid);     /*向用户套接字发送应答消息*/
        blocking_notifier_call_chain(&inetaddr_chain, NETDEV_UP, ifa);
                /*执行通知链 inetaddr_chain 中通知，设备 IP 地址有变化，添加路由选择表等*/
        return 0;
    }
```

__inet_insert_ifa()函数最后将执行 inetaddr_chain 通知链中的通知，其中 fib_netdev_notifier 通知实例中的回调函数将向路由选择表添加本地路由选择表项，详见下文。

●添加本地路由

前面介绍的__inet_insert_ifa()函数最后将调用执行 **inetaddr_chain** 通知链中的通知。在路由选择表初始化函数 **ip_fib_init()**函数中，调用 register_inetaddr_notifier(&**fib_inetaddr_notifier**)函数向 **inetaddr_chain** 通知链注册通知 fib_inetaddr_notifier 实例，定义如下（/net/ipv4/fib_frontend.c）：

```
static struct notifier_block fib_inetaddr_notifier = {
    .notifier_call = fib_inetaddr_event,     /*通知回调函数*/
};

fib_inetaddr_notifier 通知回调函数为 fib_inetaddr_event()，定义如下（/net/ipv4/fib_frontend.c）：
static int fib_inetaddr_event(struct notifier_block *this, unsigned long event, void *ptr)
{
    struct in_ifaddr *ifa = (struct in_ifaddr *)ptr;      /*指向 in_ifaddr 实例*/
    struct net_device *dev = ifa->ifa_dev->dev;
    struct net *net = dev_net(dev);

    switch (event) {
    case NETDEV_UP:
        fib_add_ifaddr(ifa);     /*添加本地路由表项，/net/ipv4/fib_frontend.c*/
```

```
        #ifdef CONFIG_IP_ROUTE_MULTIPATH       /*多路径路由选择*/
            fib_sync_up(dev, RTNH_F_DEAD);
        #endif
            atomic_inc(&net->ipv4.dev_addr_genid);
            rt_cache_flush(dev_net(dev));          /*net->ipv4.rt_genid 值加 1*/
            break;
        case NETDEV_DOWN:
            fib_del_ifaddr(ifa, NULL);          /*删除本地路由表项*/
            atomic_inc(&net->ipv4.dev_addr_genid);
            if (!ifa->ifa_dev->ifa_list) {
                fib_disable_ip(dev, event);
            } else {
                rt_cache_flush(dev_net(dev));
            }
            break;
        }
        return NOTIFY_DONE;
    }
```

__inet_insert_ifa()函数中调用 blocking_notifier_call_chain()函数执行 inetaddr_chain 通知链中通知时的设备事件为 NETDEV_UP，fib_inetaddr_notifier 通知回调函数 fib_inetaddr_event()中调用 **fib_add_ifaddr(ifa)** 函数向路由选择表添加本地路由选择表项，函数定义如下（/net/ipv4/fib_frontend.c）：

```
    void fib_add_ifaddr(struct in_ifaddr *ifa)
    {
        struct in_device *in_dev = ifa->ifa_dev;
        struct net_device *dev = in_dev->dev;
        struct in_ifaddr *prim = ifa;       /*in_ifaddr 实例*/
        __be32 mask = ifa->ifa_mask;       /*子网掩码*/
        __be32 addr = ifa->ifa_local;       /*本地地址*/
        __be32 prefix = ifa->ifa_address & mask;       /*目的 IP 地址与掩码相与*/

        if (ifa->ifa_flags & IFA_F_SECONDARY) {
            prim = inet_ifa_byprefix(in_dev, prefix, mask);
            ...     /*错误处理*/
        }

        fib_magic(RTM_NEWROUTE, RTN_LOCAL, addr, 32, prim);
                                    /*添加本地地址（单播）路由表项，地址类型为 RTN_LOCAL*/
        if (!(dev->flags & IFF_UP))
            return;

        /*添加广播地址路由选择表项*/
        if (ifa->ifa_broadcast && ifa->ifa_broadcast != htonl(0xFFFFFFFF))
            fib_magic(RTM_NEWROUTE, RTN_BROADCAST, ifa->ifa_broadcast, 32, prim);

        if (!ipv4_is_zeronet(prefix) && !(ifa->ifa_flags & IFA_F_SECONDARY) &&
```

```
            (prefix != addr || ifa->ifa_prefixlen < 32)) {       /*没有设置 IFA_F_SECONDARY*/
            fib_magic(RTM_NEWROUTE,dev->flags & IFF_LOOPBACK ? RTN_LOCAL :
                                        RTN_UNICAST,prefix, ifa->ifa_prefixlen, prim);

            if (ifa->ifa_prefixlen < 31) {    /**/
                fib_magic(RTM_NEWROUTE, RTN_BROADCAST, prefix, 32, prim);
                fib_magic(RTM_NEWROUTE, RTN_BROADCAST, prefix | ~mask,32, prim);
            }
        }
    }
```

fib_add_ifaddr(ifa)函数内将调用 **fib_magic()**函数为网络接口添加本地地址、广播地址等的路由选择表项，fib_magic()函数定义如下（/net/ipv4/fib_frontend.c）：

```
static void fib_magic(int cmd, int type, __be32 dst, int dst_len, struct in_ifaddr *ifa)
/*cmd：命令，此处为 RTM_NEWROUTE，type：地址类型，dst：IP 地址，dst_len：前缀长度*/
{
    struct net *net = dev_net(ifa->ifa_dev->dev);
    struct fib_table *tb;
    struct fib_config cfg = {        /*定义 fib_config 实例*/
        .fc_protocol = RTPROT_KERNEL,
        .fc_type = type,       /*地址类型，本地地址为 RTN_LOCAL*/
        .fc_dst = dst,         /*本地 IP 地址*/
        .fc_dst_len = dst_len,       /*匹配前缀长度*/
        .fc_prefsrc = ifa->ifa_local,       /*本地地址*/
        .fc_oif = ifa->ifa_dev->dev->ifindex,
        .fc_nlflags = NLM_F_CREATE | NLM_F_APPEND,
        .fc_nlinfo = {
            .nl_net = net,
        },
    };

    if (type == RTN_UNICAST)     /*查找或创建路由选择表项*/
        tb = fib_new_table(net, RT_TABLE_MAIN);       /*主表*/
    else
        tb = fib_new_table(net, RT_TABLE_LOCAL);     /*本地表，此处选本地表*/
    if (!tb)
        return;

    cfg.fc_table = tb->tb_id;

    if (type != RTN_LOCAL)
        cfg.fc_scope = RT_SCOPE_LINK;
    else
        cfg.fc_scope = RT_SCOPE_HOST;

    if (cmd == RTM_NEWROUTE)
```

```
            fib_table_insert(tb, &cfg);        /*添加路由选择表项，前面介绍过*/
        else
            fib_table_delete(tb, &cfg);
    }
```

fib_magic()函数与前面介绍的 RTM_NEWROUTE 类型消息的处理函数 inet_rtm_newroute()非常相似，所做的工作基本相同，这里就不再重复了。

需要注意的是此处添加的路由表项类型为 RTN_LOCAL，表示是本机地址。如果数据包目的 IP 地址与此表项匹配，表示数据包是发送给本机的，此数据包将投递到本机的传输层，而不是转发。

6 查找路由选择表项

前面介绍了路由选择表的创建，路由选择表项的管理（添加）等，下面我们看一下如何在路由选择表中查找数据包目的 IP 地址匹配的表项。

在路由选择中将根据数据包目的 IP 地址在路由选择表中查找匹配的表项，以确定数据包下一步走向，是由本机接收，还是通过某个网络接口发送（组播），又或者是丢弃数据包等。

■查找结果表示

在查找匹配的路由选择表项时，查找结果由 fib_result 结构体表示。flowi4 结构体用于查找过程中暂存重要的参数。

●fib_result

fib_result 结构体定义如下（/include/net/ip_fib.h）：

```
struct fib_result {
    unsigned char  prefixlen;       /*匹配前缀长度（掩码中 1 的位数）*/
    unsigned char  nh_sel;      /*查找到下一跳的编号*/
    unsigned char  type;        /*路由类型*/
    unsigned char  scope;       /*地址范围*/
    u32        tclassid;
    struct fib_info *fi;            /*指向路由选择表项 fib_info 实例*/
    struct fib_table *table;    /*指向路由选择表 fib_table 实例*/
    struct hlist_head *fa_head;   /*指向 TRIE 树中叶节点 leaf 成员（fib_alias 实例链表）*/
};
```

●flowi4

flowi4 结构体定义在/include/net/flow.h 头文件，用于保存路由查找过程中至关重要的参数：

```
struct flowi4 {
    struct flowi_common     __fl_common;
    ...                         /*__fl_common 成员中字段宏定义*/
    __be32          saddr;    /*源 IP 地址，大端序*/
    __be32          daddr;    /*目的 IP 地址，大端序*/

    union flowi_uli             uli;      /*联合体*/
    ...                         /*uli 成员中字段宏定义*/
} __attribute__((__aligned__(BITS_PER_LONG/8)));
```

flowi4 结构体中包含 flowi_common 结构体和 flowi_uli 联合体，以及源地址和目的地址等成员。

flowi_common 结构体定义如下，表示路由查找中的参数（/include/net/flow.h）：

```
struct flowi_common {
    int    flowic_oif;        /*输出网络接口编号*/
    int    flowic_iif;        /*输入网络接口编号*/
    __u32     flowic_mark;
    __u8      flowic_tos;      /*服务类型*/
    __u8      flowic_scope;    /*范围*/
    __u8      flowic_proto;    /*传输层协议类型*/
    __u8      flowic_flags;    /*标志*/
  #define   FLOWI_FLAG_ANYSRC       0x01
  #define   FLOWI_FLAG_KNOWN_NH       0x02
    __u32     flowic_secid;
};
```

flowi_uli 是一个联合体定义如下（/include/net/flow.h）：

```
union flowi_uli {
    struct {
        __be16    dport;
        __be16    sport;
    } ports;      /*端口号*/

    struct {
        __u8      type;
        __u8      code;
    } icmpt;

    struct {
        __le16    dport;
        __le16    sport;
    } dnports;

    __be32          spi;
    __be32          gre_key;

    struct {
        __u8      type;
    } mht;
};
```

■查找函数

路由（表项）查找的接口函数为 fib_lookup()，假设内核配置没有选择 IP_MULTIPLE_TABLES 选项（不支持策略路由），fib_lookup()函数定义如下（/include/net/ip_fib.h）：

static inline int **fib_lookup**(struct net *net, const struct flowi4 *flp,struct fib_result *res, unsigned int flags)
/*net：网络命名空间，flp：保存查找参数，res：保存查找结果，flags：标记*/

```
{
    struct fib_table *tb;
    int err = -ENETUNREACH;

    rcu_read_lock();

    tb = fib_get_table(net, RT_TABLE_MAIN);     /*获取主路由选择表，/include/net/ip_fib.h*/
    if (tb && !fib_table_lookup(tb, flp, res, flags | FIB_LOOKUP_NOREF))
        err = 0;

    rcu_read_unlock();
    return err;       /*成功返回 0*/
}
```

fib_lookup()函数首先获取主路由选择表，然后调用 **fib_table_lookup()**函数，在主路由选择表中查找匹配的表项。

flowi4 结构体用于向 fib_table_lookup()函数传递查找操作重要的参数，其中 flowi4.**daddr** 表示数据包目的 IP 地址，以此查找匹配的路由选择表项。在路由选择表 TRIE 树中可能存在与 daddr 完全匹配的叶节点，即叶节点中键值与目的 IP 地址相同，路由表项将流量导向一个指定的 IP 地址，而不是一个子网。TRIE 树中也可能不存存与目的 IP 地址完全匹配的叶节点，此时需要查找匹配前缀与目的 IP 地址相同的长度最长的叶节点。fib_table_lookup()函数查找到与目的 IP 地址最长匹配的叶节点后，在其下查找最合适的路由表项 fib_info 实例。

假设路由选择表中存在以下两个表项：

目的地址/前缀	掩码	网络接口	下一跳 IP 地址
192.168.2.0/24	255.255.255.0	0	192.168.1.1
192.168.2.3/32	255.255.255.255	1	192.168.3.1

在路由选择表 TRIE 树中，存在两个叶节点。第 1 个表项对应的叶节点键值为 192.168.2.0，前缀长度为 24，第 2 个表项叶节点键值为 192.168.2.3，前缀长度为 32。

对于发往 192.168.2.3 主机（接口）的数据包，其目的 IP 地址将与第 2 个表项完全匹配，查找路由选择表项时，匹配第 2 个表项对应的 TRIE 树叶节点，在其下的 fib_info 实例的下一跳信息中指示了输出网络接口 1，下一跳 IP 地址为 192.168.3.1。

对于发往 192.168.2.4 主机（接口）的数据包，表中没有与目的 IP 地址完全匹配的表项，在 TRIE 树中没有与此地址完全匹配的叶节点，因此只能查找前缀匹配最长的表项（叶节点），查找到第 1 个表项对应的 TRIE 叶节点，匹配前缀长度为 24，数据包由网络接口 0 转发。

fib_table_lookup()函数定义在/net/ipv4/fib_trie.c 文件内，代码如下：

```
int fib_table_lookup(struct fib_table *tb, const struct flowi4 *flp,struct fib_result *res, int fib_flags)
/*tb：路由选择表，flp：查找参数，res：保存查找结果，fib_flags：标记*/
{
    struct trie *t = (struct trie *) tb->tb_data;     /*TRIE 树根节点*/
#ifdef CONFIG_IP_FIB_TRIE_STATS
    struct trie_use_stats __percpu *stats = t->stats;
#endif
    const t_key key = ntohl(flp->daddr);     /*目的 IP 地址作为键值*/
    struct key_vector *n, *pn;
    struct fib_alias *fa;
```

```
    unsigned long index;
    t_key cindex;

    pn = t->kv;     /*TRIE 树根节点*/
    cindex = 0;

    n = get_child_rcu(pn, cindex);     /*TRIE 树根节点下的子节点*/
    if (!n)
        return -EAGAIN;

#ifdef CONFIG_IP_FIB_TRIE_STATS
    this_cpu_inc(stats->gets);
#endif

    /*第一步：查找最长匹配节点，可能是叶节点，也可能是内部节点*/
    for (;;) {
        index = get_cindex(key, n);     /*键值在节点 n 中的指针数组项索引值*/
        if (index >= (1ul << n->bits))   /*键值与 n 节点不匹配*/
            break;           /*跳出循环*/

        if (IS_LEAF(n))     /*目的 IP 地址与叶节点 n 键值相同，完全匹配*/
            goto found;     /*跳转至 found 处，找到了需要的叶节点*/

        if (n->slen > n->pos) {     /*键值与 n 节点（前缀）匹配，且 n 节点为内部节点*/
            pn = n;
            cindex = index;
        }

        n = get_child_rcu(n, index);     /*查找 n 节点下匹配的子节点*/
        if (unlikely(!n))     /*n 节点下匹配的子节点为 NULL*/
            goto backtrace;        /*跳转到 backtrace 处*/
    }      /*第一步查找最长匹配节点结束*/

    /*第二步：不存在与 IP 地址完全匹配的叶节点，pn 指向最长匹配的内部节点，
    *IP 地址与 pn 下的子节点 n 不匹配。*/
    for (;;) {
        struct key_vector __rcu **cptr = n->tnode;     /*指向 n 节点子节点指针数组*/

        if (unlikely(prefix_mismatch(key, n)) || (n->slen == n->pos))     /*key 与节点 n 不匹配*/
            goto backtrace;

        if (unlikely(IS_LEAF(n)))       /*n 是叶节点，n 是最长前缀匹配叶节点，跳出循环*/
            break;

        while ((n = rcu_dereference(*cptr)) == NULL) {     /*向上遍历父节点*/
    backtrace:
```

```
            #ifdef CONFIG_IP_FIB_TRIE_STATS
                if (!n)
                    this_cpu_inc(stats->null_node_hit);
            #endif
                while (!cindex) {     /**/
                    t_key pkey = pn->key;        /*父节点键值*/
                    if (IS_TRIE(pn))
                        return -EAGAIN;
                #ifdef CONFIG_IP_FIB_TRIE_STATS
                    this_cpu_inc(stats->backtrack);
                #endif
                    pn = node_parent_rcu(pn);     /*父节点*/
                    cindex = get_index(pkey, pn);
                }   /*while (!cindex)  结束*/

                cindex &= cindex - 1;
                cptr = &pn->tnode[cindex];
            } /*while ((n = rcu_dereference(*cptr)) == NULL)结束*/
        }    /*第二步结束*/

found:
        /*找到最长匹配的叶节点 n*/
        index = key ^ n->key;         /*目的 IP 地址中与键值不同的比特位*/

        /*第三步：在叶节点中查找合适的 fib_info 实例*/
        hlist_for_each_entry_rcu(fa, &n->leaf, fa_list) {   /*遍历叶节点 leaf 链表中 fib_alias 实例*/
            struct fib_info *fi = fa->fa_info;     /*fib_alias 关联的 fib_info 实例*/
            int nhsel, err;

            if ((index >= (1ul << fa->fa_slen)) &&
                ((BITS_PER_LONG > KEYLENGTH) || (fa->fa_slen != KEYLENGTH)))
                continue;         /*选择前缀匹配的 fa_alias 实例*/
            if (fa->fa_tos && fa->fa_tos != flp->flowi4_tos)
                continue;
            if (fi->fib_dead)
                continue;
            if (fa->fa_info->fib_scope < flp->flowi4_scope)
                continue;
            fib_alias_accessed(fa);     /*设置 fa->fa_state 成员 FA_S_ACCESSED 标记位*/
            err = fib_props[fa->fa_type].error;       /*路由类型错误码*/
            if (unlikely(err < 0)) {
                ...
                return err;       /*错误的路由类型，返回错误码*/
            }
            if (fi->fib_flags & RTNH_F_DEAD)
                continue;
```

```
            /*找到了匹配的 fib_info 实例*/
            for (nhsel = 0; nhsel < fi->fib_nhs; nhsel++) {     /*遍历 fib_info 实例中下一跳 fib_nh 数组*/
                const struct fib_nh *nh = &fi->fib_nh[nhsel];
                struct in_device *in_dev = __in_dev_get_rcu(nh->nh_dev); /*表示网络设备 in_device 实例*/

                if (nh->nh_flags & RTNH_F_DEAD)
                    continue;
                if (in_dev &&IN_DEV_IGNORE_ROUTES_WITH_LINKDOWN(in_dev) &&
                    nh->nh_flags & RTNH_F_LINKDOWN &&
                    !(fib_flags & FIB_LOOKUP_IGNORE_LINKSTATE))
                    continue;
                if (flp->flowi4_oif && flp->flowi4_oif != nh->nh_oif)
                    continue;

                if (!(fib_flags & FIB_LOOKUP_NOREF))
                    atomic_inc(&fi->fib_clntref);

                /*查找到合适的下一跳 fib_nh 实例，设置查找结果*/
                res->prefixlen = KEYLENGTH - fa->fa_slen;     /*匹配前缀长度（比特位数）*/
                res->nh_sel = nhsel;          /*下一跳编号*/
                res->type = fa->fa_type;      /*路由类型*/
                res->scope = fi->fib_scope;   /*范围*/
                res->fi = fi;                 /*fib_info 实例*/
                res->table = tb;              /*路由选择表*/
                res->fa_head = &n->leaf;      /*fib_alias.leaf*/
            #ifdef CONFIG_IP_FIB_TRIE_STATS
                this_cpu_inc(stats->semantic_match_passed);
            #endif
                return err;     /*成功返回 0*/
            }     /*遍历下一跳结束*/
        }/*遍历叶节点 fib_alias 实例链表结束*/
        ...
    }
```

fib_table_lookup()函数主要执行的工作如下：

（1）查找与目的地址最长匹配的 TRIE 树节点，如果最长匹配节点是叶节点（目的地址与叶节点键值相同），跳转至步骤（3），如果最长匹配节点是内部节点，则执行步骤（2）。

（2）如果最长匹配节点是内部节点，则查找前缀最长匹配的叶节点，执行步骤（3）。

（3）遍历叶节点下的 fib_alias 实例链表查找匹配的 fib_info 实例及其中的下一跳信息，赋予 res 参数指向的 fib_result 实例，用于保存查找结果。

■查询地址类型

接口函数 **inet_addr_type**()用于查询 IP 地址的（路由）类型，函数定义如下（/net/ipv4/fib_frontend.c）：

```
unsigned int inet_addr_type(struct net *net, __be32 addr)
/*net：指向网络命名空间，addr：IP 地址*/
```

```
{
    return __inet_dev_addr_type(net, NULL, addr);     /*/net/ipv4/fib_frontend.c*/
}
```

__inet_dev_addr_type()函数定义如下：

```
static inline unsigned int __inet_dev_addr_type(struct net *net,const struct net_device *dev,__be32 addr)
{
    struct flowi4          fl4 = { .daddr = addr };       /*flowi4 实例，addr 表示 IP 地址*/
    struct fib_result      res;
    unsigned int ret = RTN_BROADCAST;
    struct fib_table *local_table;

    if (ipv4_is_zeronet(addr) || ipv4_is_lbcast(addr))       /*广播地址*/
        return RTN_BROADCAST;
    if (ipv4_is_multicast(addr))        /*组播地址*/
        return RTN_MULTICAST;

    rcu_read_lock();

    local_table = fib_get_table(net, RT_TABLE_LOCAL);       /*获取路由选择表*/
    if (local_table) {
        ret = RTN_UNICAST;
        if (!fib_table_lookup(local_table, &fl4, &res, FIB_LOOKUP_NOREF)) {
            if (!dev || dev == res.fi->fib_dev)
                ret = res.type;         /*返回路由类型*/
        }
    }

    rcu_read_unlock();
    return ret;         /*返回地址类型*/
}
```

inet_addr_type()函数以 addr 为目的 IP 地址，调用 fib_table_lookup()函数在本地路由选择表中查找表项，返回查找到路由表项中的路由类型，指示地址是本地地址、单播地址、广播地址等。

7 路由选择

网络层在发送和接收通道，收到数据包后的第一件事就是以数据包目的 IP 地址执行路由选择，以确定数据包的下一步走向。路由选择子系统向发送/接收通道返回的路由选择结果由 rtable 结构体表示，结构体中的函数指针成员用于数据包的下一步处理。

■选择结果

rtable 结构体用于表示路由选择子系统返回给网络层数据包接收/发送路径的查找结果，而前面介绍的 fib_result 结构体是在路由选择表中查找表项的结果。

发送/接收通道中调用的路由选择接口函数，将调用函数 fib_lookup()在路由选择表中查找目的 IP 地址匹配的表项及下一跳 fib_nh 实例，然后在下一跳中查找是否存在缓存的且有效的 rtable 实例，如果存在，

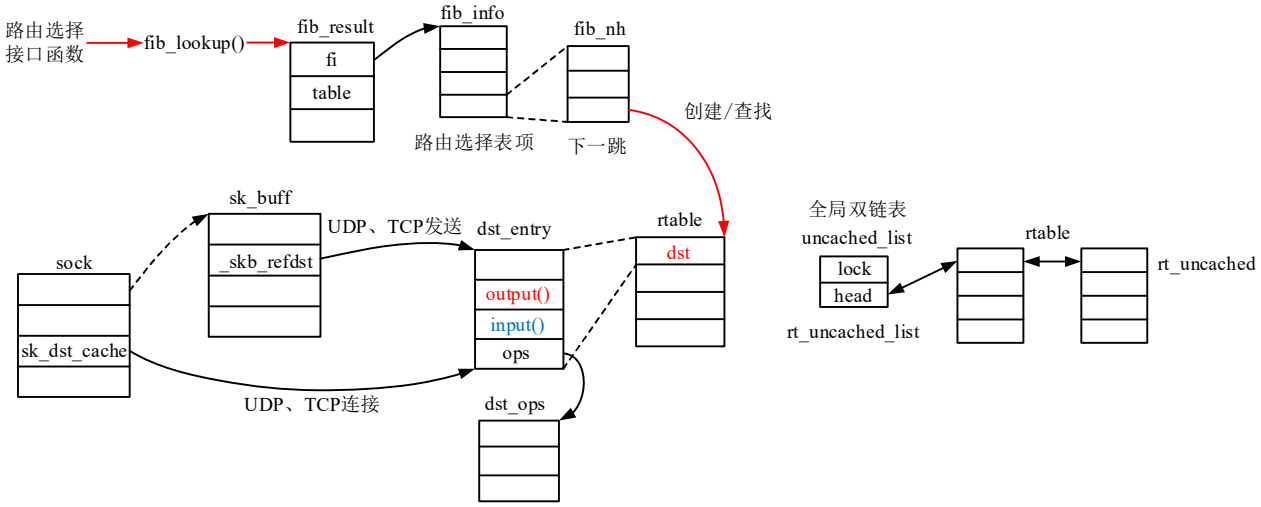
则返回此 rtable 实例指针。如果不存在则创建并设置 rtable 实例，新实例通常缓存到下一跳 fib_nh 实例中，也可能缓存到全局双链表 rt_uncached_list 中，接口函数最后返回此新建 rtable 实例指针，如下图所示。

传输层协议在建立连接的系统调用中，将调用路由选择接口函数，并将结果 rtable 实例中的 dst 成员指针（dst_entry 实例指针）赋予套接字 sock.sk_dst_cache 成员。

在发送数据包的操作中，如果套接字 sock.sk_dst_cache 成员为 NULL，将调用路由选择接口函数，并将结果 rtable 实例中的 dst 成员指针赋予 sk_buff._skb_refdst 成员。如果套接字 sock.sk_dst_cache 成员不为 NULL，则直接将 sock.sk_dst_cache 成员值赋予 sk_buff._skb_refdst 成员（dst_entry 实例指针），不需要执行路由选择操作。

在接收数据包时，将执行路由选择，路由结果 rtable.dst 实例指针直接赋予 sk_buff._skb_refdst 成员。

dst_entry 实例中的 output()和 input()函数指针分别表示了输出、输入数据包下一步处理需要调用的函数。发送/接收通道在执行完路由选择操作后，将调用 sk_buff._skb_refdst 指向 dst_entry 实例中的 output()或 input()函数处理数据包。



下面介绍路由选择结果相关的数据结构，后面将介绍路由选择函数的实现。

●rtable

rtable 结构体表示路由选择操作的结果，简称它为路由结果，以便与路由选择表项区分开来，结构体定义如下（/include/net/route.h）：

```
struct rtable {
    struct dst_entry    dst;    /*表示数据包目的地，/include/net/dst.h*/

    int             rt_genid;
    unsigned int    rt_flags;    /*标志*/
    __u16           rt_type;     /*路由类型，同 fib_info 实例中的路由类型*/
    __u8            rt_is_input;    /*一个标志，对于输入路由设置为 1*/
    __u8            rt_uses_gateway;    /*下一跳为网关还是直接路由*/
    int             rt_iif;    /*输入网络接口编号*/

    /*在邻居中的信息*/
    __be32          rt_gateway;      /*网关地址（下一跳）*/
    u32             rt_pmtu;      /*路径 MTU 值*/

    struct list_head      rt_uncached;       /*将 rtable 实例添加到 uncached_list 链表*/
    struct uncached_list  *rt_uncached_list;    /*指向带自旋锁的双链表，/net/ipv4/route.c*/
```

};

rtable 结构体主要成员简介如下：

◎**dst**：dst_entry 结构体成员，表示数据包目的地，即下一步的操作，结构体定义见下文。

◎**rt_flags**：路由选择结果标志成员，取值定义如下（/include/uapi/linux/in_route.h）：

```
#define   RTCF_DEAD          RTNH_F_DEAD      /*下一跳 fib_nh 中的标志位*/
#define   RTCF_ONLINK        RTNH_F_ONLINK

/* Obsolete flag. About to be deleted */
#define   RTCF_NOPMTUDISC    RTM_F_NOPMTUDISC

#define   RTCF_NOTIFY            0x00010000
#define   RTCF_DIRECTDST         0x00020000 /*未使用*/
#define   RTCF_REDIRECTED        0x00040000 /*需要重定向*/
#define   RTCF_TPROXY            0x00080000 /*未使用*/

#define   RTCF_FAST              0x00200000 /*未使用*/
#define   RTCF_MASQ              0x00400000 /*未使用*/
#define   RTCF_SNAT              0x00800000 /*未使用*/
#define   RTCF_DOREDIRECT        0x01000000 /*执行重定向*/
#define   RTCF_DIRECTSRC         0x04000000
#define   RTCF_DNAT              0x08000000
#define   RTCF_BROADCAST         0x10000000 /*广播数据包*/
#define   RTCF_MULTICAST         0x20000000 /*组播数据包*/
#define   RTCF_REJECT            0x40000000 /*未使用*/
#define   RTCF_LOCAL             0x80000000 /*数据包投递给本机*/
```

◎**rt_uncached**：双链表成员。内核在 ip_rt_init()函数中为每个 CPU 创建了 uncached_list 链表（带自旋锁的双链表，/net/ipv4/route.c），用于缓存 rtable 实例。rt_uncached 成员用于插入到此链表。rt_uncached_list 成员指向 uncached_list 链表头。

●dst_entry

dst_entry 结构体在路由选择结果中表示数据包下一个目的地，结构体定义在/include/net/dst.h 头文件：

```
struct dst_entry {
        struct rcu_head      rcu_head;
        struct dst_entry     *child;
        struct net_device    *dev;       /*外出网络设备*/
        struct  dst_ops      *ops;       /*dst_entry 操作结构*/
        unsigned long        _metrics;
        unsigned long        expires;
        struct dst_entry     *path;
        struct dst_entry     *from;
#ifdef CONFIG_XFRM
        struct xfrm_state    *xfrm;
#else
        void                 *__pad1;
#endif
        int                  (*input)(struct sk_buff *);       /*接收的数据包下一步处理函数*/
```

```
    int             (*output)(struct sock *sk, struct sk_buff *skb);    /*发送的数据包下一步处理函数*/

    unsigned short      flags;      /*标志*/
    ...                 /*flags 成员取值*/
    unsigned short      pending_confirm;
    short               error;
    short               obsolete;
    unsigned short      header_len;   /*各协议层报头长度*/
    unsigned short      trailer_len;    /*缓存区最后预留的空间长度*/
#ifdef CONFIG_IP_ROUTE_CLASSID
    __u32               tclassid;
#else
    __u32               __pad2;
#endif
#ifdef CONFIG_64BIT
    long            __pad_to_align_refcnt[2];
#endif
    atomic_t        __refcnt; /* client references */
    int             __use;
    unsigned long       lastuse;
    union {         /*联合体，将嵌入 dst_entry 结构体成员的实例链接成单链表*/
        struct dst_entry    *next;
        struct rtable __rcu  *rt_next;
        struct rt6_info      *rt6_next;
        struct dn_route __rcu    *dn_next;
    };
};
```

dst_entry 结构体主要成员简介如下：

◎**flags**：标志，取值定义如下：

```
  #define   DST_HOST            0x0001
  #define   DST_NOXFRM          0x0002
  #define   DST_NOPOLICY        0x0004
  #define   DST_NOHASH          0x0008
  #define   DST_NOCACHE         0x0010   /*rtable 不缓存到下一跳中，缓存至 uncached_list 双链表*/
  #define   DST_NOCOUNT             0x0020
  #define   DST_FAKE_RTABLE     0x0040
  #define   DST_XFRM_TUNNEL     0x0080
  #define   DST_XFRM_QUEUE      0x0100
```

◎**input()**：接收通道中数据包下一步处理的回调函数。

◎**output()**：发送通道中数据包下一步处理的回调函数。

◎**ops**：指向 dst_ops 结构体，包含 dst_entry 实例操作函数，结构体定义如下（/include/net/dst_ops.h）：

```
struct dst_ops {
    unsigned short      family;     /*协议簇*/
    unsigned int        gc_thresh;

    int             (*gc)(struct dst_ops *ops);
```

```
    struct dst_entry *    (*check)(struct dst_entry *, __u32 cookie);     /*检查 dst_entry 实例*/
    unsigned int          (*default_advmss)(const struct dst_entry *);
    unsigned int          (*mtu)(const struct dst_entry *);
    u32 *                 (*cow_metrics)(struct dst_entry *, unsigned long);
    void              (*destroy)(struct dst_entry *);
    void              (*ifdown)(struct dst_entry *,struct net_device *dev, int how);
    struct dst_entry *    (*negative_advice)(struct dst_entry *);
    void              (*link_failure)(struct sk_buff *);
    void              (*update_pmtu)(struct dst_entry *dst, struct sock *sk,struct sk_buff *skb, u32 mtu);
    void              (*redirect)(struct dst_entry *dst, struct sock *sk,struct sk_buff *skb);
    int               (*local_out)(struct sk_buff *skb);
    struct neighbour *    (*neigh_lookup)(const struct dst_entry *dst,struct sk_buff *skb,const void *daddr);
                                                                        /*查找邻居*/

    struct kmem_cache *kmem_cachep;       /*路由选择结果数据结构缓存，如 rtable 结构体缓存*/

    struct percpu_counter     pcpuc_entries ____cacheline_aligned_in_smp;
};
```

dst_entry 结构体通常嵌入到一个表示路由选择结果的数据结构中，如 rtable 结构体等。dst_ops 结构体包含的实际是对路由选择结果 rtable 实例的操作，其中包含 rtable 结构体的 slab 缓存。

内核在/net/ipv4/route.c 文件内定义了 dst_ops 结构体实例 **ipv4_dst_ops** 和 **ipv4_dst_blackhole_ops**，前者定义如下：

```
static struct dst_ops ipv4_dst_ops = {
    .family =       AF_INET,
    .check =        ipv4_dst_check,
    .default_advmss =   ipv4_default_advmss,
    .mtu =              ipv4_mtu,
    .cow_metrics =      ipv4_cow_metrics,
    .destroy =          ipv4_dst_destroy,
    .negative_advice =  ipv4_negative_advice,
    .link_failure =     ipv4_link_failure,
    .update_pmtu =      ip_rt_update_pmtu,
    .redirect =         ip_do_redirect,
    .local_out =        __ip_local_out,       /*发送数据包函数*/
    .neigh_lookup =     ipv4_neigh_lookup,        /*查找邻居函数*/
};
```

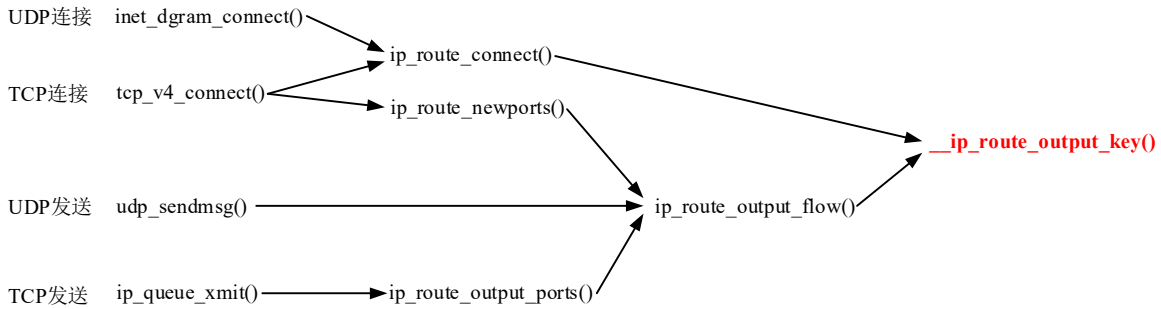
在路由子系统初始化函数 ip_rt_init()中为 ipv4_dst_ops 和 ipv4_dst_blackhole_ops 实例创建了 rtable 结构体 slab 缓存（两个实例共用），并对两个实例进行了初始化。

ip_rt_init()函数还为每个 CPU 创建了 uncached_list 链表（带自旋锁的双链表），用于缓存 rtable 实例。

■路由选择接口函数

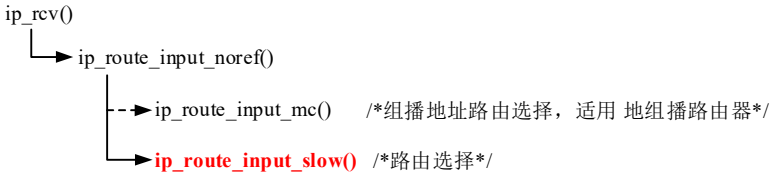
IPv4 网络层协议在发送和接收通道上都需要执行路由选择。发送通道中在建立连接的 connect()系统调

用或发送数据包时进行路由选择，下图示意了 UDP 和 TCP 在发送通道中执行路由选择的函数调用关系：



__ip_route_output_key()函数是发送通道路由选择的核心函数，此函数将查找路由选择表项，返回下一跳缓存的 rtable 实例指针，或创建并设置新 rtable 实例，并返回。

接收通道中路由选择在 IPv4 网络层协议的接收函数 ip_rcv()内进行，函数调用关系简列如下图所示：



ip_rcv()函数通常调用 ip_route_input_slow()函数执行接收通道的路由选择，并返回 rtable 实例指针。如果目的 IP 地址是组播地址，且当前机器为组播路由器，则调用 ip_route_input_mc()执行组播路由选择。

下面将分别介绍发送和接收通道路由选择函数的实现。

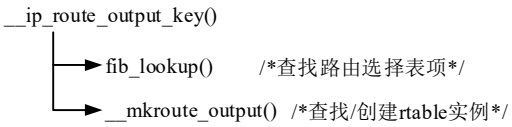
■发送通道路由选择

路由选择接口函数需要通过 flowi4 结构体传递路由选择参数，发送/接收通道需要先创建并设置 flowi4 结构体实例，然后调用路由选择接口函数，接口函数返回给发送/接收通道查找到的 rtable 实例。

发送通道各接口函数源代码请读者自行阅读，各接口函数最终调用**__ip_route_output_key()**核心函数，执行路由选择，函数返回 rtable 实例指针。

●接口函数

__ip_route_output_key()函数调用关系简列如下图所示：



fib_lookup()函数在路由选择表中查找匹配的表项及下一跳，__mkroute_output()函数在下一跳中查找缓存的（有效的）rtable 实例或创建新的 rtable 实例并缓存到下一跳中（或缓存到全局双链表）。

__ip_route_output_key()函数定义如下（/net/ipv4/route.c）：

```
struct rtable *__ip_route_output_key(struct net *net, struct flowi4 *fl4)
/*net：网络命名空间，fl4：保存查找参数*/
{
    struct net_device *dev_out = NULL;     /*输出网络设备*/
    __u8 tos = RT_FL_TOS(fl4);
    unsigned int flags = 0;
    struct fib_result res;       /*保存路由选择表的查找结果*/
```

```
struct rtable *rth;        /*保存路由选择结果*/
int orig_oif;

res.tclassid     = 0;
res.fi           = NULL;
res.table   = NULL;

orig_oif = fl4->flowi4_oif;

fl4->flowi4_iif = LOOPBACK_IFINDEX;          /*输出网络设备初始化为环回设备*/
fl4->flowi4_tos = tos & IPTOS_RT_MASK;
fl4->flowi4_scope = ((tos & RTO_ONLINK) ?RT_SCOPE_LINK : RT_SCOPE_UNIVERSE);

rcu_read_lock();
if (fl4->saddr) {       /*指定了源 IP 地址*/
      rth = ERR_PTR(-EINVAL);
      if (ipv4_is_multicast(fl4->saddr) ||ipv4_is_lbcast(fl4->saddr) ||ipv4_is_zeronet(fl4->saddr))
            goto out;     /*源地址不能是组播地址、广播地址或全零地址，/include/linux/in.h*/

      if (fl4->flowi4_oif == 0 &&(ipv4_is_multicast(fl4->daddr) ||ipv4_is_lbcast(fl4->daddr))) {
                        /*输出网络设备编号为 0，且（目的地址是广播或组播地址）*/
            dev_out = __ip_dev_find(net, fl4->saddr, false);
                          /*查找第一个具有源地址 fl4->saddr 的网络设备，/net/ipv4/devinet.c*/
            if (!dev_out)
                  goto out;

            fl4->flowi4_oif = dev_out->ifindex;   /*输出网络设备编号*/
            goto make_route;
      }

      /*指定了源 IP 地址，输出网络设备编号不为 0，或目的地址不是组播（或广播）地址*/
      if (!(fl4->flowi4_flags & FLOWI_FLAG_ANYSRC)) {
            if (!__ip_dev_find(net, fl4->saddr, false))
                  goto out;
      }
}   /*if (fl4->saddr)结束*/

if (fl4->flowi4_oif) {     /*指定了输出网络设备编号*/
      dev_out = dev_get_by_index_rcu(net, fl4->flowi4_oif);     /*查找输出设备 net_device 实例*/
      rth = ERR_PTR(-ENODEV);
      if (!dev_out)
            goto out;

      if (!(dev_out->flags & IFF_UP) || !__in_dev_get_rcu(dev_out)) {
            rth = ERR_PTR(-ENETUNREACH);     /*网络设备未开启，或没有关联 in_device 实例*/
            goto out;         /*函数返回*/
```

```
        }

        /*目的地址是组播地址，或广播地址，或 IGMP 传输层协议*/
        if (ipv4_is_local_multicast(fl4->daddr) ||ipv4_is_lbcast(fl4->daddr) ||
            fl4->flowi4_proto == IPPROTO_IGMP) {
            if (!fl4->saddr)       /*没有指定源地址*/
                fl4->saddr = inet_select_addr(dev_out, 0,RT_SCOPE_LINK);
                    /*查找输出网络设备的本地 IP 地址，赋予 fl4->saddr，/net/ipv4/devinet.c*/
            goto make_route;
        }

        if (!fl4->saddr) {     /*没有指定源地址*/
            if (ipv4_is_multicast(fl4->daddr))     /*目的地址为广播地址*/
                fl4->saddr = inet_select_addr(dev_out, 0,fl4->flowi4_scope);
            else if (!fl4->daddr)     /*没有指定目的地址*/
                fl4->saddr = inet_select_addr(dev_out, 0,RT_SCOPE_HOST);       /*设置源 IP 地址*/
        }
    }    /*if (fl4->flowi4_oif) 结束，处理指定了输出网络设备结束*/

    if (!fl4->daddr) {     /*没有指定目的地址*/
        fl4->daddr = fl4->saddr;       /*目的地址初始化为本机地址*/
        if (!fl4->daddr)
            fl4->daddr = fl4->saddr = htonl(INADDR_LOOPBACK);
                                                /*没指定本机地址，设为环回地址*/
        dev_out = net->loopback_dev;
        fl4->flowi4_oif = LOOPBACK_IFINDEX;
        res.type = RTN_LOCAL;        /*路由类型设为本地*/
        flags |= RTCF_LOCAL;
        goto make_route;
    }

    /*指定了目的 IP 地址*/
    if (fib_lookup(net, fl4, &res, 0)) {     /*查找路由选择表项*/
        /*查找路由选择表项出错，执行以下操作*/
        res.fi = NULL;
        res.table = NULL;
        if (fl4->flowi4_oif) {     /*查找到了输出网络设备*/
            if (fl4->saddr == 0)     /*如果没有指定源地址，赋予输出网络设备本机地址*/
                fl4->saddr = inet_select_addr(dev_out, 0,RT_SCOPE_LINK);
            res.type = RTN_UNICAST;    /*单播*/
            goto make_route;     /*跳转到 make_route 处*/
        }
        rth = ERR_PTR(-ENETUNREACH);
        goto out;
    }
```

```
    /*下面是处理 fib_lookup()函数查找路由选择表项成功的情形*/
    if (res.type == RTN_LOCAL) {     /*本地路由类型，目的地址为本地地址*/
        if (!fl4->saddr) {       /*没有指定源地址*/
            if (res.fi->fib_prefsrc)
                fl4->saddr = res.fi->fib_prefsrc;
            else
                fl4->saddr = fl4->daddr;
        }
        dev_out = net->loopback_dev;
        fl4->flowi4_oif = dev_out->ifindex;
        flags |= RTCF_LOCAL;
        goto make_route;
    }

    /*处理其它类型路由*/
  #ifdef CONFIG_IP_ROUTE_MULTIPATH     /*支持多路径路由选择*/
    ...
  #endif
    if (!res.prefixlen &&res.table->tb_num_default > 1 &&
                res.type == RTN_UNICAST && !fl4->flowi4_oif)     /*匹配前缀为 0 等*/
        fib_select_default(fl4, &res);     /*选择默认路由，/net/ipv4/fib_semantics.c*/

    if (!fl4->saddr)       /*源 IP 地址为 0*/
        fl4->saddr = FIB_RES_PREFSRC(net, res);     /*(res).fi->fib_prefsrc*/

    dev_out = FIB_RES_DEV(res);       /*下一跳中输出网络设备*/
    fl4->flowi4_oif = dev_out->ifindex;     /*输出网络设备编号*/

make_route:       /*根据路由选择表查找结果，创建并设置 rtable 实例*/
    rth = __mkroute_output(&res, fl4, orig_oif, dev_out, flags);       /*/net/ipv4/route.c*/

out:
    rcu_read_unlock();
    return rth;       /*返回 rtable 实例*/
}
```

查找路由选择表项及下一跳的 fib_lookup()函数前面介绍过了。下面介绍__mkroute_output()函数的实现，它用于查找或创建并设置 rtable 实例。

●查找/创建 rtable 实例

__ip_route_output_key()函数最后依据路由选择表查找结果，调用**__mkroute_output()**函数在下一跳中查找缓存的 rtable 实例，如果不存在或无效，则创建并设置 rtable 实例，最后返回 rtable 实例指针。

```
static struct rtable *__mkroute_output(const struct fib_result *res,const struct flowi4 *fl4, int orig_oif,
                                       struct net_device *dev_out,unsigned int flags)
{
    struct fib_info *fi = res->fi;       /*路由选择表项 fib_info 实例*/
    struct fib_nh_exception *fnhe;     /*下一跳例外*/
```

```
struct in_device *in_dev;        /*表示输出网络设备对应的 in_device 实例*/
u16 type = res->type;            /*目的 IP 地址类型*/
struct rtable *rth;
bool do_cache;

in_dev = __in_dev_get_rcu(dev_out);       /*输出网络设备对应的 in_device 实例*/
...     /*错误处理*/

if (likely(!IN_DEV_ROUTE_LOCALNET(in_dev)))
      if (ipv4_is_loopback(fl4->saddr) && !(dev_out->flags & IFF_LOOPBACK))
            return ERR_PTR(-EINVAL);

if (ipv4_is_lbcast(fl4->daddr))     /*目的地址为广播地址*/
      type = RTN_BROADCAST;
else if (ipv4_is_multicast(fl4->daddr))    /*组播地址*/
      type = RTN_MULTICAST;
else if (ipv4_is_zeronet(fl4->daddr))     /*全零地址*/
      return ERR_PTR(-EINVAL);

if (dev_out->flags & IFF_LOOPBACK)     /*环回设备*/
      flags |= RTCF_LOCAL;

do_cache = true;
if (type == RTN_BROADCAST) {       /*广播*/
      flags |= RTCF_BROADCAST | RTCF_LOCAL;
      fi = NULL;
} else if (type == RTN_MULTICAST) {       /*组播*/
      flags |= RTCF_MULTICAST | RTCF_LOCAL;
      if (!ip_check_mc_rcu(in_dev, fl4->daddr, fl4->saddr,fl4->flowi4_proto))
            flags &= ~RTCF_LOCAL;
      else
            do_cache = false;
      if (fi && res->prefixlen < 4)
            fi = NULL;
}

fnhe = NULL;
do_cache &= fi != NULL;
if (do_cache) {       /*获取下一跳中缓存的 rtable 实例*/
      struct rtable __rcu **prth;
      struct fib_nh *nh = &FIB_RES_NH(*res);       /*下一跳*/

      fnhe = find_exception(nh, fl4->daddr);       /*下一跳例外*/
      if (fnhe)
            prth = &fnhe->fnhe_rth_output;
      else {        /*不存在下一跳例外*/
```

```
            if (unlikely(fl4->flowi4_flags &FLOWI_FLAG_KNOWN_NH &&!(nh->nh_gw &&
                        nh->nh_scope == RT_SCOPE_LINK))) {
                do_cache = false;
                goto add;
            }
            prth = raw_cpu_ptr(nh->nh_pcpu_rth_output);
        }
        rth = rcu_dereference(*prth);       /*下一跳中缓存的 rtable 实例*/
        if (rt_cache_valid(rth)) {       /*缓存的 rtable 实例有效，返回此实例*/
            dst_hold(&rth->dst);
            return rth;
        }
    }

add:    /*下一跳中没有缓存的 rtable 实例，或无效，创建并设置 rtable 实例*/
    rth = rt_dst_alloc(dev_out,IN_DEV_CONF_GET(in_dev, NOPOLICY),
                       IN_DEV_CONF_GET(in_dev, NOXFRM),do_cache);
                                                    /*分配 rtable 实例，/net/ipv4/route.c*/
    ...    /*错误处理*/

    /*设置 rtable 实例*/
    rth->dst.output = ip_output;       /*默认设置的输出数据包函数（发送或转发）*/
    rth->rt_genid = rt_genid_ipv4(dev_net(dev_out));   /*返回 net->ipv4.rt_genid 值*/
    rth->rt_flags   = flags;     /*标志*/
    rth->rt_type    = type;        /*路由类型*/
    rth->rt_is_input = 0;
    rth->rt_iif     = orig_oif ? : 0;
    rth->rt_pmtu    = 0;
    rth->rt_gateway = 0;
    rth->rt_uses_gateway = 0;
    INIT_LIST_HEAD(&rth->rt_uncached);

    RT_CACHE_STAT_INC(out_slow_tot);
    if (flags & RTCF_LOCAL)
        rth->dst.input = ip_local_deliver;      /*投递到本机的数据包处理函数*/
    if (flags & (RTCF_BROADCAST | RTCF_MULTICAST)) {     /*广播或组播数据包*/
        if (flags & RTCF_LOCAL &&!(dev_out->flags & IFF_LOOPBACK)) {
            rth->dst.output = ip_mc_output;        /*发送广播或组播数据包函数*/
            RT_CACHE_STAT_INC(out_slow_mc);
        }
      #ifdef CONFIG_IP_MROUTE     /*本机配置为组播路由器*/
        ...        /*ip_mr_input()为转发组播数据包函数*/
      #endif
    }
    rt_set_nexthop(rth, fl4->daddr, res, fnhe, fi, type, 0);   /*缓存 rtable 实例等，/net/ipv4/route.c*/
    return rth;       /*返回 rtable 实例指针*/
```

}

__mkroute_output()函数首先在下一跳 fib_nh 实例中的发送通道 rtable 实例缓存中查找实例，如果查找到且有效则返回查找到的实例，否则创建并设置新的 rtable 实例，并调用 **rt_set_nexthop**()函数将实例添加到下一跳的 rtable 实例缓存，或插入到内核全局的 rt_uncached_list 链表（未缓存到下一跳的 rtable 实例）。rt_set_nexthop()函数中还将设置 rth->rt_gateway 成员值（下一跳 IP 地址）等。

设置 rtable 实例时，如果数据包目的地址为单播（外发）地址，rth->dst.output 设为 **ip_output()**函数指针；如果数据包目的 IP 地址是组播或广播地址时，rth->dst.output 设为 **ip_mc_output()**函数指针。

■接收通道路由选择

IPv4 网络层协议接收数据包函数 ip_rcv()中调用 **ip_route_input_noref()**函数对接收数据包目的 IP 地址执行路由选择，操作成功函数返回 0，路由选择结果 rtable.dst 实例指针赋予数据包 sk_buff._skb_refdst 成员。

ip_route_input_noref()函数定义如下（/net/ipv4/route.c）：

```
int ip_route_input_noref(struct sk_buff *skb, __be32 daddr, __be32 saddr,u8 tos, struct net_device *dev)
{
    int res;

    rcu_read_lock();
    if (ipv4_is_multicast(daddr)) {     /*目的 IP 地址为组播地址*/
        struct in_device *in_dev = __in_dev_get_rcu(dev);
        if (in_dev) {
            int our = ip_check_mc_rcu(in_dev, daddr, saddr,ip_hdr(skb)->protocol);
            if (our
            #ifdef CONFIG_IP_MROUTE
                ...     /*本机为组播路由器*/
            #endif
                ) {
                int res = ip_route_input_mc(skb, daddr, saddr,tos, dev, our);
                                                /*组播数据包路由选择，/net/ipv4/route.c*/
                rcu_read_unlock();
                return res;
            }
        }
        rcu_read_unlock();
        return -EINVAL;
    }
    res = ip_route_input_slow(skb, daddr, saddr, tos, dev);
                                                /*单播/广播数据包路由选择，/net/ipv4/route.c*/
    rcu_read_unlock();
    return res;
}
```

如果目的 IP 地址是组播地址，ip_route_input_noref()函数调用 ip_route_input_mc()函数对组播地址执行路由选择，否则调用 ip_route_input_slow()函数执行路由选择。ip_route_input_mc()函数源代码请读者自行阅读，下面将介绍 ip_route_input_slow()函数的实现。

ip_route_input_slow()函数定义如下（/net/ipv4/route.c）：

```
static int ip_route_input_slow(struct sk_buff *skb, __be32 daddr, __be32 saddr,u8 tos,
                                                                    struct net_device *dev)
/*daddr：目的 IP 地址，saddr：源 IP 地址，tos：服务类型，dev：接收数据包的网络设备*/
{
    struct fib_result res;     /*路由选择表查找结果*/
    struct in_device *in_dev = __in_dev_get_rcu(dev);
    struct flowi4   fl4;
    unsigned int    flags = 0;
    u32         itag = 0;
    struct rtable    *rth;      /*指向路由选择结果 rtable 实例*/
    int         err = -EINVAL;
    struct net       *net = dev_net(dev);
    bool do_cache;

    if (!in_dev)
        goto out;

    if (ipv4_is_multicast(saddr) || ipv4_is_lbcast(saddr))
        goto martian_source;

    res.fi = NULL;
    if (ipv4_is_lbcast(daddr) || (saddr == 0 && daddr == 0))
        goto brd_input;          /*跳转至 brd_input 处，接收广播数据包*/

    if (ipv4_is_zeronet(saddr))     /*源 IP 地址为 0*/
        goto martian_source;

    if (ipv4_is_zeronet(daddr))     /*目的 IP 地址为 0*/
        goto martian_destination;

    if (ipv4_is_loopback(daddr)) {       /*目的地址是否为环回地址*/
        if (!IN_DEV_NET_ROUTE_LOCALNET(in_dev, net))
            goto martian_destination;
    } else if (ipv4_is_loopback(saddr)) {     /*源 IP 地址为环回地址*/
        if (!IN_DEV_NET_ROUTE_LOCALNET(in_dev, net))
            goto martian_source;
    }

    /*准备执行路由选择，设置 flowi4 实例*/
    fl4.flowi4_oif = 0;
    fl4.flowi4_iif = dev->ifindex;     /*输入网络设备编号*/
    fl4.flowi4_mark = skb->mark;
    fl4.flowi4_tos = tos;
    fl4.flowi4_scope = RT_SCOPE_UNIVERSE;
    fl4.daddr = daddr;     /*目的 IP 地址*/
    fl4.saddr = saddr;     /*源 IP 地址*/
```

```
    err = fib_lookup(net, &fl4, &res, 0);       /*查找路由选择表项*/
    if (err != 0) {
        if (!IN_DEV_FORWARD(in_dev))
            err = -EHOSTUNREACH;
        goto no_route;
    }

    if (res.type == RTN_BROADCAST)      /*目的地址为广播地址*/
        goto brd_input;      /*跳至 brd_input 处，接收广播数据包*/

    if (res.type == RTN_LOCAL) {     /*目的 IP 地址为本机地址，数据包投递到本机*/
        err = fib_validate_source(skb, saddr, daddr, tos,0, dev, in_dev, &itag); /*/net/ipv4/fib_frontend.c*/
        if (err < 0)
            goto martian_source_keep_err;
        goto local_input;     /*跳至 local_input 处，处理数据包投递到本机的情形*/
    }

    /*以下处理转发数据包的情形*/
    if (!IN_DEV_FORWARD(in_dev)) {
        err = -EHOSTUNREACH;
        goto no_route;
    }
    if (res.type != RTN_UNICAST)
        goto martian_destination;

    err = ip_mkroute_input(skb, &res, &fl4, in_dev, daddr, saddr, tos);
                                                /*为转发路由创建 rtable 实例，/net/ipv4/route.c*/
out: return err;        /*函数返回*/

brd_input:      /*接收广播数据包情形*/
    if (skb->protocol != htons(ETH_P_IP))
        goto e_inval;

    if (!ipv4_is_zeronet(saddr)) {
        err = fib_validate_source(skb, saddr, 0, tos, 0, dev,in_dev, &itag);
        if (err < 0)
            goto martian_source_keep_err;
    }
    flags |= RTCF_BROADCAST;
    res.type = RTN_BROADCAST;
    RT_CACHE_STAT_INC(in_brd);

local_input:      /*数据包投递到本机的情形*/
    do_cache = false;
    if (res.fi) {      /*存在路由选择表项*/
        if (!itag) {
```

```
                rth = rcu_dereference(FIB_RES_NH(res).nh_rth_input); /*下一跳缓存的输入 rtable 实例*/
                if (rt_cache_valid(rth)) {     /*如果下一跳缓存的 rtable 实例有效*/
                    skb_dst_set_noref(skb, &rth->dst);       /*rtable.dst 成员赋予 sk_buff 实例*/
                    err = 0;
                    goto out;       /*跳至 out 处，函数返回 0*/
                }
                do_cache = true;
            }
        }

        rth = rt_dst_alloc(net->loopback_dev,IN_DEV_CONF_GET(in_dev, NOPOLICY), false, do_cache);
        if (!rth)
            goto e_nobufs;

        rth->dst.input= ip_local_deliver;
        rth->dst.output= ip_rt_bug;
#ifdef CONFIG_IP_ROUTE_CLASSID
        rth->dst.tclassid = itag;
#endif

        rth->rt_genid = rt_genid_ipv4(net);
        rth->rt_flags   = flags|RTCF_LOCAL;
        rth->rt_type    = res.type;        /*路由类型*/
        rth->rt_is_input = 1;
        rth->rt_iif     = 0;
        rth->rt_pmtu    = 0;
        rth->rt_gateway     = 0;
        rth->rt_uses_gateway = 0;
        INIT_LIST_HEAD(&rth->rt_uncached);
        RT_CACHE_STAT_INC(in_slow_tot);
        if (res.type == RTN_UNREACHABLE) {
            rth->dst.input= ip_error;
            rth->dst.error= -err;
            rth->rt_flags   &= ~RTCF_LOCAL;
        }
        if (do_cache) {       /*将 rtable 实例缓存到下一跳或全局双链表*/
            if (unlikely(!rt_cache_route(&FIB_RES_NH(res), rth))) {
                                    /*rtable 实例缓存到下一跳，释放原缓存实例*/
                rth->dst.flags |= DST_NOCACHE;
                rt_add_uncached_list(rth);       /*rtable 实例缓存到全局双链表*/
            }
        }
        skb_dst_set(skb, &rth->dst);     /*rtable.dst 成员赋予 sk_buff 实例*/
        err = 0;
        goto out;
        ...
```

}

ip_route_input_slow()函数虽然比较长，但函数并不复杂，比较容易理解，以上代码中已增加了注释。

总之，ip_route_input_noref()函数执行的结果是：

路由选择结果 rtable.dst 实例指针赋予数据包 sk_buff._skb_refdst 成员，rtable.dst.input 及 output()函数指针赋值如下：

rtable.dst.input= **ip_local_deliver** /*数据包投递到本机，含单播、广播、组播数据包*/

rtable.dst.input= **ip_forward** /*转发数据包*/

rtable.dst.output=**ip_output** /*ip_forward()函数中调用 dst.outpu()函数转发数据包*/

rtable.dst.input=ip_mr_input /*本机为组播路由器，转发组播数据包*/

组播路由 rtable 实例在 ip_route_input_mc()函数中创建并设置，转发路由 rtable 实例在__mkroute_input()函数中创建并设置，这两个函数源代码请读者自行阅读。

IPv4 接收数据包的 ip_rcv()函数在执行路由选择后，调用 sk_buff._skb_refdst 成员指向 dst_entry 实例中的 input()函数，处理数据包。也就是说所有投递到本机的数据包由 **ip_local_deliver()**函数处理，转发的数据包由 **ip_forward()**函数处理，组播路由器转发的组播数据包由 ip_mr_input()函数处理。在 12.9.5 小节将介绍 ip_local_deliver()和 ip_forward()函数的实现。

13.1.3 邻居子系统

路由选择结果中指示了数据包下一步将由哪个函数处理，对于单播外发的数据包由 ip_output()函数处理，组播或广播的数据包由函数 ip_mc_output()处理，转发的数据包由 ip_forward()函数处理。这三个函数最终都调用 ip_finish_output()函数发送数据包。

ip_finish_output()函数依据路由选择结果中输出网络接口编号、下一跳 IP 地址等信息，在邻居子系统的邻居表中查找邻居 neighbour 实例（如果不存在则创建），如果邻居已解析了物理地址（和缓存了 L2 层报头），则对数据包写入 L2 层报头，将数据包发往数据链路层。如果邻居物理地址还没解析，则调用邻居实例中 neighbour.output()函数，先解析邻居物理地址，然后再发送数据包。

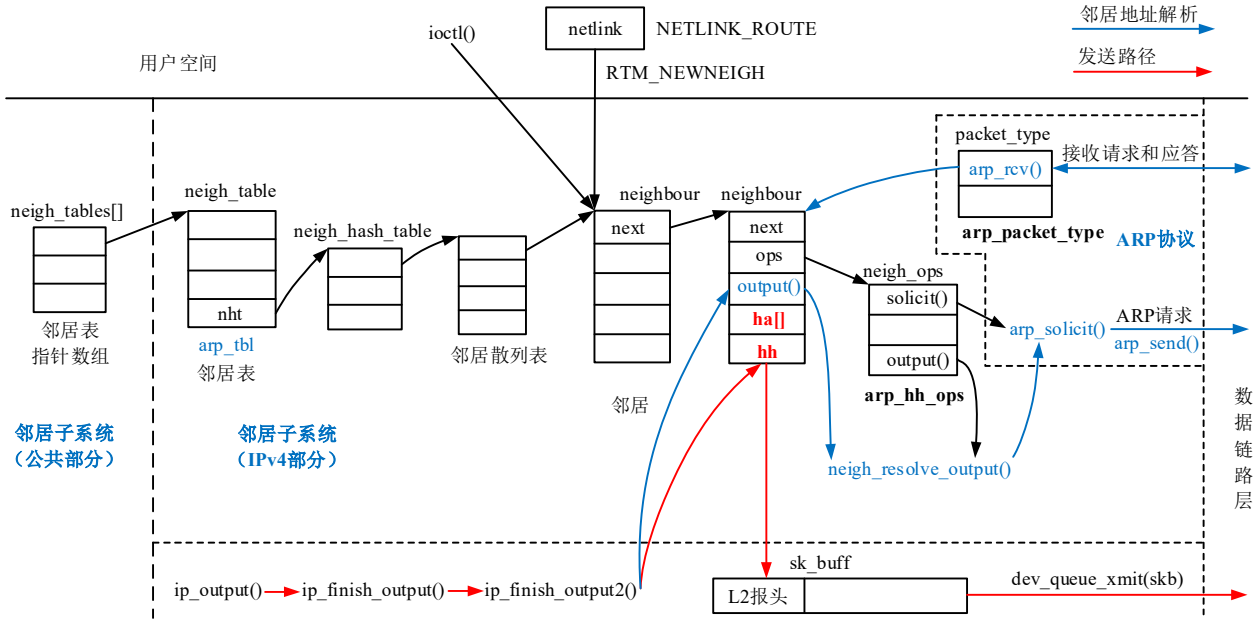
1 概述

邻居是指与本机网络接口相连的下一跳（网络节点）的网络接口。在路由选择子系统中下一跳 fib_nh 结构体记录的是下一跳网络接口的网络层信息，如 IP 地址等。邻居中记录的是下一跳网络接口的物理信息（数据链路层信息），如物理地址（MAC 地址）等，以便实现数据包在相邻节点之间传输。

邻居中缓存了下一跳物理地址（和数据链路层报头），在发送数据包时生成 L2 层报头填充至数据包，然后数据包即可发往数据链路层。

邻居子系统可适用于多个网络层协议，目前适用于三种网络层协议，分别是 IPv4、IPv6 和 Decnet 协议。邻居实例由邻居表管理，邻居子系统分别为每种适用的网络层协议维护着一个邻居表。

IPv4 网络层协议中邻居子系统框架如下图所示：



内核全局指针数组 neigh_tables[]，用于维护邻居表，目前包括 3 个数组项，分别指向 IPv4、IPv6 和 Decnet 网络层协议中的邻居表，本节只考虑 IPv4 中的邻居表（又称为 ARP 表或 ARP 缓存）。

邻居表由 neigh_table 结构体表示，邻居由 neighbour 结构体表示。邻居表中包含管理邻居实例的散列表。邻居实例中主要包含下一跳物理地址 ha[]、L2 层报头缓存 hh、解析邻居物理地址的 output()函数指针等成员。其中 output()函数指针来源于邻居关联的邻居操作结构 neigh_ops 实例。

IPv4 网络层协议中输出数据包函数最终都调用 ip_finish_output2()函数发送数据包，此函数以输出网络接口编号及下一跳 IP 地址，在 ARP 邻居表中查找邻居实例，如果实例不存在则先创建实例。如邻居物理地址已解析，即 ha[]成员中保存了邻居物理地址，hh 成员中保存了 L2 层报头，即依此向数据包写入 L2 层报头，将数据包发往数据链路层。

如果邻居物理地址尚未解析，则调用邻居实例中的 **output()**函数先解析邻居物理地址然后再发送数据包。邻居实例中的 output()函数来自于 neigh_ops 实例中的 ouput()函数，IPv4 中为 neigh_resolve_output()，此函数将数据包缓存到邻居未解析数据包队列中，然后调用 neigh_ops 实例中的 solicit()函数，即 arp_solicit()函数，发送（广播）ARP 请求（ARP 协议），用于解析邻居物理地址。邻居收到 ARP 请求后，发回 ARP 应答，内含邻居物理地址信息，本机收到 ARP 应答后，将邻居物理地址写入 ha[]成员，并生成数据链路层报头写入 hh 成员。neigh_resolve_output()函数最后发送邻居未解析数据包队列中的数据包。

在发送数据包的 ip_finish_output2()函数中，如果邻居不存在，将创建邻居实例。另外，用户进程也可以向内核发送指令直接创建邻居实例。用户进程可通过 NETLINK_ROUTE 类型 netlink 套接字、ioctl()系统调用等操作邻居表、邻居，包括创创建、删除邻居等。

邻居子系统中公共部分代码在/net/core/neighbour.c 文件内实现，主要实现邻居表的初始化、邻居的创建、添加、查找等操作函数，以及用户接口等。IPv4 中邻居子系统代码在/net/ipv4/arp.c 文件内实现，主要用于定义 IPv4 邻居表、实现 ARP 协议、实现 IPv4 邻居操作结构实例等。

2 数据结构

邻居子系统中邻居表由 neigh_table 结构体表示，邻居由 neighbour 结构体表示。neigh_table 和 neighbour 结构体不随网络层协议而异，适用于 IPv4、IPv6 等网络层协议。

■neigh_table

邻居表在 IPv4 中称为 ARP 表（又称 ARP 缓存），在 IPv6 称为 NDSIC 表。这两个表都是由 neigh_table 结构体表示，结构体定义如下（/include/net/neighbour.h）：

```
struct neigh_table {
    int             family;      /*网络层协议类型*/
    int             entry_size;    /*本邻居表下邻居实例的总长度*/
    int             key_len;      /*邻居中键值长度（IP 地址长度）*/
    __be16          protocol;      /*数据链路层报头中指示的网络层协议类型，如 IPv4 等*/
    __u32           (*hash)(const void *pkey,const struct net_device *dev,__u32 *hash_rnd);
                                                    /*为邻居计算在散列表中散列值的函数*/
    bool            (*key_eq)(const struct neighbour *, const void *pkey);
                        /*在查找邻时调用，判断指定键值是否与指定邻居中的键值相等*/
    int             (*constructor)(struct neighbour *);    /*创建邻居实例时的回调函数，构造函数*/
    int             (*pconstructor)(struct pneigh_entry *);    /*IPv6 使用的构造函数*/
    void            (*pdestructor)(struct pneigh_entry *);      /*IPv6 使用的析构函数*/
    void            (*proxy_redo)(struct sk_buff *skb);      /*处理函数指针*/
    char            *id;          /*邻居表名称，IPv4 为“arp_cache”，IPv6 为“ndisc_cache”*/
    struct neigh_parms   parms;        /*邻居参数*/
    struct list_head     parms_list;    /*管理邻居参数 neigh_parms 实例的双链表*/
    int             gc_interval;      /*回收间隔时间*/
    int             gc_thresh1;       /*回收最小阀值*/
    int             gc_thresh2;       /*回收中等阀值*/
    int             gc_thresh3;       /*回收最大阀值*/
    unsigned long        last_flush;    /*最近回收时间*/
    struct delayed_work     gc_work;    /*延时工作，异步垃圾收集处理程序*/
    struct timer_list    proxy_timer;    /*代理定时器，主机被设置为 ARP 代理时使用*/
    struct sk_buff_head  proxy_queue;   /*代理队列，由 sk_buff 实例组成*/
    atomic_t        entries;    /*邻居实例数量*/
    rwlock_t        lock;
    unsigned long        last_rand;    /*最近更新时间*/
    struct neigh_statistics     __percpu *stats;    /*统计量*/
    struct neigh_hash_table __rcu *nht;          /*指向管理邻居实例的散列表*/
    struct pneigh_entry **phash_buckets;      /*ARP 代理，保存其它网络接口 IP 地址*/
};
```

neigh_table 结构体主要成员简介如下：

●**entry_size**：邻居表中邻居实例的长度。邻居由 neighbour 结构体表示，结构体最后是邻居的键值，不同网络层协议中邻居键值长度不同，因此邻居实例的长度也不同。IPv4 中邻居键值为下一跳的 IP 地址。

●**key_len**：邻居中键值的长度。

●**hash**：计算邻居散列值的函数指针。

●**key_eq**：以键值查找邻居时，比对参数键值与邻居中键值的函数指针。

●**constructor**：邻居构造函数指针，在创建邻居时调用。函数由网络层协议实现，用于设置邻居实例。

●**nht**：指向 neigh_hash_table 结构体实例，表示邻居散列表，结构体定义如下（/include/net/neighbour.h）：

```
struct neigh_hash_table {
    struct neighbour __rcu   **hash_buckets;    /*指向邻居指针数组，构成散列表*/
    unsigned int        hash_shift;        /*2^hash_shift 表示散列表指针数组项数*/
```

```
    __u32               hash_rnd[NEIGH_NUM_HASH_RND];
                        /*随机数，用于邻居表 hash()函数中的第 3 个参数，参与邻居散列值的计算*/
    struct rcu_head     rcu;        /*包含的回调函数用于释放散列表及 neigh_hash_table 实例*/
};
```

接口函数 neigh_hash_alloc(shift)用于创建并设置 neigh_hash_table 实例及其下邻居指针数组（散列表），hash_buckets 成员指向指针数组，函数参数 shift 赋予 hash_shift 成员，指针数组项数为 2^{shift}。hash_rnd[]数组保存一组随机数，在 neigh_hash_alloc()函数中设置，它用于邻居表计算邻居散列值函数 neigh_table.hash()中的第 3 个参数，参与邻居散列值的计算。

●**parms**：邻居参数 neigh_parms 结构体成员，结构体定义如下（/include/net/neighbour.h）：

```
struct neigh_parms {
    possible_net_t net;
    struct net_device *dev;     /*本机输出网络接口*/
    struct list_head list;      /*双链表成员，将实例链入到 parms_list 双链表*/
    int    (*neigh_setup)(struct neighbour *);      /*启动函数*/
    void (*neigh_cleanup)(struct neighbour *);
    struct neigh_table *tbl;

    void *sysctl_table;
    int dead;
    atomic_t refcnt;        /*引用计数*/
    struct rcu_head rcu_head;
    int    reachable_time;
    int    data[NEIGH_VAR_DATA_MAX];      /*邻居参数数组*/
    DECLARE_BITMAP(data_state, NEIGH_VAR_DATA_MAX);
};
```

●**parms_list**：双链表成员，管理邻居参数 neigh_parms 实例。

●**phash_buckets**：指向 pneigh_entry 结构体指针数组，用于 ARP 代理。

■neighbour

neighbour 结构体定义如下（/include/net/neighbour.h）：

```
struct neighbour {
    struct neighbour __rcu   *next;     /*组成邻居队列*/
    struct neigh_table   *tbl;          /*指向邻居表*/
    struct neigh_parms *parms;      /*指向邻居参数，参数来源于网络设备 in_device 实例*/
    unsigned long       confirmed;  /*确认时间*/
    unsigned long       updated;    /*更新时间*/
    rwlock_t        lock;
    atomic_t        refcnt;     /*引用计数*/
    struct sk_buff_head    arp_queue;     /*等待邻居地址解析的 sk_buff 队列*/
    unsigned int        arp_queue_len_bytes;       /*队列长度，字节数*/
    struct timer_list      timer;       /*定时器*/
    unsigned long       used;       /*使用时间*/
    atomic_t        probes;   /*失败计数器*/
    __u8            flags;     /*标志，/include/uapi/linux/neighbour.h*/
    __u8            nud_state;   /*状态标志，/include/uapi/linux/neighbour.h*/
```

```
    __u8            type;    /*类型*/
    __u8            dead;    /*删除标志*/
    seqlock_t       ha_lock;
    unsigned char       ha[ALIGN(MAX_ADDR_LEN, sizeof(unsigned long))];
                                                /*邻居数据链路层地址（物理地址）*/
    struct hh_cache     hh;      /*数据链路层报头缓存*/
    int             (*output)(struct neighbour *, struct sk_buff *);
                                            /*发送数据包函数，来自邻居操作结构*/
    const struct neigh_ops  *ops;      /*邻居操作结构*/
    struct rcu_head         rcu;
    struct net_device   *dev;       /*本机输出网络接口（设备）*/
    u8              primary_key[0];      /*查找邻居时的键值，一般是网关地址，即下一跳 IP 地址*/
};
```

neighbour 结构体主要成员简介如下：

●**nud_state**：邻居状态，取值定义如下（/include/uapi/linux/neighbour.h）：

```
#define  NUD_INCOMPLETE  0x01    /*未完成状态*/
#define  NUD_REACHABLE   0x02    /*邻居可达状态*/
#define  NUD_STALE       0x04    /*过期状态*/
#define  NUD_DELAY       0x08    /*延迟状态*/
#define  NUD_PROBE       0x10    /*探测状态（发送 ARP 请求）*/
#define  NUD_FAILED      0x20    /*失败状态*/
/* Dummy states */
#define  NUD_NOARP       0x40    /*不需要解析*/
#define  NUD_PERMANENT   0x80    /*静态 ARP*/
#define  NUD_NONE        0x00    /*初始状态*/

/*标志合集，/include/net/neighbour.h*/
#define  NUD_IN_TIMER (NUD_INCOMPLETE|NUD_REACHABLE|NUD_DELAY|NUD_PROBE)
#define  NUD_VALID
(NUD_PERMANENT|NUD_NOARP|NUD_REACHABLE|NUD_PROBE|NUD_STALE|NUD_DELAY)
#define  NUD_CONNECTED   (NUD_PERMANENT|NUD_NOARP|NUD_REACHABLE)
```

●**flags**：邻居标志，取值定义如下（/include/uapi/linux/neighbour.h）：

```
#define  NTF_USE         0x01    /*用户添加邻居，由 ARP 解析物理地址*/
#define  NTF_SELF        0x02
#define  NTF_MASTER      0x04
#define  NTF_PROXY       0x08     /* == ATF_PUBL */
#define  NTF_EXT_LEARNED    0x10
#define  NTF_ROUTER          0x80
```

●**arp_queue**：发送数据包队列，当邻居物理地址未解析时，将发送数据包添加到本队列，待发送 ARP 请求，解析了邻居物理地址后再发送数据包。

●**timer**：定时器，在定时器到期回调函数中发送 ARP 请求。

●**ha[]**：缓存邻居物理地址。

●**hh**：hh_cache 结构体实例，缓存 L2 层报头，结构体定义如下（/include/linux/netdevice.h）：

```
struct hh_cache {
```

```
    u16         hh_len;       /*报头长度*/
    u16         __pad;
    seqlock_t hh_lock;
    ...     /*宏定义*/
    unsigned long hh_data[HH_DATA_ALIGN(LL_MAX_HEADER) / sizeof(long)];     /*缓存报头*/
};
```

●**ops**：邻居操作结构 neigh_ops 实例指针，在邻居构造函数中赋值，实例由网络层协议定义。

●**primary_key[0]**：邻居的键值，为邻居 IP 地址。

●**output()**：发送数据包函数，来自 neigh_ops 实例。若邻居物理地址未解析，函数内将发送 ARP 请求，若邻居地址已解析，则填充 L2 层报头后将数据包发送到数据链路层。

●**ops**：指向邻居操作结构 neigh_ops 结构体实例，结构体定义见下文。

■neigh_ops

neigh_ops 表示邻居操作结构，结构体定义如下（/include/net/neighbour.h）：

```
struct neigh_ops {
    int             family;       /*网络层协议类型，如 AF_INET*/
    void            (*solicit)(struct neighbour *, struct sk_buff *);       /*发送邻居请求，解析物理地址*/
    void            (*error_report)(struct neighbour *, struct sk_buff *);
    int             (*output)(struct neighbour *, struct sk_buff *);
                                    /*邻居 IP 地址已知，但物理地址未知时，发送数据包*/
    int             (*connected_output)(struct neighbour *, struct sk_buff *);
                                    /*邻居物理地址已解析时，发送数据包*/
};
```

neigh_ops 结构体实例需由网络层协议实现，因为不同的网络层协议操作函数不同。neigh_ops 结构体中主要包含函数指针，各函数实现的功能简介如下：

solicit()：解析邻居物理地址的函数，例如：IPv4 中用于发送 ARP 请求。

output()：用于在邻居物理地址未解析时发送数据包。此函数会将数据包添加到邻居未解析数据包队列，然后调用 solicit()函数解析邻居物理地址，邻居物理地址解析后再发送数据包。新创建邻居时（物理地址未解析时）此函数指针赋予邻居 output()函数指针成员。

connected_ouput()：函数用于邻居物理地址已解析时（邻居中未缓存 L2 层报头）发送数据包。在解析了邻居物理地址（未缓存 L2 层报头）时，此函数指针赋予邻居 output()函数指针成员。

3 初始化

邻居子系统的初始化分为两步，一是公共部分的初始化，二是特定于网络层协议的初始化。公共部分的初始化由 neigh_init()函数完成，特定于 IPv4 的初始化由 arp_init()函数完成，下面分别介绍这两个函数的实现。

■公共部分初始化

邻居子系统公共部分初始化函数 neigh_init()定义如下（/net/core/neighbour.c）：

```
static int __init neigh_init(void)
{
    rtnl_register(PF_UNSPEC, RTM_NEWNEIGH, neigh_add, NULL, NULL);
                                                    /*添加邻居消息处理函数*/
```

```
    rtnl_register(PF_UNSPEC, RTM_DELNEIGH, neigh_delete, NULL, NULL);
    rtnl_register(PF_UNSPEC, RTM_GETNEIGH, NULL, neigh_dump_info, NULL);

    rtnl_register(PF_UNSPEC, RTM_GETNEIGHTBL, NULL, neightbl_dump_info,NULL);
    rtnl_register(PF_UNSPEC, RTM_SETNEIGHTBL, neightbl_set, NULL, NULL);   /*设置邻居表*/
    return 0;
}
subsys_initcall(neigh_init);      /*内核启动阶段调用此函数*/
```

用户可以通过 NETLINK_ROUTE 类型 netlink 套接字操作邻居表和邻居。neigh_init()函数主要是为添加邻居、删除邻居、获取邻居表、设置邻居表等类型消息设置处理函数。这些处理函数适用于所有网络层协议中的邻居表。后面将会以 RTM_NEWNEIGH 消息为例，介绍添加邻居消息处理函数的实现。

■IPv4 初始化

IPv4 协议层在/net/ipv4/arp.c 文件内定义了邻居表 neigh_table 结构体实例 **arp_tbl**，简列如下：

```
struct neigh_table arp_tbl = {
    .family    = AF_INET,
    .key_len  = 4,        /*键值长度，4 字节*/
    .protocol = cpu_to_be16(ETH_P_IP),    /*网络层协议*/
    .hash           = arp_hash,        /*为邻居计算散列值的函数*/
    .key_eq         = arp_key_eq,   /*键值（IP 地址）比对函数，检查是否相等，在查找邻居时调用*/
    .constructor    = arp_constructor,   /*邻居构造函数，创建邻居时调用，见下文*/
    .proxy_redo    = parp_redo,
    .id         = "arp_cache",         /*邻居表名称*/
    .parms          = {          /*邻居参数*/
        .tbl              = &arp_tbl,
        .reachable_time     = 30 * HZ,
        .data= {
            ...     /*参数数组*/
        },
    },
    .gc_interval    = 30 * HZ,
    .gc_thresh1     = 128,
    .gc_thresh2     = 512,
    .gc_thresh3     = 1024,
};
```

初始化函数 **arp_init()**用于 IPv4 邻居子系统初始化，此函数由 inet_init()函数调用，arp_init()函数定义如下（/net/ipv4/arp.c）：

```
void __init arp_init(void)
{
    neigh_table_init(NEIGH_ARP_TABLE, &arp_tbl); /*初始化 arp_tbl 邻居表，/net/core/neighbour.c*/

    dev_add_pack(&arp_packet_type);
      /*注册 ARP 协议（网络层协议）packet_type 实例，用于接收 ARP 请求和发送应答，见下文*/
```

```
    arp_proc_init();        /*完成 ARP 在 procfs 中的初始化*/
#ifdef CONFIG_SYSCTL
    neigh_sysctl_register(NULL, &arp_tbl.parms, NULL);
#endif
    register_netdevice_notifier(&arp_netdev_notifier);     /*/net/core/dev.c*/
                                        /*向 netdev_chain 通知链注册通知 arp_netdev_notifier 实例*/
}
```

arp_init()函数内调用 neigh_table_init()函数初始化邻居表 arp_tbl 实例（ARP 表），函数代码见下文；arp_init()函数随后注册 ARP 协议定义的 packet_type 结构体实例 arp_packet_type，用于接收 ARP 请求和发送 ARP 应答，详见下文；调用 arp_proc_init()完成邻居子系统在 procfs 中的初始化，用于创建文件；最后向 netdev_chain 通知链注册通知 arp_netdev_notifier 实例，通知回调函数用于在输出网络接口物理地址变化时，修改邻居实例中的参数（状态）。

下面看一下初始化邻居表 neigh_table_init()函数的实现，arp_packet_type 实例中的接收数据包函数后面将详细介绍。

●初始化邻居表

neigh_table_init()是公共的初始化邻居表函数，定义如下（/net/core/neighbour.c）：

```
void neigh_table_init(int index, struct neigh_table *tbl)
{
    unsigned long now = jiffies;
    unsigned long phsize;

    INIT_LIST_HEAD(&tbl->parms_list);
    list_add(&tbl->parms.list, &tbl->parms_list);       /*将内嵌的 parms 成员添加到 parms_list 双链表*/
    write_pnet(&tbl->parms.net, &init_net);
    atomic_set(&tbl->parms.refcnt, 1);          /*参数引用计数设为 1*/
    tbl->parms.reachable_time =
                neigh_rand_reach_time(NEIGH_VAR(&tbl->parms, BASE_REACHABLE_TIME));

    tbl->stats = alloc_percpu(struct neigh_statistics);     /*分配统计量数据结构*/
    ...     /*错误处理*/

  #ifdef CONFIG_PROC_FS
    if (!proc_create_data(tbl->id, 0, init_net.proc_net_stat,&neigh_stat_seq_fops, tbl))
        panic("cannot create neighbour proc dir entry");
  #endif

    RCU_INIT_POINTER(tbl->nht, neigh_hash_alloc(3));       /*分配管理邻居实例的散列表*/

    phsize = (PNEIGH_HASHMASK + 1) * sizeof(struct pneigh_entry *);
    tbl->phash_buckets = kzalloc(phsize, GFP_KERNEL);     /*分配 pneigh_entry 结构体指针数组*/

    ...     /*错误处理*/

    if (!tbl->entry_size)
```

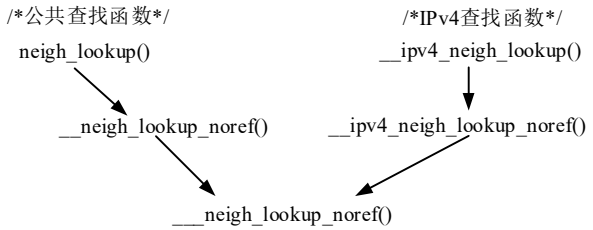
```
        tbl->entry_size = ALIGN(offsetof(struct neighbour, primary_key) +
                                                tbl->key_len, NEIGH_PRIV_ALIGN);   /*邻居结构长度*/
    else
        WARN_ON(tbl->entry_size % NEIGH_PRIV_ALIGN);

    rwlock_init(&tbl->lock);
    INIT_DEFERRABLE_WORK(&tbl->gc_work, neigh_periodic_work);
    queue_delayed_work(system_power_efficient_wq, &tbl->gc_work,tbl->parms.reachable_time);
    setup_timer(&tbl->proxy_timer, neigh_proxy_process, (unsigned long)tbl);
    skb_queue_head_init_class(&tbl->proxy_queue,&neigh_table_proxy_queue_class);

    tbl->last_flush = now;
    tbl->last_rand = now + tbl->parms.reachable_time * 20;

    neigh_tables[index] = tbl;        /*全局指针数组项指向邻居表实例*/
}
```

4 邻居操作

IPv4 发送数据包的路径中，在执行完路由选择后，将根据输出网络接口编号和下一跳 IP 地址，在邻居表中查找表示下一跳的邻居实例，如果不存在则创建。下面将先介绍邻居的查找操作，然后介绍创建邻居操作。

■查找邻居

在邻居表中查找邻居需要指定键值（邻居 IP 地址）和本机输出网络设备的 net_device 实例。邻居子系统公共层代码提供了查找邻居的接口函数 **neigh_lookup()**（/net/core/neighbour.c），IPv4 也提供了查找邻居的接口函数**__ipv4_neigh_lookup()**，函数调用关系如下图所示：



查找函数都比较简单，下面以 IPv4 查找函数为例，介绍其实现，函数定义在/include/net/arp.h 头文件：

```
static inline struct neighbour *__ipv4_neigh_lookup(struct net_device *dev, u32 key)
/*dev：输出网络设备，key：下一跳 IP 地址*/
{
    struct neighbour *n;

    rcu_read_lock_bh();
    n = __ipv4_neigh_lookup_noref(dev, key);     /*/include/net/arp.h*/
    if (n && !atomic_inc_not_zero(&n->refcnt))
        n = NULL;
    rcu_read_unlock_bh();
    return n;
```

```
}
```

__ipv4_neigh_lookup_noref()函数定义如下：

```
static inline struct neighbour *__ipv4_neigh_lookup_noref(struct net_device *dev, u32 key)
{
    return ___neigh_lookup_noref(&arp_tbl, neigh_key_eq32, arp_hashfn, &key, dev);
                                        /*在邻居表中查找邻居，/include/net/neighbour.h*/
}
```

___neigh_lookup_noref()函数是一个公共的函数，它在邻居表中的散列表中查找键值与 key 匹配（调用邻居表中 tbl->key_eq()函数检查是否匹配），且输出网络设备为 dev 的邻居实例，arp_hashfn()函数用于计算散列值，若查找到了匹配的邻居则返回邻居实例指针，否则返回 NULL。

■创建邻居

创建邻居的公共接口函数为 neigh_create()，定义如下（/include/net/neighbour.h）：

```
static inline struct neighbour *neigh_create(struct neigh_table *tbl,const void *pkey,struct net_device *dev)
/*tbl：指向邻居表，pkey：指向新建邻居的键值（IP 地址），dev：指向网络设备*/
{
    return __neigh_create(tbl, pkey, dev, true);
}
```

函数内调用__neigh_create()函数完成创建邻居操作，函数定义如下（/net/core/neighbour.c）：

```
struct neighbour *__neigh_create(struct neigh_table *tbl, const void *pkey,struct net_device *dev,
                                                                    bool want_ref)
/*tbl：邻居表，pkey：邻居（下一跳）IP 地址，dev：输出网络设备，want_ref：此处为 true*/
{
    u32 hash_val;
    int key_len = tbl->key_len;
    int error;
    struct neighbour *n1, *rc, *n = neigh_alloc(tbl, dev);
                                    /*分配邻居实例并初始化，/net/core/neighbour.c*/
    struct neigh_hash_table *nht;     /*散列表*/
    ...     /*错误处理*/
    memcpy(n->primary_key, pkey, key_len);   /*复制键值至邻居，IP 地址*/
    n->dev = dev;          /*输出网络设备赋予邻居实例*/
    dev_hold(dev);

    /*调用邻居表定义的邻居构造函数，ARP 表为 arp_constructor()函数*/
    if (tbl->constructor &&  (error = tbl->constructor(n)) < 0) {
        ...     /*错误处理*/
    }

     /*调用网络设备操作结构定义的邻居构造函数*/
    if (dev->netdev_ops->ndo_neigh_construct) {
        error = dev->netdev_ops->ndo_neigh_construct(n);
        ...       /*错误处理*/
```

```
    }

    /*调用网络设备（网络层设备表示）定义的启动函数，邻居参数来源于网络设备 in_device 实例*/
    if (n->parms->neigh_setup &&(error = n->parms->neigh_setup(n)) < 0) {
        ...   /*错误处理*/
    }

    n->confirmed = jiffies - (NEIGH_VAR(n->parms, BASE_REACHABLE_TIME) << 1);

    write_lock_bh(&tbl->lock);
    nht = rcu_dereference_protected(tbl->nht,lockdep_is_held(&tbl->lock));      /*邻居散列表*/

    if (atomic_read(&tbl->entries) > (1 << nht->hash_shift))   /*扩展邻居散列表*/
        nht = neigh_hash_grow(tbl, nht->hash_shift + 1);

    hash_val = tbl->hash(pkey, dev, nht->hash_rnd) >> (32 - nht->hash_shift);   /*计算邻居散列值*/

    if (n->parms->dead) {
        ...   /*错误处理*/
    }
    /*以下代码将邻居实例插入散列表*/
    for (n1 = rcu_dereference_protected(nht->hash_buckets[hash_val],lockdep_is_held(&tbl->lock));
         n1 != NULL;
         n1 = rcu_dereference_protected(n1->next,lockdep_is_held(&tbl->lock))) {
         if (dev == n1->dev && !memcmp(n1->primary_key, pkey, key_len)) {
             if (want_ref)
                 neigh_hold(n1);
             rc = n1;
             goto out_tbl_unlock;
         }
    }

    n->dead = 0;
    if (want_ref)
        neigh_hold(n);
    rcu_assign_pointer(n->next,rcu_dereference_protected(nht->hash_buckets[hash_val],
                                                         lockdep_is_held(&tbl->lock)));
    rcu_assign_pointer(nht->hash_buckets[hash_val], n);      /*新实例插入散列链表头部*/
    write_unlock_bh(&tbl->lock);
    neigh_dbg(2, "neigh %p is created\n", n);
    rc = n;
out:
    return rc;      /*返新邻居实例指针*/
    ...
}
```

__neigh_create()函数主要执行以下工作：

（1）调用 neigh_alloc(tbl, dev)函数创建并初始化邻居实例。
（2）调用网络层协议（邻居表）定义的邻居构造函数 tbl->constructor()。
（3）调用网络设备操作结构中定义的邻居构造函数 dev->netdev_ops->ndo_neigh_construct(n)。
（4）调用网络设备 in_devcie 实例中邻居参数中的 n->parms->neigh_setup(n)函数。
（5）调用邻居表中的 hash()函数为邻居计算散列值，将邻居实例插入散列表。
__neigh_create()函数返回新建邻居实例指针。
下面介绍分配邻居实例 neigh_alloc(tbl, dev)函数，以及 IPv4 构造邻居函数的代码。

●分配邻居

公共接口函数 neigh_alloc(tbl, dev)用于分配邻居实例，函数定义在/net/core/neighbour.c 文件内，代码简列如下：

```
static struct neighbour *neigh_alloc(struct neigh_table *tbl, struct net_device *dev)
{
    struct neighbour *n = NULL;
    unsigned long now = jiffies;
    int entries;

    entries = atomic_inc_return(&tbl->entries) - 1;
    /*调用同步垃圾收集器（清理 ARP 表，删除无用项）*/
    if (entries >= tbl->gc_thresh3 ||(entries >= tbl->gc_thresh2 &&
                        time_after(now, tbl->last_flush + 5 * HZ))) {
        if (!neigh_forced_gc(tbl) &&entries >= tbl->gc_thresh3)    /*/net/core/neighbour.c*/
            goto out_entries;
    }
    /*如果邻居数量大于 gc_thresh3（默认为 1024），或者邻居数量大于 gc_thresh2（默认为 512），
    *且最后一次的刷新频率高于 5Hz，将调用同步垃极收集器。如果运行 neigh_forced_gc()函数后，
    *邻居数量依然大于 gc_thresh3（1024），将不分配邻居实例并返回 NULL。*/

    n = kzalloc(tbl->entry_size + dev->neigh_priv_len, GFP_ATOMIC);
                        /*分配邻居实例，dev->neigh_priv_len 表示邻居中网络设备私有数据大小*/
    ...    /*错误处理*/
    __skb_queue_head_init(&n->arp_queue);      /*初始化数据包队列头*/
    rwlock_init(&n->lock);
    seqlock_init(&n->ha_lock);
    n->updated        = n->used = now;
    n->nud_state      = NUD_NONE;      /*邻居状态初始值*/
    n->output    = neigh_blackhole;      /*初始化 output()函数*/
    seqlock_init(&n->hh.hh_lock);
    n->parms    = neigh_parms_clone(&tbl->parms);      /*复制邻居表中的邻居参数*/
    setup_timer(&n->timer, neigh_timer_handler, (unsigned long)n);   /*设置定时器，但未激活*/
                                        /*邻居定时器处理函数为 neigh_timer_handler()*/
    NEIGH_CACHE_STAT_INC(tbl, allocs);
    n->tbl              = tbl;      /*指向邻居表*/
    atomic_set(&n->refcnt, 1);     /*引用计数加 1*/
    n->dead             = 1;
out:
```

```
    return n;     /*返回新建邻居实例指针*/
    ...
}
```

●构造邻居

在创建邻居的操作中，将调用邻居表中定义的构造函数，完成特定于网络层协议的邻居构造操作。IPv4 邻居表 arp_tbl 实例中的 constructor()构造函数为 **arp_constructor()**。

arp_constructor()函数定义如下（/net/ipv4/arp.c）：

```
static int arp_constructor(struct neighbour *neigh)
{
    __be32 addr = *(__be32 *)neigh->primary_key;     /*邻居 IP 地址*/
    struct net_device *dev = neigh->dev;       /*网络设备*/
    struct in_device *in_dev;
    struct neigh_parms *parms;     /*指向邻居参数*/

    rcu_read_lock();
    in_dev = __in_dev_get_rcu(dev);     /*网络设备在 IPv4 中的表示 in_device 实例*/
    ...        /*错误处理*/
    neigh->type = inet_addr_type(dev_net(dev), addr);     /*邻居 IP 地址类型*/

    parms = in_dev->arp_parms;     /*指向 in_device 实例中定义的邻居参数*/
    __neigh_parms_put(neigh->parms);
    neigh->parms = neigh_parms_clone(parms);    /*复制 in_device 实例中邻居参数到邻居实例*/
    rcu_read_unlock();

    if (!dev->header_ops) {        /*如果网络设备没有定义链路层报头操作结构，一般会定义*/
        neigh->nud_state = NUD_NOARP;
        neigh->ops = &arp_direct_ops;
        neigh->output = neigh_direct_output;
    } else {          /*网络设备定义了链路层报头操作结构*/
        if (neigh->type == RTN_MULTICAST) {       /*邻居 IP 地址为组播地址*/
            neigh->nud_state = NUD_NOARP;
            arp_mc_map(addr, neigh->ha, dev, 1);
        } else if (dev->flags & (IFF_NOARP | IFF_LOOPBACK)) {    /*设备关闭了 ARP 或环回设备*/
            neigh->nud_state = NUD_NOARP;
            memcpy(neigh->ha, dev->dev_addr, dev->addr_len);
        } else if (neigh->type == RTN_BROADCAST ||(dev->flags & IFF_POINTOPOINT)) {
            neigh->nud_state = NUD_NOARP;         /*邻居为广播或点对点地址*/
            memcpy(neigh->ha, dev->broadcast, dev->addr_len);
        }

        if (dev->header_ops->cache) /*如果定义了由邻居生成 L2 报头缓存的函数，以太网定义了*/
            neigh->ops = &arp_hh_ops;     /*邻居操作结构实例，具有 L2 报头缓存，/net/ipv4/arp.c*/
        else
            neigh->ops = &arp_generic_ops;
                                    /*邻居操作结构实例，不具有缓存 L2 报头，/net/ipv4/arp.c*/
```

```
            if (neigh->nud_state & NUD_VALID)     /*邻居状态初始值为 NUD_NONE*/
                neigh->output = neigh->ops->connected_output;   /*如邻居已连接，物理地址已解析*/
            else
                neigh->output = neigh->ops->output;   /*邻居地址未解析时，对邻居 output()成员赋值*/
        }
        return 0;
    }
```

arp_constructor()函数的主要工作是对邻居实例 ops 和 output 成员赋值。如果网络设备关联的 L2 层报头操作结构 header_ops 实例中定义了 cache()函数（可以生成 L2 报头缓存写入邻居实例，以太网报头操作结构中定义了此函数），邻居操作结构指针 ops 成员赋值为 **arp_hh_ops** 实例指针。如果 header_ops 实例中没有定义 cache()函数，则 ops 成员赋值为 **arp_generic_ops** 实例指针。**arp_hh_ops 和 arp_generic_ops** 实例后面将会介绍。arp_constructor()函数中邻居实例 output 成员赋值为 neigh->ops->output，即邻居操作结构实例中的 output()函数。

■查找/创建邻居

前面介绍了查找、创建邻居的接口函数。邻居子系统公共部分还定义了查找并创建邻居的接口函数：**__neigh_lookup()、__neigh_lookup_errno()。**这两个函数封装了 neigh_lookup()和 neigh_create()函数，这两个函数都定义在/include/net/neighbour.h 头文件。

__neigh_lookup()函数最后一个参数中用于指示是否在没找到邻居实例时创建邻居，函数定义如下：

```
static inline struct neighbour *__neigh_lookup(struct neigh_table *tbl, const void *pkey,
                                               struct net_device *dev, int creat)
{
    struct neighbour *n = neigh_lookup(tbl, pkey, dev);

    if (n || !creat)     /*如果 creat 为 false，返回 n*/
        return n;

    n = neigh_create(tbl, pkey, dev);        /*creat 为 true，没找到邻居则创建*/
    return IS_ERR(n) ? NULL : n;
}
```

__neigh_lookup_errno()函数在没查找到邻居实例时自动创建新实例，不需要参数指示，函数定义如下：

```
static inline struct neighbour *__neigh_lookup_errno(struct neigh_table *tbl, const void *pkey,
                                                     struct net_device *dev)
{
    struct neighbour *n = neigh_lookup(tbl, pkey, dev);
    if (n)
        return n;
    return neigh_create(tbl, pkey, dev);
}
```

5 输出数据包

下面介绍在 IPv4 发送数据包路径中，数据包到达邻居子系统后，如何处理。在路由选择子系统中，数

据包由 ip_output()函数发出，数据包传输路径如下：

ip_output()->ip_finish_output()->ip_finish_output2()

ip_finish_output2()函数将根据输出网络设备和下一跳 IP 地址，在邻居子系统中查找表示下一跳的邻居实例，如果邻居实例尚不存在则创建。

ip_finish_output2()函数代码简列如下（/net/ipv4/ip_output.c）：

```
static int ip_finish_output2(struct sock *sk, struct sk_buff *skb)
{
    ...
    nexthop = (__force u32) rt_nexthop(rt, ip_hdr(skb)->daddr);   /*下一跳 IP 地址，/include/net/route.h*/
    neigh = __ipv4_neigh_lookup_noref(dev, nexthop);     /*查找邻居实例*/
    if (unlikely(!neigh))
        neigh = __neigh_create(&arp_tbl, &nexthop, dev, false);   /*没有找到邻居实例，则创建实例*/
    if (!IS_ERR(neigh)) {
        int res = dst_neigh_output(dst, neigh, skb);     /*输出数据包，/include/net/dst.h*/
        rcu_read_unlock_bh();
        return res;
    }
    ...
}
```

ip_finish_output2()函数通过网络设备 net_device 实例和下一跳 IP 地址，调用__ipv4_neigh_lookup_noref()函数在邻居表中查找邻居实例，若未找到则调用__neigh_create()函数创建邻居，最后调用 dst_neigh_output()函数继续处理数据包。dst_neigh_output()函数定义如下（/include/net/dst.h）：

```
static inline int dst_neigh_output(struct dst_entry *dst, struct neighbour *n,struct sk_buff *skb)
{
    const struct hh_cache *hh;      /*指向 L2 层报头缓存结构*/

    if (dst->pending_confirm) {
        unsigned long now = jiffies;
        dst->pending_confirm = 0;
        if (n->confirmed != now)
            n->confirmed = now;
    }

    hh = &n->hh;     /*指向邻居实例中的 L2 层报头缓存结构*/
    if ((n->nud_state & NUD_CONNECTED) && hh->hh_len)     /*邻居缓存了 L2 层报头*/
        return neigh_hh_output(hh, skb);     /*直接发送数据包，/include/net/neighbour.h*/
    else        /*邻居没有缓存 L2 层报头，调用邻居中的 output()函数*/
        return n->output(n, skb);
}
```

在邻居结构中具有缓存邻居物理地址和 L2 层报头的成员，如下所示：

```
struct neighbour {
    ...
    unsigned char       ha[ALIGN(MAX_ADDR_LEN, sizeof(unsigned long))];   /*缓存邻居物理地址*/
    struct hh_cache     hh;      /*缓存数据链路层报头*/
```

```
    int             (*output)(struct neighbour *, struct sk_buff *);
                                /*hh 示缓存 L2 报头时，发送数据包的函数，来自邻居操作结构*/
    ...
};
```

如果邻居实例中已经缓存了 L2 层报头，ip_finish_output2()函数将调用 neigh_hh_output(hh, skb)函数发送数据包，此函数内将缓存的 L2 报头写入数据包，即可将数据包发往数据链路层。

如果邻居实例中没有缓存 L2 层报头，ip_finish_output2()函数将调用邻居实例中的 output()函数继续处理数据包，此函数内将先解析邻居的物理地址，然后再发送数据包，详见下文。

■已缓存 L2 报头

假设邻居实例中缓存了 L2 层报头，数据包将交由 neigh_hh_output(hh, skb)函数继续处理，函数定义如下（/include/net/neighbour.h）：

```
static inline int neigh_hh_output(const struct hh_cache *hh, struct sk_buff *skb)
{
    unsigned int seq;
    int hh_len;

    do {      /*复制邻居中缓存的 L2 层报头至数据包*/
        seq = read_seqbegin(&hh->hh_lock);
        hh_len = hh->hh_len;
        if (likely(hh_len <= HH_DATA_MOD)) {
            memcpy(skb->data - HH_DATA_MOD, hh->hh_data, HH_DATA_MOD);
        } else {
            int hh_alen = HH_DATA_ALIGN(hh_len);
            memcpy(skb->data - hh_alen, hh->hh_data, hh_alen);
        }
    } while (read_seqretry(&hh->hh_lock, seq));

    skb_push(skb, hh_len);
    return dev_queue_xmit(skb);      /*将数据包发送到数据链路层*/
}
```

neigh_hh_output()函数比较简单，它复制邻居实例中缓存的 L2 层报头至数据包，然后调用接口函数 dev_queue_xmit()将数据包发送往数据链路层，下一小将介绍 dev_queue_xmit()函数的实现。

■未缓存 L2 报头

假设邻居实例中没有缓存 L2 层报头，在 dst_neigh_output()函数中将调用邻居实例中的 output()函数处理数据包，此函数内要先解析出邻居的物理地址，然后再发送数据包。

在前面介绍的 IPv4 邻居构造函数 arp_constructor()中，如果网络设备关联的数据链路层报头操作结构 header_ops 实例中定义了 cache()函数（以太网定义了此函数），邻居操作结构实例赋值为 **arp_hh_ops** 实例，arp_hh_ops 实例中的 output()函数指针将赋予邻居 output()函数指针成员。

下面先看数据链路层报头操作结构 header_ops 的定义（/include/linux/netdevice.h）：

```
struct header_ops {
    int    (*create) (struct sk_buff *skb, struct net_device *dev,
                      unsigned short type, const void *daddr,const void *saddr, unsigned int len);
```

```
                                        /*由指定源和目的物理地址生成 L2 层报头，写入数据包 skb*/
    ...
    int    (*cache)(const struct neighbour *neigh, struct hh_cache *hh, __be16 type);
                                /*由邻居实例中网络设备和邻居物理地址 ha，生成 L2 报头写入 hh 缓存*/
    void (*cache_update)(struct hh_cache *hh,const struct net_device *dev,const unsigned char *haddr);
                                        /*用新的邻居物理地址 haddr 更新 L2 报头缓存 hh*/
};
```

邻居 output()函数通过某个协议解析出邻居物理地址后，会将物理地址写入 neighbour.ha[]成员。如果 header_ops 实例中定义了 cache()函数，output()函数将调用 cache()函数由输出网络设备物理地址和邻居物理地址生成 L2 层报头，并缓存到邻居 hh 成员。

由于以太网定义的 header_ops 实例中定义 cache()函数，在 arp_constructor()函数中邻居关联的邻居操作结构实例为 **arp_hh_ops**，定义如下（/net/ipv4/arp.c）：

```
static const struct neigh_ops arp_hh_ops = {
    .family =        AF_INET,
    .solicit =       arp_solicit,          /*发送 ARP 请求，解析邻居物理地址*/
    .error_report =      arp_error_report,
    .output =        neigh_resolve_output,      /*解析邻居地址，并将 L2 报头缓存到邻居*/
    .connected_output =      neigh_resolve_output,
};
```

arp_hh_ops 实例中 output()函数 neigh_resolve_output()，将赋予邻居 output()函数指针成员。在邻居实例中尚未缓存 L2 层报头时，将由 neigh_resolve_output()函数处理数据包。

neigh_resolve_output()函数定义如下（/net/core/neighbour.c）：

```
int neigh_resolve_output(struct neighbour *neigh, struct sk_buff *skb)
{
    int rc = 0;

    if (!neigh_event_send(neigh, skb)) {     /*是否需要解析邻居物理地址，/include/net/neighbour.h*/
        int err;            /*如果地址已解析则立即发送数据包，否则在邻居定时器处理函数中发送*/
        struct net_device *dev = neigh->dev;
        unsigned int seq;

        if (dev->header_ops->cache && !neigh->hh.hh_len)
            neigh_hh_init(neigh);
              /*构造 L2 报头（header_ops->cache()），写入邻居 hh 成员，/net/core/neighbour.c*/
        do {
            __skb_pull(skb, skb_network_offset(skb));
            seq = read_seqbegin(&neigh->ha_lock);
            err = dev_hard_header(skb, dev, ntohs(skb->protocol),neigh->ha, NULL, skb->len);
                                /*由邻居生成 L2 报头填充至数据包缓存区，/include/linux/netdevice.h*/
        } while (read_seqretry(&neigh->ha_lock, seq));

        if (err >= 0)
            rc = dev_queue_xmit(skb);       /*将数据包发往数据链路层，/include/linux/netdevice.h*/
```

```
            else
                goto out_kfree_skb;
        }
out:
        return rc;
        ...
}
```

neigh_event_send()函数判断是否需要解析邻居物理地址，如果不需要（已解析）则返回 0。如果需要解析物理地址，则 neigh_event_send()函数将数据包添加到邻居未解析数据包队列 neigh->arp_queue 末尾，激活邻居定时器 neigh->timer，函数返回 1。定时器回调函数中将解析邻居物理地址，并生成 L2 报头写入邻居 hh 成员，然后发送 neigh->arp_queue 队列中的数据包。

如果 neigh_event_send()函数返回 0，neigh_resolve_output()函数由邻居中缓存的邻居物理地址，调用函数 neigh_hh_init()生成 L2 层报头，并写入邻居 hh 成员（如果尚未生成），然后写入（生成）L2 层报头至数据包，最后调用接口函数 dev_queue_xmit(skb)发送数据包至数据链路层。

如时 neigh_event_send()函数返回 1，neigh_resolve_output()函数返回 0。

6 解析邻居地址

前面介绍了，在邻居物理地址尚未解析时，在 neigh_event_send()函数中将激活 neigh->timer 定时器，定时器回调函数中将解析邻居物理地址。

IPv4 中通过地址解析协议（ARP）来实现邻居物理地址的解析，在 IPv6 中采用的是邻居发现协议（NDISC 或 ND）。

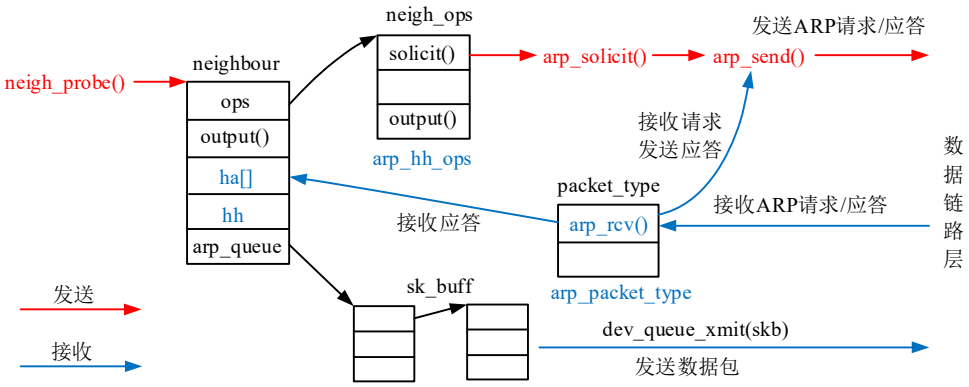
■概述

在分配邻居的 neigh_alloc()函数中，邻居 neigh->timer 定时器的回调函数设为 **neigh_timer_handler()**，在此函数中将解析邻居物理地址，函数调用关系简列如下：

neigh_probe()函数内将调用邻居操作结构中的 solicit()函数，解析邻居物理地址。对于邻居操作结构实例 arp_hh_ops 此函数为 arp_solicit()，在 arp_solicit()函数内将发送 ARP 请求。

ARP 是一个网络层协议，它定义并注册了 packet_type 结构体实例 arp_packet_type，在其 arp_rcv()处理函数中，如果接收到的是 ARP 请求，将会发送 ARP 应答，应答本机物理地址。如果接收到的是 ARP 应答，将会更新邻居实例中的物理地址，生成并写入 L2 报头，发送邻居 neigh->arp_queue 队列中的数据包。

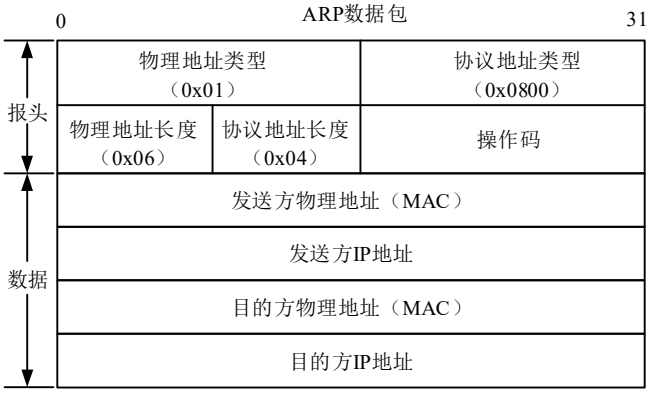
下图示意了发送 ARP 请求、接收 ARP 应答的流程：



arp_solicit()函数内将调用 arp_send()函数发送 ARP 请求（或应答）数据包。arp_rcv()函数用于接收 ARP 请求或应答，如果接收到的是 ARP 请求，将调用 arp_send()函数发送 ARP 应答。如果接收到的是 ARP 应答，将更新邻居实例中的 ha[]（邻居物理地址）和 hh 成员（L2 层报头缓存），并发送 arp_queue 队列中的数据包。

■ARP 协议

IPv4 中通过 ARP 协议解析邻居物理地址。ARP 可视为一个网络层协议，ARP 数据包由 ARP 报头和数据组成，如下图所示：



数据区依次是发送方硬件地址（MAC 地址）、发送方 IP 地址、目的硬件地址（MAC）、目的 IP 地址。ARP 数据包包括请求数据包和应答数据包，数据包由谁发出，谁就是发送方，另一方为目的方。

ARP 报头包括硬件地址类型（2 字节）、协议地址类型（2 字节）、硬件地址长度（1 字节）、协议地址长度（1 字节）、操作码（2 字节）字段。

ARP 报头由 arphdr 结构体表示，定义如下（/include/uapi/linux/if_arp.h）：

```
struct arphdr {
    __be16          ar_hrd;         /*物理地址类型，网络字节序，以太网（10Mbps）为 0x01*/
    __be16          ar_pro;         /*协议地址类型（网络层协议），网络字节序，IPv4 为 0x8000*/
    unsigned char   ar_hln;         /*物理地址长度，字节数（0x06）*/
    unsigned char   ar_pln;         /*协议地址长度，字节数（0x04），IPv4*/
    __be16          ar_op;          /*操作码，网络字节序*/
}
```

arphdr 结构体成员简介如下：

●**ar_hrd**：网络设备（接口）物理地址类型，16bit，类型列表定义在/include/uapi/linux/if_arp.h 头文件，例如：

```
#define   ARPHRD_NETROM    0        /* from KA9Q: NET/ROM pseudo*/
#define   ARPHRD_ETHER     1        /* Ethernet 10Mbps，以太网物理地址*/
```

```
#define   ARPHRD_EETHER     2        /* Experimental Ethernet*/
#define   ARPHRD_AX25       3        /* AX.25 Level 2*/
#define   ARPHRD_PRONET     4        /* PROnet token ring*/
#define   ARPHRD_CHAOS      5        /* Chaosnet*/
#define   ARPHRD_IEEE802    6        /* IEEE 802.2 Ethernet/TR/TB*/
...
```

●**ar_pro**：网络层协议地址类型，类型值定义在头文件/include/uapi/linux/if_ether.h 头文件，例如：

```
#define   ETH_P_LOOP     0x0060      /* Ethernet Loopback packet  */
#define   ETH_P_PUP      0x0200      /* Xerox PUP packet*/
#define   ETH_P_PUPAT    0x0201      /* Xerox PUP Addr Trans packet   */
#define   ETH_P_IP       0x0800      /*IPv4*/
#define   ETH_P_X25      0x0805      /* CCITT X.25*/
#define   ETH_P_ARP      0x0806      /*ARP 协议*/
...
#define   ETH_P_IPV6     0x86DD      /* IPv6 over bluebook*/
...
```

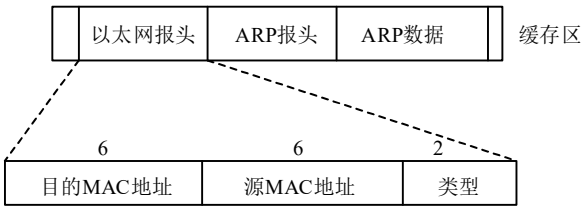
●**ar_hln**：物理地址长度，字节数，以太网地址为 6 字节。

●**ar_pln**：网络层协议地址长度，字节数，IPv4 为 4 字节。

●**ar_op**：操作码，取值定义如下（/include/uapi/linux/if_arp.h）：

```
#define   ARPOP_REQUEST     1        /* ARP 请求*/
#define   ARPOP_REPLY       2        /* ARP 应答*/
#define   ARPOP_RREQUEST    3        /* RARP 请求*/
#define   ARPOP_RREPLY      4        /* RARP 应答*/
#define   ARPOP_InREQUEST   8        /* InARP request*/
#define   ARPOP_InREPLY     9        /* InARP reply*/
#define   ARPOP_NAK         10       /* (ATM)ARP NAK*/
```

ARP 数据包在发往数据链路层时，需要加上数据链路层报头。例如，若发送到以太网则需要添加以太网报头，如下图所示：



在发送 ARP 请求时，目的 MAC 地址未知，写入广播地址即：FF:FF:FF:FF:FF:FF。类型字段为网络层协议类型标识，这里为 **ETH_P_ARP**，表示 ARP 协议。

■发送 ARP 请求

邻居定时器到期回调函数 **neigh_timer_handler()**中将调用 neigh_probe()函数解析邻居物理地址，函数数定义如下（/net/core/neighbour.c）：

```
static void neigh_probe(struct neighbour *neigh)__releases(neigh->lock)
{
    struct sk_buff *skb = skb_peek_tail(&neigh->arp_queue);      /*取队列末属 sk_buff 实例*/
    /* keep skb alive even if arp_queue overflows */
```

```
        if (skb)
            skb = skb_copy(skb, GFP_ATOMIC);     /*复制 sk_buff 实例*/
        write_unlock(&neigh->lock);
        neigh->ops->solicit(neigh, skb);     /*调用邻居操作结构中的 solicit()函数，发送 ARP 请求*/
        atomic_inc(&neigh->probes);
        kfree_skb(skb);     /*释放复制的 sk_buff 实例，原实例还在邻居队列中*/
    }
```

neigh_probe()函数取出邻居 sk_buff 队列最末尾的成员，复制后调用 neigh->ops->solicit(neigh, skb)函数，最后释放复制的 sk_buff 实例，原 sk_buff 实例还留在邻居未解析数据包队列中。

在前面介绍的 IPv4 邻居操作结构 arp_hh_ops 实例中 solicit()函数为 **arp_solicit()**，定义如下：

```
static void arp_solicit(struct neighbour *neigh, struct sk_buff *skb)     /*/net/ipv4/arp.c*/
{
    __be32 saddr = 0;
    u8 dst_ha[MAX_ADDR_LEN], *dst_hw = NULL;
    struct net_device *dev = neigh->dev;
    __be32 target = *(__be32 *)neigh->primary_key;    /*邻居 IP 地址*/
    int probes = atomic_read(&neigh->probes);
    struct in_device *in_dev;

    rcu_read_lock();
    in_dev = __in_dev_get_rcu(dev);    /*输出网络设备 in_device 实例，若不存在，函数返回*/
    ...
    switch (IN_DEV_ARP_ANNOUNCE(in_dev)) {     /*哪些本机 IP 地址可作为 ARP 数据包源地址*/
    default:
    case 0:          /*可使用任意本机 IP 地址，这是默认值*/
        if (skb && inet_addr_type(dev_net(dev),ip_hdr(skb)->saddr) == RTN_LOCAL)
            saddr = ip_hdr(skb)->saddr;       /*从网络层报头中获取源 IP 地址*/
        break;
    case 1:          /* Restrict announcements of saddr in same subnet */
        ...
        break;
    case 2:          /* Avoid secondary IPs, get a primary/preferred one */
        break;
    }
    rcu_read_unlock();

    if (!saddr)          /*如果发送数据包网络层报头中未指定源地址*/
        saddr = inet_select_addr(dev, target, RT_SCOPE_LINK);     /*设置源 IP 地址*/

    probes -= NEIGH_VAR(neigh->parms, UCAST_PROBES);     /*检查邻居参数的探测值*/
    if (probes < 0) {
        if (!(neigh->nud_state & NUD_VALID))
            pr_debug("trying to ucast probe in NUD_INVALID\n");
        neigh_ha_snapshot(dst_ha, neigh, dev);
        dst_hw = dst_ha;
    } else {
```

```
            probes -= NEIGH_VAR(neigh->parms, APP_PROBES);
            if (probes < 0) {
                neigh_app_ns(neigh);
                return;
            }
        }
        arp_send(ARPOP_REQUEST, ETH_P_ARP, target, dev, saddr,dst_hw, dev->dev_addr, NULL);
                                                        /*发送 ARP 请求，/net/ipv4/arp.c*/
    }
```

arp_solicit()函数最后调用 **arp_send()**函数发送 ARP 请求，**arp_send()**函数还是发送 ARP 应答的接口函数，函数定义如下：

```
void arp_send(int type, int ptype, __be32 dest_ip,struct net_device *dev, __be32 src_ip,
              const unsigned char *dest_hw, const unsigned char *src_hw,
              const unsigned char *target_hw)
/*
*type：操作码，ptype：ARP 协议标识，为 ETH_P_ARP，dest_ip：目的 IP 地址，dev：网络设备，
*src_ip：源 IP 地址，dest_hw：写入 L2 报头的目的物理地址，此处为 NULL，src_hw：源物理地址，
*target_hw：写入 ARP 数据区的目的方物理地址。
*/
{
    struct sk_buff *skb;
    if (dev->flags&IFF_NOARP)     /*如果网络设备不支持 ARP，函数返回*/
        return;

    skb = arp_create(type, ptype, dest_ip, dev, src_ip,dest_hw, src_hw, target_hw); /*创建 ARP 数据包*/
    if (!skb)
        return;

    arp_xmit(skb);     /*发送 ARP 数据包*/
}
```

arp_send()函数主要分两步，一是创建 ARP 数据包，二是发送 ARP 数据包，下面分别介绍这两步的实现。

●创建 ARP 数据包

arp_create()函数用于创建 ARP 数据包，函数定义如下：

```
struct sk_buff *arp_create(int type, int ptype, __be32 dest_ip,
                    struct net_device *dev, __be32 src_ip,const unsigned char *dest_hw,
                    const unsigned char *src_hw,const unsigned char *target_hw)
{
    struct sk_buff *skb;
    struct arphdr *arp;       /*ARP 报头*/
    unsigned char *arp_ptr;
    int hlen = LL_RESERVED_SPACE(dev);     /*数据链路层报头长度*/
    int tlen = dev->needed_tailroom;
```

```
skb = alloc_skb(arp_hdr_len(dev) + hlen + tlen, GFP_ATOMIC);     /*分配 sk_buff 实例*/
...
skb_reserve(skb, hlen);
skb_reset_network_header(skb);
arp = (struct arphdr *) skb_put(skb, arp_hdr_len(dev));
skb->dev = dev;
skb->protocol = htons(ETH_P_ARP);     /*网络层协议标识*/
if (!src_hw)
      src_hw = dev->dev_addr;
if (!dest_hw)       /*目的物理地址为 0，则设为广播地址*/
      dest_hw = dev->broadcast;

if (dev_hard_header(skb, dev, ptype, dest_hw, src_hw, skb->len) < 0)    /*填充 ARP 报头*/
      goto out;

switch (dev->type) {
default:
      arp->ar_hrd = htons(dev->type);    /*设置物理地址类型*/
      arp->ar_pro = htons(ETH_P_IP);    /*设置 ARP 报头中网络层协议地址类型*/
      break;
...
}

arp->ar_hln = dev->addr_len;
arp->ar_pln = 4;
arp->ar_op = htons(type);     /*操作码*/

arp_ptr = (unsigned char *)(arp + 1);    /*指向 ARP 数据区*/

memcpy(arp_ptr, src_hw, dev->addr_len);    /*ARP 数据区写入发送方物理地址*/
arp_ptr += dev->addr_len;
memcpy(arp_ptr, &src_ip, 4);    /*写入发送方 IP 地址*/
arp_ptr += 4;

switch (dev->type) {
...
default:
      if (target_hw)
            memcpy(arp_ptr, target_hw, dev->addr_len);
                                          /*复制目的方物理地址至 ARP 数据包数据区*/
      else
            memset(arp_ptr, 0, dev->addr_len);     /*ARP 数据区目的方物理地址，写入 0*/
      arp_ptr += dev->addr_len;
}
memcpy(arp_ptr, &dest_ip, 4);     /*ARP 数据区写入目的方 IP 地址*/
return skb;
```

...

}

●发送 ARP 数据包

发送 ARP 数据包的函数为 arp_xmit(skb)，定义如下（/net/ipv4/arp.c）：

```
void arp_xmit(struct sk_buff *skb)
{
    NF_HOOK(NFPROTO_ARP, NF_ARP_OUT, NULL, skb,NULL, skb->dev, dev_queue_xmit_sk);
}
```

arp_xmit()先调用 NF_ARP_OUT 钩子注册的回调函数，然后调用 **dev_queue_xmit_sk()**函数将数据包发往数据链路层。

■接收 ARP 请求或应答

本机发送 ARP 请求后，需要接收邻居发回的应答消息。目的机器 ARP 协议在接收到 ARP 请求数据包后，需要发送 ARP 应答数据包。本机 ARP 还需要接收 ARP 应答，依此更新邻居实例。

在 ARP 初始化函数 arp_init()中注册了 ARP 协议在网络层的 packet_type 结构体 **arp_packet_type** 实例，实例定义如下：

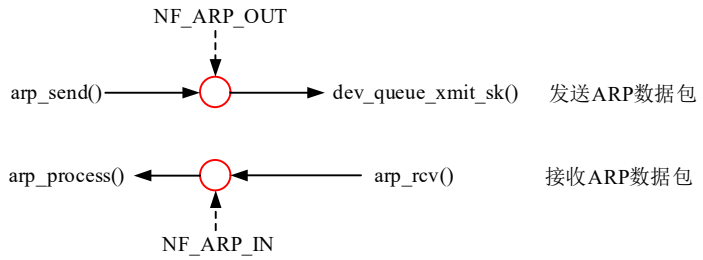
```
static struct packet_type arp_packet_type __read_mostly = {
    .type =    cpu_to_be16(ETH_P_ARP),    /*ARP 协议类型在以太网报头中的标识*/
    .func =    arp_rcv,    /*接收 ARP 数据包函数*/
};
```

arp_rcv()函数中需要负责接收 ARP 请求和应答。

ARP 协议为网络层协议，与 IPv4 协议一样，在数据包发送接收流程中也插入了 netfilter 钩子，钩子编号定义如下（/include/uapi/linux/netfilter_arp.h）：

```
#define   NF_ARP_IN          0
#define   NF_ARP_OUT         1
#define   NF_ARP_FORWARD     2
#define   NF_ARP_NUMHOOKS    3
```

下图示意了 NF_ARP_IN 和 NF_ARP_OUT 钩子的位置：



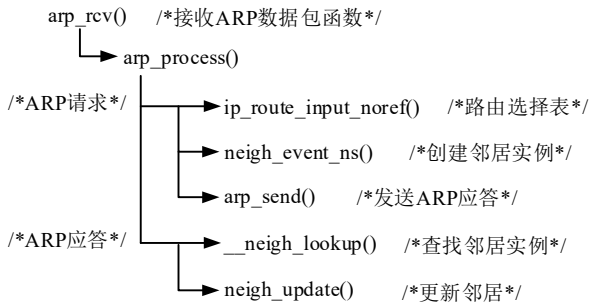
●接收函数

ARP 协议接收数据包函数 **arp_rcv()**，定义如下：

```
static int arp_rcv(struct sk_buff *skb, struct net_device *dev,
                                        struct packet_type *pt, struct net_device *orig_dev)
{
    const struct arphdr *arp;
```

```
    /*检查网络设备是否设置了 IFF_NOARP 标志，或数据包不是发给当前主机的，
    *或数据包是发给环回设备的，就必须将数据包丢弃。*/
    if (dev->flags & IFF_NOARP ||skb->pkt_type == PACKET_OTHERHOST ||
                skb->pkt_type == PACKET_LOOPBACK)
        goto consumeskb;

    skb = skb_share_check(skb, GFP_ATOMIC);   /*如果数据包是共享的就复制一个副本*/
    if (!skb)
        goto out_of_mem;

     /* ARP header, plus 2 device addresses, plus 2 IP addresses.    */
    if (!pskb_may_pull(skb, arp_hdr_len(dev)))
        goto freeskb;

    arp = arp_hdr(skb);       /*指向 ARP 报头*/
    if (arp->ar_hln != dev->addr_len || arp->ar_pln != 4)     /*检查地址长度是否正确*/
        goto freeskb;

    memset(NEIGH_CB(skb), 0, sizeof(struct neighbour_cb));

    return NF_HOOK(NFPROTO_ARP, NF_ARP_IN, NULL, skb,dev, NULL, arp_process);
                /*调用 NF_ARP_IN 钩子注册的回调函数，返回 1，然后调用 arp_process()函数*/
    ...
}
```

arp_process()函数用于处理 ARP 请求和应答数据包。对于 ARP 请求，需要以本机 MAC 地址构建数据包发送应答数据包给请求方，并在本机创建表示请求方的邻居实例。如果收到的是应答数据包，则利用数据包中信息更新本地邻居实例。arp_process()函数调用关系简列如下图所示：



arp_process()函数定义如下：

```
static int arp_process(struct sock *sk, struct sk_buff *skb)
{
    struct net_device *dev = skb->dev;
    struct in_device *in_dev = __in_dev_get_rcu(dev);
    struct arphdr *arp;
    unsigned char *arp_ptr;
    struct rtable *rt;     /*保存路由选择查找结果*/
    unsigned char *sha;
```

```
__be32 sip, tip;
u16 dev_type = dev->type;   /*设备类型*/
int addr_type;
struct neighbour *n;
struct net *net = dev_net(dev);
bool is_garp = false;
...
arp = arp_hdr(skb);       /*指向 ARP 报头*/

switch (dev_type) {       /*地址类型和协议类型检查*/
default:
      if (arp->ar_pro != htons(ETH_P_IP) ||htons(dev_type) != arp->ar_hrd)
            goto out;
      break;
case ARPHRD_ETHER:
case ARPHRD_FDDI:
case ARPHRD_IEEE802:
      if ((arp->ar_hrd != htons(ARPHRD_ETHER) &&arp->ar_hrd != htons(ARPHRD_IEEE802)) ||
                  arp->ar_pro != htons(ETH_P_IP))
            goto out;
      break;
...
}
if (arp->ar_op != htons(ARPOP_REPLY) &&arp->ar_op != htons(ARPOP_REQUEST))
      goto out;                  /*若不是 ARP 应答也不是请求，跳转至 out 处*/

arp_ptr = (unsigned char *)(arp + 1);     /*指向 ARP 数据区*/
sha  = arp_ptr;            /*sha 指向发送方物理地址*/
arp_ptr += dev->addr_len;         /*arp_ptr 指向发送方 IP 地址*/
memcpy(&sip, arp_ptr, 4);     /*复制发送方 IP 地址至 sip 变量*/
arp_ptr += 4;       /*arp_ptr 指向目的方物理地址*/
switch (dev_type) {
...
default:
      arp_ptr += dev->addr_len;     /*arp_ptr 指向目的方 IP 地址*/
}
memcpy(&tip, arp_ptr, 4);     /*目的方 IP 地址（本机）保存至 tip 变量*/
if (ipv4_is_multicast(tip) ||(!IN_DEV_ROUTE_LOCALNET(in_dev) && ipv4_is_loopback(tip)))
      goto out;            /*目的方 IP 地址检查，不能是组播和发给环回设备的数据包*/
...
if (sip == 0) {     /*特殊情形：发送方 IP 地址为 0*/
      if (arp->ar_op == htons(ARPOP_REQUEST) &&inet_addr_type(net, tip) == RTN_LOCAL &&
                                                !arp_ignore(in_dev, sip, tip))
            arp_send(ARPOP_REPLY, ETH_P_ARP, sip, dev, tip, sha,dev->dev_addr, sha);
                                                /*向发送方发送 ARP 应答消息*/
      goto out;
```

```
}
/*如果是 ARP 请求，执行路由选择查找，发送 ARP 应答*/
if (arp->ar_op == htons(ARPOP_REQUEST) &&ip_route_input_noref(skb, tip, sip, 0, dev) == 0) {
    rt = skb_rtable(skb);   /*rt 指向 rtable 实例，表示路由选择结果*/
    addr_type = rt->rt_type;

    if (addr_type == RTN_LOCAL) {     /*数据包是发给本机的*/
        int dont_send;
        dont_send = arp_ignore(in_dev, sip, tip);     /*是否不执行 ARP 应答，1 不应答，0 应答*/
        if (!dont_send && IN_DEV_ARPFILTER(in_dev))
            dont_send = arp_filter(sip, tip, dev);
                                        /*执行路由选择查找，输入、输出网络接口应相同*/
        if (!dont_send) {
            n = neigh_event_ns(&arp_tbl, sha, &sip, dev);   /*创建表示发送方的邻居实例*/
            if (n) {
                arp_send(ARPOP_REPLY, ETH_P_ARP, sip,dev, tip, sha, dev->dev_addr,sha);
                                                        /*发送 ARP 应答*/
                neigh_release(n);       /*递减邻居计数*/
            }
        }
        goto out;
    } else if (IN_DEV_FORWARD(in_dev)) {
        ...                             /*处理 ARP 代理*/
    }
}

/*如果收到的是 ARP 应答，执行以下操作*/
n = __neigh_lookup(&arp_tbl, &sip, dev, 0);     /*查找表示下一跳的邻居实例*/

if (IN_DEV_ARP_ACCEPT(in_dev)) {    /*检查是否被设置了接受 ARP 请求*/
    is_garp = arp->ar_op == htons(ARPOP_REQUEST) && tip == sip &&
                                    inet_addr_type(net, sip) == RTN_UNICAST;

    if (!n &&((arp->ar_op == htons(ARPOP_REPLY)    &&
            inet_addr_type(net, sip) == RTN_UNICAST) || is_garp))
        n = __neigh_lookup(&arp_tbl, &sip, dev, 1);
}

if (n) {        /*查找到了邻居实例*/
    int state = NUD_REACHABLE;     /*邻居可达状态*/
    int override;
    override = time_after(jiffies,n->updated +NEIGH_VAR(n->parms, LOCKTIME)) ||is_garp;
    if (arp->ar_op != htons(ARPOP_REPLY) ||skb->pkt_type != PACKET_HOST)
        state = NUD_STALE;
    neigh_update(n, sha, state,override ? NEIGH_UPDATE_F_OVERRIDE : 0);
                                /*更新邻居实例，发送邻居实例队列中未解析数据包*/
```

```
            neigh_release(n);
        }
    out:
        consume_skb(skb);
        return 0;
    }
```

arp_process()函数内需要进行一些检查，如果收到的是 ARP 请求数据包，则创建表示发送方的邻居实例，发送 ARP 应答。

如果收到的是 ARP 应答，则查找表示发送方（下一跳）的邻居实例，调用 **neigh_update()**函数更新邻居实例，主要是更新其中的邻居物理地址、L2 报头缓存，以及发送邻居缓存的未解析数据包。

●更新邻居实例

neigh_update()函数代码简列如下（/net/core/neighbour.c）：

```
int neigh_update(struct neighbour *neigh, const u8 *lladdr, u8 new,u32 flags)
/*neigh：邻居实例，lladdr：邻居物理地址，new：这里为邻居可达状态，flags：过期状态*/
{
    u8 old;
    int err;
    int notify = 0;
    struct net_device *dev;
    int update_isrouter = 0;

    write_lock_bh(&neigh->lock);

    dev      = neigh->dev;     /*网络设备*/
    old      = neigh->nud_state;     /*邻居旧状态*/
    err      = -EPERM;
    ...
    if (!(new & NUD_VALID)) {     /*新状态为 NUD_REACHABLE，相与结果不为 0*/
        ...
    }

    if (!dev->addr_len) {     /*如果本机网络设备物理地址为 0*/
        lladdr = neigh->ha;
    } else if (lladdr) {     /*lladdr 指向邻居物理地址*/
        if ((old & NUD_VALID) &&!memcmp(lladdr, neigh->ha, dev->addr_len))
            lladdr = neigh->ha;             /*如果邻居物理地址没有改变*/
    } else {       /*如是 lladdr 为 NULL*/
        err = -EINVAL;
        if (!(old & NUD_VALID))
            goto out;
        lladdr = neigh->ha;
    }

    if (new & NUD_CONNECTED)
        neigh->confirmed = jiffies;
```

```
neigh->updated = jiffies;        /*记录当前时间*/

err = 0;
update_isrouter = flags & NEIGH_UPDATE_F_OVERRIDE_ISROUTER;   /*本机是否是路由器*/
if (old & NUD_VALID) {        /*邻居原为有效状态*/
        ...
}

if (new != old) {       /*新旧状态不一样*/
        neigh_del_timer(neigh);
        if (new & NUD_PROBE)
                atomic_set(&neigh->probes, 0);
        if (new & NUD_IN_TIMER)
                neigh_add_timer(neigh, (jiffies +((new & NUD_REACHABLE) ?
                                                  neigh->parms->reachable_time :0)));
        neigh->nud_state = new;         /*设置新状态*/
        notify = 1;
}

if (lladdr != neigh->ha) {     /*邻居物理地址改变了*/
        write_seqlock(&neigh->ha_lock);
        memcpy(&neigh->ha, lladdr, dev->addr_len);     /*复制物理地址至邻居 ha 成员*/
        write_sequnlock(&neigh->ha_lock);
        neigh_update_hhs(neigh);     /*更新邻居 hh 成员等，/net/core/neighbour.c*/
        if (!(new & NUD_CONNECTED))
                neigh->confirmed = jiffies -
                                   (NEIGH_VAR(neigh->parms, BASE_REACHABLE_TIME) << 1);
        notify = 1;
}
if (new == old)
        goto out;
if (new & NUD_CONNECTED)     /*状态为可达时，NUD_CONNECTED 状态有效*/
        neigh_connect(neigh);     /*设置 neigh->output = neigh->ops->connected_output*/
else
        neigh_suspect(neigh);
if (!(old & NUD_VALID)) {        /*邻居原状态不为有效，发送 arp_queue 队列中数据包*/
        struct sk_buff *skb;
        /*新状态为可达，NUD_VALID 有效，发送 arp_queue 队列中数据包*/
        while (neigh->nud_state & NUD_VALID &&
                        (skb = __skb_dequeue(&neigh->arp_queue)) != NULL) {
                struct dst_entry *dst = skb_dst(skb);
                struct neighbour *n2, *n1 = neigh;
                write_unlock_bh(&neigh->lock);

                rcu_read_lock();
                n2 = NULL;
```

```
                if (dst) {
                    n2 = dst_neigh_lookup_skb(dst, skb);
                            /*调用 dst_entry 实例操作结构中的查找邻居实例*/
                           /*dst->ops->neigh_lookup(dst, skb, NULL)，/include/net/dst.h*/
                    if (n2)
                        n1 = n2;     /*n1 赋予 dst->ops->neigh_lookup()函数查找的实例*/
                }
                n1->output(n1, skb);    /*发送数据包，此时邻居地址已解析，直接发送数据包*/
                if (n2)
                    neigh_release(n2);
                rcu_read_unlock();
                write_lock_bh(&neigh->lock);
            }
            __skb_queue_purge(&neigh->arp_queue);
            neigh->arp_queue_len_bytes = 0;
        }
    out:
        if (update_isrouter) {
            neigh->flags = (flags & NEIGH_UPDATE_F_ISROUTER) ?
                (neigh->flags | NTF_ROUTER) :(neigh->flags & ~NTF_ROUTER);
        }
        write_unlock_bh(&neigh->lock);

        if (notify)
            neigh_update_notify(neigh);
             /*执行 netevent_notif_chain 通知链，向用户 netlink 套接字发送消息，/net/core/neighbour.c*/
        return err;
    }
```

neigh_update()函数将 ARP 应答中的邻居物理地址写入 neigh->ha 成员，调用 neigh_update_hhs()函数更新 neigh->hh 成员（L2 层报头），发送邻居 arp_queue 队列中数据包。

在发送数据包时，调用 dst_entry->ops->neigh_lookup()函数查找邻居实例，并调用其中的 output()函数发送数据包。在前面介绍的 IPv4 构建邻居实例函数中，邻居 output()函数为 neigh_resolve_output()，在此函数中，如果邻居物理地址已解析，则直接对发送数据包写入 L2 层报头，将数据包发往数据链路层即可。

7 用户接口

在发送数据包时，IPv4 在发送数据包流程中会自动创建邻居实例，并通过 ARP 协议解析邻居物理地址。用户进程也可以通过 netlink 套接字，ioctl()系统调用等直接对邻居表和邻居进行操作，下文中将以通过 RTM_NEWNEIGH 消息添加邻居为例，介绍邻居子系统与用户接口的实现。

■RTM_NEWNEIGH 消息

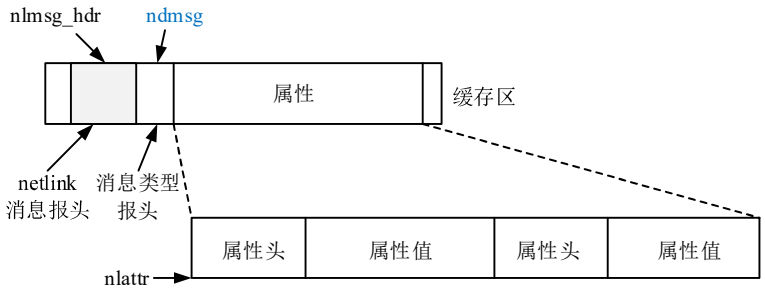
用户进程可通过 NETLINK_ROUTE 套接字操作邻居表和邻居。

邻居子系统公共部分初始化函数 neigh_init()为操作邻居表、邻居消息注册了处理函数。例如，添加新邻居的 RTM_NEWNEIGH 消息处理函数为 neigh_add()：

```
static int __init neigh_init(void)     /*/net/core/neighbour.c*/
```

```
{
    rtnl_register(PF_UNSPEC, RTM_NEWNEIGH, neigh_add, NULL, NULL);
    ...
}
```

NETLINK_ROUTE 套接字添加邻居的 RTM_NEWNEIGH 消息格式如下图所示：



RTM_NEWNEIGH 消息类型报头由 ndmsg 结构体表示，定义如下（/include/uapi/linux/neighbour.h）：

```
struct ndmsg {
    __u8        ndm_family;     /*协议簇，AF_INET（IPv4）等*/
    __u8        ndm_pad1;
    __u16       ndm_pad2;
    __s32       ndm_ifindex;    /*输出网络设备（接口）编号*/
    __u16       ndm_state;      /*邻居状态*/
    __u8        ndm_flags;      /*标志，NTF_USE、NTF_PROXY 等*/
    __u8        ndm_type;       /*类型*/
};
```

属性类型定义如下：

```
enum {
    NDA_UNSPEC,
    NDA_DST,        /*邻居 IP 地址*/
    NDA_LLADDR,     /*邻居物理地址*/
    NDA_CACHEINFO,
    NDA_PROBES,
    NDA_VLAN,
    NDA_PORT,
    NDA_VNI,
    NDA_IFINDEX,    /*本机输出网络设备编号*/
    NDA_MASTER,
    NDA_LINK_NETNSID,
    __NDA_MAX       /*最大属性数量*/
};
```

■添加邻居

RTM_NEWNEIGH 消息的处理函数为 **neigh_add()**，函数定义如下（/net/core/neighbour.c）：

```
static int neigh_add(struct sk_buff *skb, struct nlmsghdr *nlh)
{
```

```
int flags = NEIGH_UPDATE_F_ADMIN | NEIGH_UPDATE_F_OVERRIDE;
struct net *net = sock_net(skb->sk);
struct ndmsg *ndm;        /*消息类型报头*/
struct nlattr *tb[NDA_MAX+1];     /*属性头指针数组*/
struct neigh_table *tbl;
struct net_device *dev = NULL;
struct neighbour *neigh;
void *dst, *lladdr;
int err;

ASSERT_RTNL();
err = nlmsg_parse(nlh, sizeof(*ndm), tb, NDA_MAX, NULL);     /*解析消息中属性*/
...
if (tb[NDA_DST] == NULL)     /*NDA_DST 属性不能为空，邻居 IP 地址*/
        goto out;

ndm = nlmsg_data(nlh);     /*消息中传递的报头 ndmsg 实例*/
if (ndm->ndm_ifindex) {     /*由输出网络设备编号查找 net_device 实例*/
        dev = __dev_get_by_index(net, ndm->ndm_ifindex);
        ...
        if (tb[NDA_LLADDR] && nla_len(tb[NDA_LLADDR]) < dev->addr_len) /*本地地址有效性*/
                goto out;
}

tbl = neigh_find_table(ndm->ndm_family);        /*由协议簇查找邻居表*/
...
dst = nla_data(tb[NDA_DST]);        /*邻居 IP 地址*/
lladdr = tb[NDA_LLADDR] ? nla_data(tb[NDA_LLADDR]) : NULL;     /*指向邻居物理地址*/

if (ndm->ndm_flags & NTF_PROXY) {       /*ARP 代理*/
        ...
}
...
neigh = neigh_lookup(tbl, dst, dev);     /*查找邻居*/
if (neigh == NULL) {     /*如果邻居尚不存在*/
        if (!(nlh->nlmsg_flags & NLM_F_CREATE)) {    /*netlink 消息标志中没有指定创建新实例*/
                err = -ENOENT;
                goto out;
        }
        neigh = __neigh_lookup_errno(tbl, dst, dev);       /*邻居不存在则创建*/
        ...
} else {        /*如果邻居已存存*/
        if (nlh->nlmsg_flags & NLM_F_EXCL) { /*套接字指定了 NLM_F_EXCL 标志*/
                err = -EEXIST;
                neigh_release(neigh);   /*引用计数减 1，为 0 则释放邻居*/
                goto out;
```

```
            }
            /*邻居已存在，但套接字没有指定 NLM_F_EXCL 标志*/
            if (!(nlh->nlmsg_flags & NLM_F_REPLACE))
                flags &= ~NEIGH_UPDATE_F_OVERRIDE;
        }

        if (ndm->ndm_flags & NTF_USE) {
            neigh_event_send(neigh, NULL);
                                    /*发送 ARP 请求，解析物理地址等，/include/net/neighbour.h*/
            err = 0;
        } else      /*没有指定 NTF_USE 标志，直接更新邻居实例*/
            err = neigh_update(neigh, lladdr, ndm->ndm_state, flags);   /*更新邻居，/net/core/neighbour.c*/
        neigh_release(neigh);

    out:
        return err;
    }
```

neigh_add()函数主要工作是根据消息属性中传递的邻居 IP 地址、输出网络设备编号查找邻居实例是否已存在，如果不存在则创建，然后根据消息中传递的邻居物理地址更新邻居实例。

其它邻居表、邻居相关消息的处理函数请读者自行阅读源代码。

13.1.4 发送数据包

理解了路由选择子系统和邻居子系统，就翻过了网络层协议中的两座大山。本小节和下一小节将串联起网络层发送和接收数据包的流程，介绍数据包在网络层传输的路径。

1 概述

在介绍数据包在 IPv4 网络层发送数据包流程前，先回顾一下 IPv4 网络层报头，然后介绍发送数据包在网络层中所经过的路径。

■ IPv4 报头

IPv4 网络层报头结构在前面 12.4.4 小节已经介绍过了，报头结构如下图所示：

IP 报头在内核中由 iphdr 结构体表示，结构体定义如下（/include/uapi/linux/ip.h）：

```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)     /*处理器为小端序*/
    __u8    ihl:4,              /*报头长度*/
            version:4;          /*版本*/
```

```
#elif defined (__BIG_ENDIAN_BITFIELD)     /*处理器为大端序*/
        __u8        version:4,
                    ihl:4;
#else
#error      "Please fix <asm/byteorder.h>"
#endif
        __u8        tos;          /*服务类型*/
        __be16      tot_len;      /*数据包总长度*/
        __be16      id;           /*标识，用于数据包分段与重组*/
        __be16      frag_off;     /*偏移量，用于数据包分段与重组*/
        __u8        ttl;          /*跳数，数据包在传输路径中经过的最大节点数*/
        __u8        protocol;     /*传输层协议类型*/
        __sum16 check;            /*检验和*/
        __be32      saddr;        /*源 IP 地址，网络字节序*/
        __be32      daddr;        /*目的 IP 地址，网络字节序*/
        /*IP 选项从这里开始*/
};
```

iphdr 结构体与 IPv4 报头是对应的关系，结构体即 IPv4 报头，结构体各成员简介如下：

●**version**：版本号，4bit，此处必须为 4，IPv6 为 6。

●**ihl**：4bit，报头以 4 字节为单位的长度，如果没有使用 IP 选项，则报头长度为 20 字节，此字段值为 5。

●**tos**：8bit，服务类型（TOS）表示 IP 数据包的类型，如实时数据包、非实时数据包（优先级）等。

●**tot_len**：16bit，数据包总长度（首部加上数据），以字节计。由于其长度为 16bit，因此数据包的理论最大长度为 65535 字节。然而，数据包很少有超过 1500 字节的，因为数据链路层数据包的长度值最大通常不会超过 1500 字节。

●**id**：标识，对于分段来说，id 字段很重要。分段是指网络层数据包的长度大于数据链路层帧的最大长度，因此在发往数据链路层前需要将数据包进行分段，在接收端的网络层需要对分段进行重组。同一个网络层数据包分段后的数据包 id 值相同。

●**frag_off**：偏移量，低 3bit 和高 13bit 语义如下：

□**标志**：低 3bit，各标志位语义如下：

xx1：分段数据包中，置位表示本分段后面还有其它的分段（More Fragments，MF），除最后一个分段外，其它的分段都必须设置这个标志。最后一个分段此标志位设为 xx0。

x1x：表示数据包不允许分段（Don’t Fragments，DF）。

1xx：表示拥塞（Congestion，CE），用于显示传递拥塞信息。

□**分段偏移量**：高 13bit，表示本分段中起始数据在原始数据包中的偏移量，偏移量以 8 字节为单位，第一个分段偏移量为 0。

●**ttl**：8bit，寿命（存活时间），用来确保数据包不会被无休止地传输。在主机发送数据包时会对该字段设置一个初始值，数据包每经过一个路由器，该字段值会减 1，当值为 0 时，此数据包将会被丢弃，并向发送端发回一条 ICMPv4“超时”信息（在网络层协议中实现）。

●**protocol**：8bit，数据包使用的传输层协议，如 IPPROTO_TCP、IPPROTO_UDP 等。

●**check**：16bit，仅根据报头计算的校验和，每次报头数据改变后需要重新计算校验和。

●**saddr**：32bit，发送主机的 IP 地址。

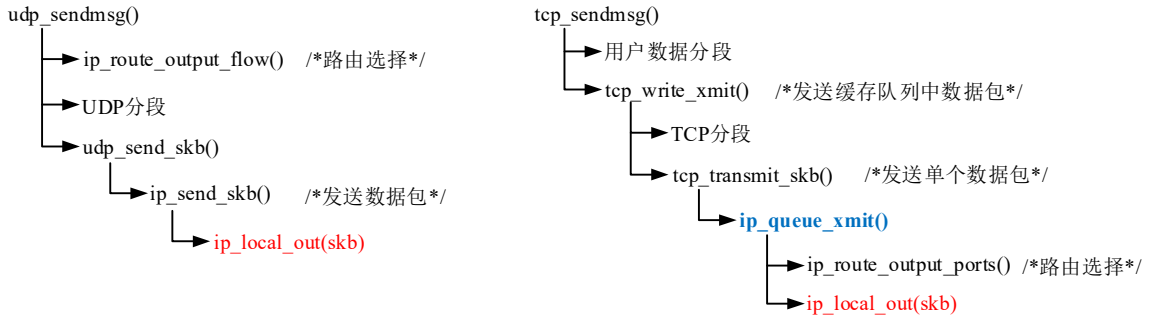
●**daddr**：32bit，目的主机 IP 地址。发送主机通常通过 DNS 查找来决定目的主机，DNS 查找是通过 DNS 服务器将域名转换成 IP 地址。

源 IP 地址和目的 IP 地址是指通信两端端系统中的 IP 地址，而不是中间路由器端口的 IP 地址，这两个 IP 地址在数据包转发过程中是不会改变的（除非路由器使用 NAT）。

●IP **选项**：0~40 字节，选项字段允许 IP 报头被扩展，选项字段很少使用，因此本书暂不介绍。在 IPv6 报头中不支持选项字段，而是通过扩展报头来实现选项的功有。

■发送流程

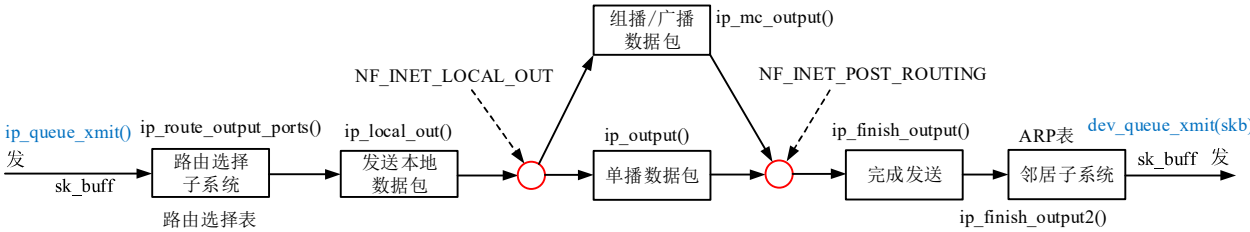
UDP、TCP 发送数据包函数调用关系简列如下图所示：



UDP 发送数据包函数 udp_sendmsg()先执行路由选择查找（如果需要），然后对用户数据进行分段，构建一个 sk_buff 实例（用户数据保存在分散数据块中）或一个 sk_buff 实例链表，最后调用 udp_send_skb()函数发送表示一个数据报的数据包（链表）。udp_send_skb()函数对数据包写入 UDP 报头，然后调用 **ip_local_out()**函数将数据包发送数据包。

TCP 发送数据包函数 tcp_sendmsg()对用户数据进行分段，生成的数据包添加到套接字发送缓存队列，然后从发送缓存队列中取出数据包逐个发往网络层。在发送数据包时，可能还需要对数据包进行分段，分段后数据包最后由 IPv4 定义的 ip_queue_xmit()函数接收。ip_queue_xmit()函数内先执行路由选择查找，然后调用 **ip_local_out()**函数将数据包发送数据包。

udp_sendmsg()函数前面介绍过了，下面以 ip_queue_xmit()函数为例，介绍其执行流程，如下图所示：



ip_queue_xmit()函数内先执行路由选择查找，然后调用 ip_local_out()函数继续处理数据包。ip_local_out()函数调用路由选择结果 dst_entry 实例中的 output()函数处理数据包。对于单播的数据包由 ip_output()函数处理，组播或组播的数据包由 ip_mc_output()函数处理，这两个函数都调用 ip_finish_output()函数。

ip_finish_output()函数调用 ip_finish_output2()函数，在邻居子系统中查找邻居实例（如果未找到则创建），如果邻居物理地址已解析则对数据包写入 L2 层报头，将数据包发往数据链路层。如果邻居物理地址未解析，则先发送 ARP 请求，解析邻居物理地址后再发送数据包。

2 发送 TCP 数据包

ip_queue_xmit()函数是 TCP 向网络层发送数据包调用的接口函数，函数代码简列如下：

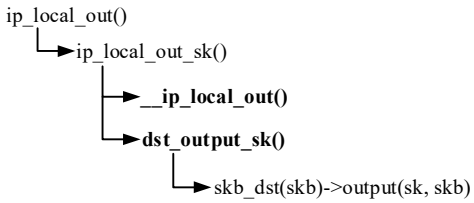
```
int ip_queue_xmit(struct sock *sk, struct sk_buff *skb, struct flowi *fl)      /*/net/ipv4/ip_output.c*/
/*sk：套接字，skb：数据包，fl：保存路由选择参数*/
{
    struct inet_sock *inet = inet_sk(sk);
    struct ip_options_rcu *inet_opt;    /*IP 选项，/include/net/inet_sock.h*/
    struct flowi4 *fl4;      /*指向路由选择参数*/
```

```
    struct rtable *rt;        /*指向路由选择结果*/
    struct iphdr *iph;        /*指向网络层报头*/
    int res;

    rcu_read_lock();
    inet_opt = rcu_dereference(inet->inet_opt);
    fl4 = &fl->u.ip4;
    rt = skb_rtable(skb);        /*skb->_skb_refdst 指向 dst_entry 实例（rtable 实例）*/
    if (rt)        /*rt 不为 NULL，说明已经执行了路由选择*/
        goto packet_routed;        /*跳转至 packet_routed*/

    /*rt 为 NULL，检查 sock 是否关联了 rtable 实例*/
    rt = (struct rtable *)__sk_dst_check(sk, 0); /*sock 实例关联的 dst_entry 实例，已执行了连接操作*/
    if (!rt) {        /*sock 也没有关联 rtable 实例，需要执行路由选择*/
        __be32 daddr;

        daddr = inet->inet_daddr;      /*目的 IP 地址*/
        if (inet_opt && inet_opt->opt.srr)      /*IP 报头具有选项*/
            daddr = inet_opt->opt.faddr;

        /*执行路由选择，见 12.9.3 小节*/
        rt = ip_route_output_ports(sock_net(sk), fl4, sk,daddr, inet->inet_saddr,inet->inet_dport,
                        inet->inet_sport,sk->sk_protocol,RT_CONN_FLAGS(sk),sk->sk_bound_dev_if);
        if (IS_ERR(rt))
            goto no_route;
        sk_setup_caps(sk, &rt->dst);
                            /*根据网络设备功能设置 sk->sk_route_caps 成员等，/net/core/sock.c*/
    }
    skb_dst_set_noref(skb, &rt->dst);      /*/include/linux/skbuff.h*/
                            /*设置 skb->_skb_refdst 成员（dst_entry 实例指针）为路由选择结果*/

packet_routed:        /*数据包关联了 dst_entry 实例*/
    if (inet_opt && inet_opt->opt.is_strictroute && rt->rt_uses_gateway)
        goto no_route;

    /*分配填充 IP 报头*/
    skb_push(skb, sizeof(struct iphdr) + (inet_opt ? inet_opt->opt.optlen : 0));
    skb_reset_network_header(skb);
    iph = ip_hdr(skb);      /*指向网络层报头*/
    *((__be16 *)iph) = htons((4 << 12) | (5 << 8) | (inet->tos & 0xff));      /*设置网络层报头*/
    if (ip_dont_fragment(sk, &rt->dst) && !skb->ignore_df)    /*是否不需要分段*/
        iph->frag_off = htons(IP_DF);    /*设置不允许分段标志，不允许分段*/
    else
        iph->frag_off = 0;
    iph->ttl          = ip_select_ttl(inet, &rt->dst);      /*设置 ttl 值*/
    iph->protocol = sk->sk_protocol;        /*传输层协议*/
```

```
    ip_copy_addrs(iph, fl4);      /*复制源、目的 IP 址至 IP 报头*/

    if (inet_opt && inet_opt->opt.optlen) {     /*处理 IP 选项*/
        iph->ihl += inet_opt->opt.optlen >> 2;
        ip_options_build(skb, &inet_opt->opt, inet->inet_daddr, rt, 0);
    }

    ip_select_ident_segs(sock_net(sk), skb, sk,skb_shinfo(skb)->gso_segs ?: 1); /*设置报头分段 id 值*/
    skb->priority = sk->sk_priority;     /*优先级*/
    skb->mark = sk->sk_mark;

    res = ip_local_out(skb);     /*执行下一步的输出数据包操作，/include/net/ip.h*/
    rcu_read_unlock();
    return res;
    ...
}
```

ip_queue_xmit()函数首先判断是否需要执行路由选择，如果需要则执行路由选择，并将结果 dst_entry 实例设置到 sk_buff 实例，然后对数据包写入 IP 报头，最后调用 ip_local_out(skb)函数执行下一步的操作。

ip_local_out(skb)函数调用关系如下图所示：



ip_local_out(skb)函数调用 ip_local_out_sk()函数，定义如下（/net/ipv4/ip_output.c）：

```
int ip_local_out_sk(struct sock *sk, struct sk_buff *skb)
{
    int err;

    err = __ip_local_out(skb);      /*函数内只执行 NF_INET_LOCAL_OUT 钩子注册的回调函数*/
    if (likely(err == 1))          /*err 为 1 表示继续处理数据包*/
        err = dst_output_sk(sk, skb);     /*err 为 1 调用 dst_output_sk()函数*/

    return err;
}
```

__ip_local_out()函数内只执行 **NF_INET_LOCAL_OUT** 钩子处注册的回调函数，回调函数返回 1，则调用 **dst_output_sk()**函数继续处理数据包。

dst_output_sk()函数定义如下（/include/net/dst.h）：

```
static inline int dst_output_sk(struct sock *sk, struct sk_buff *skb)
{
    return skb_dst(skb)->output(sk, skb);
}
```

dst_output_sk()函数调用 sk_buff 实例关联的 dst_entry 实例中的 output()函数，此函数在路由选择操作中赋值，例如，对于单播数据包此函数为 **ip_output()**，组播或广播的数据包此函数为 **ip_mc_output()**，这

两个函数的实现详见下文。

3 发送单播数据包

ip_output()函数用于处理单播数据包，函数定义在/net/ipv4/ip_output.c 文件内，代码如下：

```
int ip_output(struct sock *sk, struct sk_buff *skb)
{
    struct net_device *dev = skb_dst(skb)->dev;

    IP_UPD_PO_STATS(dev_net(dev), IPSTATS_MIB_OUT, skb->len);

    skb->dev = dev;
    skb->protocol = htons(ETH_P_IP);        /*设置协议类型*/

    return NF_HOOK_COND(NFPROTO_IPV4, NF_INET_POST_ROUTING, sk, skb,
                NULL, dev,ip_finish_output,!(IPCB(skb)->flags & IPSKB_REROUTED));
}
```

ip_output()函数内先调用 **NF_INET_POST_ROUTING** 钩子注册的回调函数，然后将数包交给函数 **ip_finish_output()**函数处理。

ip_finish_output()函数定义如下（/net/ipv4/ip_output.c）：

```
static int ip_finish_output(struct sock *sk, struct sk_buff *skb)
{
    unsigned int mtu;

#if defined(CONFIG_NETFILTER) && defined(CONFIG_XFRM)     /*如果启用了 IPsec*/
    if (skb_dst(skb)->xfrm) {
        IPCB(skb)->flags |= IPSKB_REROUTED;
        return dst_output_sk(sk, skb);
    }
#endif
    mtu = ip_skb_dst_mtu(skb);       /*传输路径中链路层数据包最大长度*/
    if (skb_is_gso(skb))       /*如果存在分散数据块*/
        return ip_finish_output_gso(sk, skb, mtu);
                                        /*如果需要分段，分段后调用 ip_finish_output2()函数*/
    if (skb->len > mtu || (IPCB(skb)->flags & IPSKB_FRAG_PMTU))     /*数据包是否需要分段*/
        return ip_fragment(sk, skb, mtu, ip_finish_output2);
                            /*数据包分段，然后调用 ip_finish_output2()函数，/net/ipv4/ip_output.c*/
    return ip_finish_output2(sk, skb);     /*数据包不用分段直接发送*/
}
```

XFRM 框架用于实现 IPsec 系统，此系统通过 Netfilter 子系统在钩子处注册回调函数以实现。数据包在执完回调函数后再执行正常的发送流程（如果配置了 NETFILTER 和 XFRM）。

ip_finish_output()函数检查数据包是否存在分散数据块，或数据包长度大于链路 MTU 值，如果是则需要对数据包进行分段，然后调用 ip_finish_output2()函数继续发送数据包。数据包的分段和重组后面再介绍，这里假设单个数据包的长度没有超过 MTU 值，直接调用 ip_finish_output2()函数发送数据包。

ip_finish_output2()函数代码简列如下（/net/ipv4/ip_output.c）：

```
static int ip_finish_output2(struct sock *sk, struct sk_buff *skb)
```

```
{
    struct dst_entry *dst = skb_dst(skb);     /*路由选择结果*/
    struct rtable *rt = (struct rtable *)dst;
    struct net_device *dev = dst->dev;       /*输了网络设备*/
    unsigned int hh_len = LL_RESERVED_SPACE(dev);     /*链路层报头长度*/
    struct neighbour *neigh;     /*邻居实例*/
    u32 nexthop;

    if (rt->rt_type == RTN_MULTICAST) {         /*组播数据包*/
        IP_UPD_PO_STATS(dev_net(dev), IPSTATS_MIB_OUTMCAST, skb->len);
    } else if (rt->rt_type == RTN_BROADCAST)     /*广播数据包*/
        IP_UPD_PO_STATS(dev_net(dev), IPSTATS_MIB_OUTBCAST, skb->len);

    if (unlikely(skb_headroom(skb) < hh_len && dev->header_ops)) {   /*如果数据包缓存区空间不足*/
        struct sk_buff *skb2;
        skb2 = skb_realloc_headroom(skb, LL_RESERVED_SPACE(dev));       /*重新分配数据包*/
        if (!skb2) {
            kfree_skb(skb);
            return -ENOMEM;
        }
        if (skb->sk)
            skb_set_owner_w(skb2, skb->sk);
        consume_skb(skb);
        skb = skb2;
    }

    rcu_read_lock_bh();
    nexthop = (__force u32) rt_nexthop(rt, ip_hdr(skb)->daddr);   /*下一跳 IP 地址，/include/net/route.h*/
    neigh = __ipv4_neigh_lookup_noref(dev, nexthop);     /*查找邻居实例*/
    if (unlikely(!neigh))
        neigh = __neigh_create(&arp_tbl, &nexthop, dev, false);   /*没有找到邻居实例，则创建实例*/
    if (!IS_ERR(neigh)) {
        int res = dst_neigh_output(dst, neigh, skb);     /*/include/net/dst.h*/

        rcu_read_unlock_bh();
        return res;
    }
    ...
}
```

ip_finish_output2()函数通过发送网络设备（接口）net_device 实例和下一跳 IP 地址查找邻居实例（未找到则创建），然后调用 dst_neigh_output()函数继续处理数据包，函数定义如下：

```
static inline int dst_neigh_output(struct dst_entry *dst, struct neighbour *n,struct sk_buff *skb)
{
    const struct hh_cache *hh;

    if (dst->pending_confirm) {
```

```
            unsigned long now = jiffies;
            dst->pending_confirm = 0;
            if (n->confirmed != now)
                n->confirmed = now;
        }

        hh = &n->hh;     /*邻居实例中的数据链路层报头缓存*/
        if ((n->nud_state & NUD_CONNECTED) && hh->hh_len)     /*链路层报头缓存已经存在*/
            return neigh_hh_output(hh, skb);     /*对数据包写入 L2 层报头，发送数据包*/
        else
            return n->output(n, skb);     /*调用邻居实例中的 output()函数，见上一小节*/
    }
```

dst_neigh_output()函数在前一小节介绍过了，函数内判断邻居实例中是否缓存了 L2 层报头，如果是则对数据包写入 L2 层报头，发送到数据链路层，如果邻居没有缓存 L2 层报头，则调用邻居实例中的 output()函数，先解析邻居物理地址，然后再生成 L2 层报头，写入数据包，最后发送数据包。

网络层数据包发往数据链路层的接口函数为 dev_queue_xmit()，后面将介绍此函数的实现。

4 发送组播/广播数据包

在前面介绍的发送通道路由选择中，对于组播或广播的数据包，路由选择结果 dst_entry 实例中 output()函数设为 **ip_mc_output()**，函数定义如下（/net/ipv4/ip_output.c）：

```
int ip_mc_output(struct sock *sk, struct sk_buff *skb)
{
    struct rtable *rt = skb_rtable(skb);       /*路由选择结果*/
    struct net_device *dev = rt->dst.dev;
    IP_UPD_PO_STATS(dev_net(dev), IPSTATS_MIB_OUT, skb->len);

    skb->dev = dev;
    skb->protocol = htons(ETH_P_IP);

    if (rt->rt_flags&RTCF_MULTICAST) {
        if (sk_mc_loop(sk)        /*如果是回环的组播数据包*/
    #ifdef CONFIG_IP_MROUTE     /*如果当前机器为组播路由器*/
            ...
    #endif
            ) {
            struct sk_buff *newskb = skb_clone(skb, GFP_ATOMIC);       /*克隆数据包*/
            if (newskb)
                NF_HOOK(NFPROTO_IPV4, NF_INET_POST_ROUTING,
                    sk, newskb, NULL, newskb->dev,dev_loopback_xmit);
                                                /*由 dev_loopback_xmit()函数处理*/
        }
        if (ip_hdr(skb)->ttl == 0) {
            kfree_skb(skb);
            return 0;
        }
```

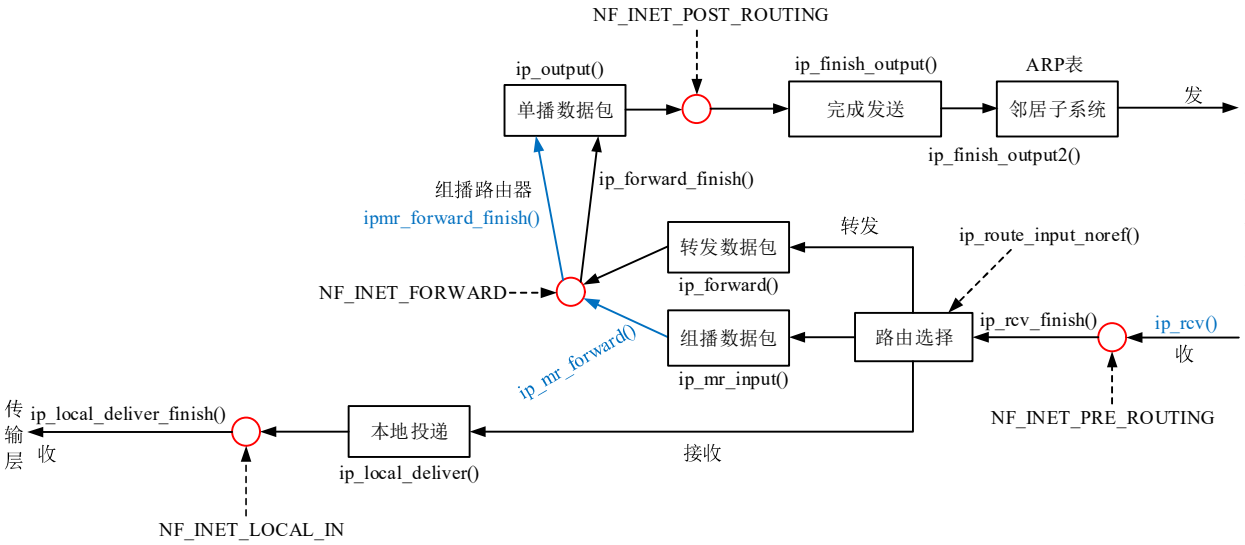
```
    }

    if (rt->rt_flags&RTCF_BROADCAST) {          /*广播数据包*/
        struct sk_buff *newskb = skb_clone(skb, GFP_ATOMIC);        /*复制数据包*/
        if (newskb)
            NF_HOOK(NFPROTO_IPV4, NF_INET_POST_ROUTING, sk, newskb,
                                                NULL, newskb->dev, dev_loopback_xmit);
    }

    return NF_HOOK_COND(NFPROTO_IPV4, NF_INET_POST_ROUTING, sk, skb, NULL,
                skb->dev, ip_finish_output,!(IPCB(skb)->flags & IPSKB_REROUTED));
                                        /*原始数据包由 ip_finish_output()函数处理*/
}
```

ip_mc_output()对于环回的组播数据包和广播数据包，复制一个副本交由 dev_loopback_xmit()函数处理，经网络设备发回给本主机，原始数据包交由 ip_finish_output()函数继续处理。

13.1.5 接收数据包

每个网络层协议需要定义并注册 packet_type 实例，实例中需要定义接收数据包的函数。网络层接收数据包从此函数开始。

IPv4 网络层协议注册 packet_type 实例的为 **ip_packet_type**，实例中接收数据包函数为 **ip_rcv()**，此函数是数据包进入网络层的入口。ip_rcv()函数执行流程如下图所示：



ip_rcv()函数首先执路由选择，由 ip_route_input_noref()函数完成，确定数据包下一步的走向。如果数据包是投递给本机的则调用 ip_local_deliver()函数将数据包传递给本机传输层。如果数据包是需要组播的数据包且本机为组播路由器，则调用 ip_mr_input()函数发送数据包。如果是需要转发的数据包则调用函数 ip_forward()处理。ip_mr_input()和 ip_forward()函数最终都调用前面介绍过的 ip_output()函数将数据包发送出去。IPv4 接收数据包的函数主要在/net/ipv4/ip_input.c 文件内实现。

1 接收函数

IPv4 协议在/net/ipv4/af_inet.c 文件内定义了 packet_type 结构体实例 ip_packet_type，如下：

```
static struct packet_type ip_packet_type __read_mostly = {
    .type = cpu_to_be16(ETH_P_IP),
```

```
    .func = ip_rcv,        /*接收数据包函数*/
};
```

ip_packet_type 实例在初始化函数 inet_init()中注册。

IPv4 网络层接收数据包函数 **ip_rcv()**，定义如下（/net/ipv4/ip_input.c）：

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device *orig_dev)
{
    const struct iphdr *iph;     /*网络层报头指针*/
    u32 len;

    if (skb->pkt_type == PACKET_OTHERHOST)     /*数据包类型*/
        goto drop;


    IP_UPD_PO_STATS_BH(dev_net(dev), IPSTATS_MIB_IN, skb->len);

    skb = skb_share_check(skb, GFP_ATOMIC);
          /*检查是否是共享数据包，是则克隆数据包，/include/linux/skbuff.h*/
    ...

    if (!pskb_may_pull(skb, sizeof(struct iphdr)))     /**/
        goto inhdr_error;

    iph = ip_hdr(skb);     /*指向网络层报头*/

    if (iph->ihl < 5 || iph->version != 4)     /*报头检查*/
        goto inhdr_error;
    ...

    if (unlikely(ip_fast_csum((u8 *)iph, iph->ihl)))     /*检查检验和*/
        goto csum_error;

    len = ntohs(iph->tot_len);     /*数据包长度*/
    ...

    skb->transport_header = skb->network_header + iph->ihl*4;     /*传输层报头指针*/
    memset(IPCB(skb), 0, sizeof(struct inet_skb_parm));

    skb_orphan(skb);   /*取消数据包的所有者，使其成为孤儿，/include/linux/skbuff.h*/

    return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, NULL, skb,dev, NULL,
                                                        ip_rcv_finish);
    ...
}
```

ip_rcv()函数在执行一些检查后，调用 NF_INET_PRE_ROUTING 钩子注册的回调函数，然后调用函数

ip_rcv_finish()继续处理数据包。

ip_rcv_finish()函数定义在/net/ipv4/ip_input.c 文件内，函数定义如下：

```
static int ip_rcv_finish(struct sock *sk, struct sk_buff *skb)
{
    const struct iphdr *iph = ip_hdr(skb);     /*指向网络层报头*/
    struct rtable *rt;

    if (sysctl_ip_early_demux && !skb_dst(skb) && !skb->sk) {
        const struct net_protocol *ipprot;
        int protocol = iph->protocol;

        ipprot = rcu_dereference(inet_protos[protocol]);     /*传输层协议注册的 net_protocol 实例*/
        if (ipprot && ipprot->early_demux) {
            ipprot->early_demux(skb);     /*调用 net_protocol 实例的 early_demux()函数*/
            iph = ip_hdr(skb);
        }
    }

    if (!skb_dst(skb)) {        /*需要并执行路由选择*/
        int err = ip_route_input_noref(skb, iph->daddr, iph->saddr,iph->tos, skb->dev);
        ...
    }

#ifdef CONFIG_IP_ROUTE_CLASSID
    if (unlikely(skb_dst(skb)->tclassid)) {
        struct ip_rt_acct *st = this_cpu_ptr(ip_rt_acct);
        u32 idx = skb_dst(skb)->tclassid;
        st[idx&0xFF].o_packets++;
        st[idx&0xFF].o_bytes += skb->len;
        st[(idx>>16)&0xFF].i_packets++;
        st[(idx>>16)&0xFF].i_bytes += skb->len;
    }
#endif

    if (iph->ihl > 5 && ip_rcv_options(skb))    /处理 IP 选项*/
        goto drop;

    rt = skb_rtable(skb);     /*路由选择结果*/
    if (rt->rt_type == RTN_MULTICAST) {        /*组播数据包*/
        IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INMCAST,skb->len);
    } else if (rt->rt_type == RTN_BROADCAST)       /*广播数据包*/
        IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INBCAST,skb->len);

    return dst_input(skb);     /*调用路由选择结果中 dst_entry.input()函数*/
    ...
}
```

ip_rcv_finish()函数内执的最重要的工作就是调用 **ip_route_input_noref()**函数执行路由选择查找，函数在 12.9.2 小节已经介绍过了。查找结果中对本地接收的数据包和转发的数据包等分别赋予不同的处理函数，例如，对于本机接收的数据包处理函数为：

```
skb_dst(skb)->dst.input= ip_local_deliver;   /*投递到本机的数据包*/
```

对于转发数据包处理函数为：

```
skb_dst(skb)->dst.input = ip_forward;     /*转发函数*/
skb_dst(skb)->dst.output = ip_output;      /*转发函数调用的输出数据包函数*/
```

ip_rcv_finish()函数最后调用 dst_input()函数接收数据包，函数定义如下（/include/net/dst.h）：

```
static inline int dst_input(struct sk_buff *skb)
{
    return skb_dst(skb)->input(skb);
}
```

dst_input()函数最终调用查找结果 dst_entry 实例中的 input()函数接收数据包。

2 接收本地数据包

由上面的分析可知，对于投递到本地的数据包，接收函数为 **ip_local_deliver()**，函数定义如下：

```
int ip_local_deliver(struct sk_buff *skb)     /*/net/ipv4/ip_input.c*/
{
    if (ip_is_fragment(ip_hdr(skb))) {      /*数据包是分段数据包*/
        if (ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER))
                                        /*重组数据包，成功返回 0，下一小节再介绍*/
            return 0;
    }

    return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, NULL, skb,
                                        skb->dev, NULL,ip_local_deliver_finish);      /*接收数据包*/
}
```

ip_local_deliver()函数首先判断数据包是否是分段数据包，如果是则执行数据包重组，如果不是分段数据包则通过 NF_HOOK()宏先执行 NF_INET_LOCAL_IN 钩子回调函数，然后调用 **ip_local_deliver_finish()** 函数接收数据包。数据包的分段与重组下一小节再专门介绍，这里先看一下本地接收函数的实现。

```
static int ip_local_deliver_finish(struct sock *sk, struct sk_buff *skb)
{
    struct net *net = dev_net(skb->dev);
    __skb_pull(skb, skb_network_header_len(skb));

    rcu_read_lock();
    {
        int protocol = ip_hdr(skb)->protocol;
        const struct net_protocol *ipprot;
        int raw;

    resubmit:
        raw = raw_local_deliver(skb, protocol);   /*接收原始数据包，/net/ipv4/raw.c*/
```

```
            ipprot = rcu_dereference(inet_protos[protocol]);     /*查找传输层协议 net_protocol 实例*/
            if (ipprot) {   /*注册了传输层协议 net_protocol 实例*/
                int ret;
                if (!ipprot->no_policy) {
                    if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {     /*xfrm*/
                        ...
                    }
                    nf_reset(skb);
                }
                ret = ipprot->handler(skb);     /*调用传输层协议注册 net_protocol 实例中的处理函数*/
                ...
                IP_INC_STATS_BH(net, IPSTATS_MIB_INDELIVERS);
            } else {      /*没有注册 net_protocol 实例，发送 ICMP 消息*/
                if (!raw) {
                    if (xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
                        IP_INC_STATS_BH(net, IPSTATS_MIB_INUNKNOWNPROTOS);
                        icmp_send(skb, ICMP_DEST_UNREACH,ICMP_PROT_UNREACH, 0);
                    }
                    kfree_skb(skb);
                } else {
                    IP_INC_STATS_BH(net, IPSTATS_MIB_INDELIVERS);
                    consume_skb(skb);
                }     /*if (!raw)结束*/
            }   /* if (ipprot)  结束*/
        }
     out:
        rcu_read_unlock();
        return 0;
    }
```

ip_local_deliver_finish()函数的主要工作是查找传输层协议注册的 net_protocol 实例并调用其中的处理函数，接收数据包，数据包从此进入传输层。

3 转发数据包

对于需要转发的数据包，其 skb_dst(skb)->dst.input 函数为 ip_forward()，定义如下

```
int ip_forward(struct sk_buff *skb)   /*/net/ipv4/ip_forward.c*/
{
    u32 mtu;
    struct iphdr *iph;    /* Our header */
    struct rtable *rt;    /* Route we use */
    struct ip_options *opt    = &(IPCB(skb)->opt);     /*IP 选项*/

    if (skb->pkt_type != PACKET_HOST)     /*如果不是投递给本机的数据包*/
        goto drop;

    if (unlikely(skb->sk))
```

```
        goto drop;

    if (skb_warn_if_lro(skb))
        goto drop;

    if (!xfrm4_policy_check(NULL, XFRM_POLICY_FWD, skb))
        goto drop;

    /*如果设置了 router_alert IP 选项，调用 ip_call_ra_chain()将数据包交给所有原始套接字*/
    if (IPCB(skb)->opt.router_alert && ip_call_ra_chain(skb))    /*/net/ipv4/ip_input.c*/
        return NET_RX_SUCCESS;

    skb_forward_csum(skb);

    if (ip_hdr(skb)->ttl <= 1)      /*ttl 值检测*/
        goto too_many_hops;

    if (!xfrm4_route_forward(skb))
        goto drop;

    rt = skb_rtable(skb);       /*路由选择查找结果*/

    if (opt->is_strictroute && rt->rt_uses_gateway)
        goto sr_failed;

    IPCB(skb)->flags |= IPSKB_FORWARDED;
    mtu = ip_dst_mtu_maybe_forward(&rt->dst, true);     /*链路 MTU 值*/
    if (ip_exceeds_mtu(skb, mtu)) {       /*转发数据包长度，不能超过 MTU*/
        IP_INC_STATS(dev_net(rt->dst.dev), IPSTATS_MIB_FRAGFAILS);
        icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED,htonl(mtu));
        goto drop;
    }

    if (skb_cow(skb, LL_RESERVED_SPACE(rt->dst.dev)+rt->dst.header_len))     /*复制报头*/
        goto drop;
    iph = ip_hdr(skb);      /*指向 IP 报头*/

    if (IPCB(skb)->flags & IPSKB_DOREDIRECT && !opt->srr &&!skb_sec_path(skb))
        ip_rt_send_redirect(skb);       /*需要重定向*/

    skb->priority = rt_tos2priority(iph->tos);      /*数据包优先级，由服务类型转优先级*/
    return NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD, NULL, skb,
                                        skb->dev, rt->dst.dev, ip_forward_finish);
    ...
}
```

ip_forward()函数最后执行 NF_INET_FORWARD 钩子注册的回调函数，然后调用 ip_forward_finish()函数继续处理数据包，函数定义如下（/net/ipv4/ip_forward.c）：

```
static int ip_forward_finish(struct sock *sk, struct sk_buff *skb)
{
    struct ip_options *opt     = &(IPCB(skb)->opt);

    IP_INC_STATS_BH(dev_net(skb_dst(skb)->dev), IPSTATS_MIB_OUTFORWDATAGRAMS);
    IP_ADD_STATS_BH(dev_net(skb_dst(skb)->dev), IPSTATS_MIB_OUTOCTETS, skb->len);

    if (unlikely(opt->optlen))
        ip_forward_options(skb);

    skb_sender_cpu_clear(skb);
    return dst_output_sk(sk, skb);    /*调用 ip_output()函数*/
}
```

ip_forward_finish()函数最后调用 dst_output_sk()发送数据包，而此函数又调用 skb_dst(skb)->dst.output 函数输出数据包，此处输出函数为 **ip_output()**。ip_output()函数在前面介绍过了，与发送单播数据包时相同。

13.1.6 数据包分段与重组

前面介绍的 ip_output()函数是发送 UDP、TCP 等数据包的通用函数。UDP 数据包可能一个数据包只由一个 sk_buff 实例表示，用户数据保存在分散数据块中，也可能由一个数据包链表表示，后面的数据包链接到第一个数据包的 skb_shinfo(skb)->frag_list 链表。TCP 数据包也可能会使用分散数据块。

IPv4 网络层在发送数据包时，如果数据包长度超过了 MTU 值（网络设备不支持 GSO），或者数据包长度超过了 GSO 最大数据长度时，需要对数据包进行分段，而在接收数据包时需要对数据包进行重组。

1 分段

在 ip_finish_output()函数中需要考虑数据包的分段，相关代码简列如下：

```
static int ip_finish_output(struct sock *sk, struct sk_buff *skb)
{
    unsigned int mtu;
    ...
    mtu = ip_skb_dst_mtu(skb);       /*传输路径中链路层数据包最大长度*/
    if (skb_is_gso(skb))       /*如果数据包存在分散数据块*/
        return ip_finish_output_gso(sk, skb, mtu);       /*/net/ipv4/ip_output.c*/
                /*如果需要分段，调用 ip_fragment()分段后调用 ip_finish_output2()函数*/
    if (skb->len > mtu || (IPCB(skb)->flags & IPSKB_FRAG_PMTU))     /*数据包是否需要分段*/
        return ip_fragment(sk, skb, mtu, ip_finish_output2);
                        /*数据包分段，然后调用 ip_finish_output2()函数，/net/ipv4/ip_output.c*/
    ...
}
```

如果数据包存在分散数据块，则调用 ip_finish_output_gso()函数判断是否需要对数据包进行分段，如果要分段则调用 ip_fragment()函数进行分段。如果数据包不存在分散数据块，但数据包长度超过了 MTU 值，则调用 ip_fragment()函数进行分段。

ip_fragment()函数是 IPv4 中对数据包进行分段的接口函数，定义如下（/net/ipv4/ip_output.c）：

```
static int ip_fragment(struct sock *sk, struct sk_buff *skb,unsigned int mtu,
                                                    int (*output)(struct sock *, struct sk_buff *))
/*sock：套接字，skb：需要分段的数据包，mtu：MTU 值，output：继续处理分段后数据包函数*/
{
    struct iphdr *iph = ip_hdr(skb);

    if ((iph->frag_off & htons(IP_DF)) == 0)
        return ip_do_fragment(sk, skb, output);        /*执行分段，/net/ipv4/ip_output.c*/

    if (unlikely(!skb->ignore_df ||
                (IPCB(skb)->frag_max_size &&IPCB(skb)->frag_max_size > mtu))) {
        ...      /*发送 ICMP 消息*/
    }
    return ip_do_fragment(sk, skb, output);
}

ip_do_fragment()函数执行数据包的分段操作，函数定义如下：
int ip_do_fragment(struct sock *sk, struct sk_buff *skb,int (*output)(struct sock *, struct sk_buff *))
{
    struct iphdr *iph;
    int ptr;
    struct net_device *dev;
    struct sk_buff *skb2;
    unsigned int mtu, hlen, left, len, ll_rs;
    int offset;
    __be16 not_last_frag;
    struct rtable *rt = skb_rtable(skb);        /*路由选择结果*/
    int err = 0;

    dev = rt->dst.dev;
    iph = ip_hdr(skb);      /*指向网络层报头*/
    mtu = ip_skb_dst_mtu(skb);      /*MTU 值*/
    if (IPCB(skb)->frag_max_size && IPCB(skb)->frag_max_size < mtu)
        mtu = IPCB(skb)->frag_max_size;

    hlen = iph->ihl * 4;        /*IP 报头长度*/
    mtu = mtu - hlen;          /*分段后数据包中用户数据长度（第一个数据包中含传输层报头）*/
    IPCB(skb)->flags |= IPSKB_FRAG_COMPLETE;

    if (skb_has_frag_list(skb)) {          /*数据包的 skb_shinfo(skb)->frag_list 链表不为空*/
        struct sk_buff *frag, *frag2;
        int first_len = skb_pagelen(skb);     /*第一个数据包中分散数据块的长度*/

        if (first_len - hlen > mtu ||((first_len - hlen) & 7) ||ip_is_fragment(iph) ||skb_cloned(skb))
            goto slow_path;
```

```
skb_walk_frags(skb, frag) {     /*遍历 skb_shinfo(skb)->frag_list 链表中 sk_buff 实例*/
    if (frag->len > mtu ||((frag->len & 7) && frag->next) ||skb_headroom(frag) < hlen)
        goto slow_path_clean;       /*需要对数据包分段*/

    if (skb_shared(frag))       /*共享数据包，需要对数据包分段*/
        goto slow_path_clean;

    BUG_ON(frag->sk);
    if (skb->sk) {      /*关联套接字*/
        frag->sk = skb->sk;
        frag->destructor = sock_wfree;
    }
    skb->truesize -= frag->truesize;
}

/*skb_shinfo(skb)->frag_list 链表中 sk_buff 实例不需要分段，直接逐个发送*/
err = 0;
offset = 0;
frag = skb_shinfo(skb)->frag_list;      /*frag 指向链表头*/
skb_frag_list_init(skb);        /*skb_shinfo(skb)->frag_list 设为 NULL*/
skb->data_len = first_len - skb_headlen(skb);
skb->len = first_len;
iph->tot_len = htons(first_len);
iph->frag_off = htons(IP_MF);       /*设置 IP_MF 标志位，表示后面还有分段*/
ip_send_check(iph);       /*检查校验和*/

for (;;) {      /*逐个发送 skb_shinfo(skb)->frag_list 链表中数据包*/
    if (frag) {
        frag->ip_summed = CHECKSUM_NONE;
        skb_reset_transport_header(frag);
        __skb_push(frag, hlen);
        skb_reset_network_header(frag);
        memcpy(skb_network_header(frag), iph, hlen);
        iph = ip_hdr(frag);
        iph->tot_len = htons(frag->len);
        ip_copy_metadata(frag, skb);
        if (offset == 0)
            ip_options_fragment(frag);
        offset += skb->len - hlen;
        iph->frag_off = htons(offset>>3);
        if (frag->next)
            iph->frag_off |= htons(IP_MF);
        ip_send_check(iph);
    }

    err = output(sk, skb);      /*调用 output()函数发送数据包*/
```

```
            if (!err)
                IP_INC_STATS(dev_net(dev), IPSTATS_MIB_FRAGCREATES);
            if (err || !frag)
                break;

            skb = frag;
            frag = skb->next;
            skb->next = NULL;
        }   /*发送 skb_shinfo(skb)->frag_list 链表中数据包结束，for()循环结束*/

        if (err == 0) {
            IP_INC_STATS(dev_net(dev), IPSTATS_MIB_FRAGOKS);
            return 0;
        }

        while (frag) {      /*没发送完的数据包，释放它*/
            skb = frag->next;
            kfree_skb(frag);
            frag = skb;
        }
        IP_INC_STATS(dev_net(dev), IPSTATS_MIB_FRAGFAILS);
        return err;

slow_path_clean:      /*下面是需要对 skb_shinfo(skb)->frag_list 链表中数据包进行分段的情形*/
        skb_walk_frags(skb, frag2) {
            if (frag2 == frag)
                break;
            frag2->sk = NULL;
            frag2->destructor = NULL;
            skb->truesize += frag2->truesize;
        }
    }    /*if (skb_has_frag_list(skb))  结果*/

slow_path:      /*慢速路径*/
    if ((skb->ip_summed == CHECKSUM_PARTIAL) && skb_checksum_help(skb))
        goto fail;
    iph = ip_hdr(skb);    /*指向网络报头*/
    left = skb->len - hlen;          /*数据包中除 IP 报头之外的用户数据长度*/
    ptr = hlen;                      /*用户数据开始位置（含传输层报头）*/

    ll_rs = LL_RESERVED_SPACE(rt->dst.dev);
    offset = (ntohs(iph->frag_off) & IP_OFFSET) << 3;
    not_last_frag = iph->frag_off & htons(IP_MF);

    while (left > 0) {       /*对数据包进行分段，发送分段后的数据包，循环开始*/
```

```
len = left;
if (len > mtu)
        len = mtu;
if (len < left)   {
        len &= ~7;
}

/*分配 sk_buff 实例*/
skb2 = alloc_skb(len + hlen + ll_rs, GFP_ATOMIC);
if (!skb2) {
        err = -ENOMEM;
        goto fail;
}
ip_copy_metadata(skb2, skb);        /*复制元数据、报头*/
skb_reserve(skb2, ll_rs);
skb_put(skb2, len + hlen);
skb_reset_network_header(skb2);
skb2->transport_header = skb2->network_header + hlen;

if (skb->sk)
        skb_set_owner_w(skb2, skb->sk);

skb_copy_from_linear_data(skb, skb_network_header(skb2), hlen);    /*复制网络层报头*/

/*复制 skb 中用户数据至 skb2*/
if (skb_copy_bits(skb, ptr, skb_transport_header(skb2), len))
        BUG();
left -= len;
/*填充 skb2 中 IP 报头*/
iph = ip_hdr(skb2);
iph->frag_off = htons((offset >> 3));      /*设置偏移量*/

if (IPCB(skb)->flags & IPSKB_FRAG_PMTU)
        iph->frag_off |= htons(IP_DF);

if (offset == 0)
        ip_options_fragment(skb);

if (left > 0 || not_last_frag)
        iph->frag_off |= htons(IP_MF);
ptr += len;
offset += len;
iph->tot_len = htons(len + hlen);
ip_send_check(iph);          /*检查校验和*/

err = output(sk, skb2);      /*调用 output()函数发送分段后的数据包*/
```

```
            ...     /*错误处理*/

            IP_INC_STATS(dev_net(dev), IPSTATS_MIB_FRAGCREATES);
        }     /*while (left > 0)结束*/
        consume_skb(skb);
        IP_INC_STATS(dev_net(dev), IPSTATS_MIB_FRAGOKS);
        return err;
        ...
    }
```

ip_do_fragment()函数内检查数据包的 skb_shinfo(skb)->frag_list 链表是否为空，如果不为空，且链表中数据包长度都没超过 MTU，则直接调用 output()函数逐个发送链表中的 sk_buff 实例。

如果 skb_shinfo(skb)->frag_list 链表中有数据包的长度超过 MTU，或数据包存在分散数据块，则进入慢速路径，对数据包进行分段，并调用 output()函数发送分段后数据包。

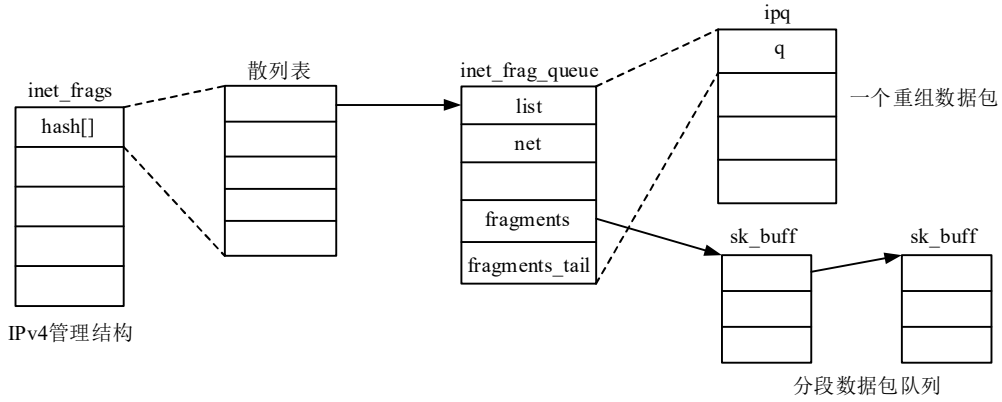
2 重组

分段数据包的重组只在端系统，即目的主机中进行，不会在路由器中进行。内核中用一个队列来管理属于同一个原始数据包的分段数据包，分段数据包都接收到之后，再重组为原始数据包，传递给传输层。

下面先介绍内核对分段数据包的管理结构，然后介绍重组函数的实现。重组函数在 ip_local_deliver()函数中被调用。

■数据结构

下图示意了内核对分段数据包的管理结构：



每个原始数据包（重组数据包）由 ipq 结构体表示，其内嵌的 inet_frag_queue 结构体中包含一个数据包队列，即收到的分段数据包队列。IPv4 中定义了 inet_frags 结构体实例，其中包含一个散列表，用于管理 ipq 实例（一个重组数据包，由内嵌的 inet_frag_queue 结构体成员添加到散列表）。

下面简要介绍以上数据结构的定义，inet_frags 和 inet_frag_queue 结构体定义在/include/net/inet_frag.h 头文件，ipq 结构体定义在/net/ipv4/ip_fragment.c 文件内。

inet_frags 结构体用于管理所有的重组数据包，定义如下：

```
struct inet_frags {
    struct inet_frag_bucket   hash[INETFRAGS_HASHSZ];        /*管理重组数据包的散列表*/

    struct work_struct   frags_work;        /*工作回调函数中释放散列表中 ipq 实例*/
    unsigned int next_bucket;
```

```
    unsigned long last_rebuild_jiffies;
    bool            rebuild;
    u32             rnd;
    seqlock_t       rnd_seqlock;
    int             qsize;      /*ipq 结构体大小，字节数*/

    unsigned int    (*hashfn)(const struct inet_frag_queue *);      /*计算散列值函数*/
    bool            (*match)(const struct inet_frag_queue *q,const void *arg);      /*匹配函数*/
    void            (*constructor)(struct inet_frag_queue *q,const void *arg);      /*构造函数*/
    void            (*destructor)(struct inet_frag_queue *);                        /*析构函数*/
    void            (*skb_free)(struct sk_buff *);
    void            (*frag_expire)(unsigned long data);     /*重组超时定时器回调函数*/
    struct kmem_cache *frags_cachep;        /*指向 ipq 结构体 slab 缓存*/
    const char          *frags_cache_name;
};
```

inet_frags 结构体主要成员简介如下：

●**hash[]**：inet_frag_bucket 结构体数组，用于构成散列表，inet_frag_bucket 结构体定义如下：

```
struct inet_frag_bucket {
    struct hlist_head   chain;      /*散列链表头*/
    spinlock_t          chain_lock;     /*自旋锁*/
};
```

●**hashfn()**：计算散列值函数。

ipq 结构体用于管理一个重组数据包的分段数据包，结构体定义如下（/net/ipv4/ip_fragment.c）：

```
struct ipq {
    struct inet_frag_queue q;       /*inet_frag_queue 结构体成员*/
    u32         user;
    __be32          saddr;      /*源 IP 地址*/
    __be32          daddr;      /*目的 IP 地址*/
    __be16          id;     /*分段 id 值*/
    u8          protocol;       /**/
    u8          ecn;            /* RFC3168 support */
    u16         max_df_size;    /*最大分段长度*/
    int                 iif;        /*输入网络设备编号*/
    unsigned int        rid;
    struct inet_peer *peer;
};
```

ipq 结构体部分成员简介如下：

●**q**：inet_frag_queue 结构体成员，表示分段数据包队列，定义如下（/include/net/inet_frag.h）：

```
struct inet_frag_queue {
    spinlock_t          lock;
    struct timer_list   timer;
    struct hlist_node   list;       /*将实例添加到 inet_frags 实例中散列表*/
    atomic_t        refcnt;         /*引用计数*/
    struct sk_buff      *fragments;         /*指向分段数据包队列头*/
    struct sk_buff      *fragments_tail;    /*指向分段数据包队列尾*/
```

```
    ktime_t             stamp;
    int                 len;      /*当前分段数据包队列中数据结束字节偏移量*/
    int                 meat;
    __u8                flags;    /*标志，如 INET_FRAG_COMPLETE，/include/net/inet_frag.h*/
    u16                 max_size;
    struct netns_frags  *net;     /*指向 netns_frags 结构体，主要包含控制参数，/include/net/inet_frag.h*/
    struct hlist_node   list_evictor;
};
```

●**id**：分段 id 值，值相同的分段数据包属于同一个原始数据包。

●**net**：**指向** netns_frags 结构体，主要包含控制参数，定义如下：

```
struct netns_frags {
    struct percpu_counter     mem ____cacheline_aligned_in_smp;    /*占用内存*/
    /*系统控制参数*/
    int                 timeout;      /*重组超时时间，如果超时分段数据包还没到齐，则重组失败*/
    int                 high_thresh;
    int                 low_thresh;
};
```

在 IPv4 网络命名空间资源结构体 netns_ipv4 结构体中包含 netns_frags 结构体成员，在初始化时将为设置此成员值：

```
struct netns_ipv4 {
    ...
    struct netns_frags    frags;      /*数据包分段参数，/include/net/inet_frag.h*/
    ...
}
```

■初始化

内核在/net/ipv4/ip_fragment.c 文件内定义了 inet_frags 结构体实例，用于管理 IPv4 中的重组数据包，并在 ipfrag_init()函数中对其进行了初始化。

```
static struct inet_frags ip4_frags;
```

在初始化函数 inet_init()中调用 **ipfrag_init()**函数完成了 IPv4 数据包分段/重组的初始化，函数定义如下：

```
void __init ipfrag_init(void)
{
    ip4_frags_ctl_register();      /*在初始网络命名空间注册系统控制参数列表*/
    register_pernet_subsys(&ip4_frags_ops);
    /*以下是初始化 ip4_frags 实例*/
    ip4_frags.hashfn = ip4_hashfn;       /*计算 ipq 实例散列值函数*/
    ip4_frags.constructor = ip4_frag_init;       /*实始化 ipq 实例函数*/
    ip4_frags.destructor = ip4_frag_free;
    ip4_frags.skb_free = NULL;
    ip4_frags.qsize = sizeof(struct ipq);
    ip4_frags.match = ip4_frag_match;       /*查找 ipq 实例时的比对函数，由 ip4_create_arg 传递参数*/
    ip4_frags.frag_expire = ip_expire;
                                    /*重组数据包超时回调函数，释放 ipq 实例，发送 ICMP 消息等*/
```

```
    ip4_frags.frags_cache_name = ip_frag_cache_name;      /*slab 缓存名称*/
    if (inet_frags_init(&ip4_frags))      /*/net/ipv4/inet_fragment.c*/
                                  /*初始化 ip4_frags 实例中散列表，创建 ipq 结构体 slab 缓存等*/
        panic("IP: failed to allocate ip4_frags cache\n");
}
```

ipfrag_init()函数调用 ip4_frags_ctl_register()函数在初始网络命名空间中注册系统控制参数列表。注册 pernet_operations 结构体实例 ip4_frags_ops，实例初始化函数中将设置网络命名空间中 net->ipv4.frags 成员，若是创建新网络命名空间调用初始化函数，还将为网络命名空间注册系统控制参数列表。

ipfrag_init()函数然后初始化 **ip4_frags** 实例，inet_frags_init()函数用于初始化实例中的散列表，并创建 ipq 结构体 slab 缓存，赋予 ip4_frags->frags_cachep 成员。

■重组函数

分段数据包重组在 ip_local_deliver()函数内进行，函数代码简列如下：

```
int ip_local_deliver(struct sk_buff *skb)
{
    if (ip_is_fragment(ip_hdr(skb))) {   /*数据包是分段数据包，报头设置了 IP_MF 或偏移量不为 0*/
        if (ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER))      /*重组数据包，成功返回 0*/
            return 0;
    }
    ...      /*处理重组后数据包*/
}
```

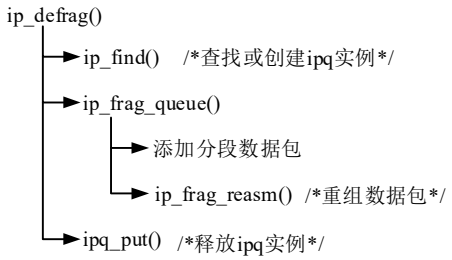
如果收到的数据包 skb 是分段数据包（IP_MF 标志位不为 0，或偏移量不为 0），则调用 **ip_defrag()** 函数对数据包进行重组。

ip_defrag()函数定义如下（/net/ipv4/ip_fragment.c）：

```
int ip_defrag(struct sk_buff *skb, u32 user)
{
    struct ipq *qp;
    struct net *net;

    net = skb->dev ? dev_net(skb->dev) : dev_net(skb_dst(skb)->dev);      /*网络命名空间*/
    IP_INC_STATS_BH(net, IPSTATS_MIB_REASMREQDS);

    qp = ip_find(net, ip_hdr(skb), user);          /*查找或创建 ipq 实例，/net/ipv4/ip_fragment.c*/
    if (qp) {
        int ret;

        spin_lock(&qp->q.lock);

        ret = ip_frag_queue(qp, skb);   /*添加分段数据包，如果分段数据包到齐，则重组*/

        spin_unlock(&qp->q.lock);
        ipq_put(qp);       /*引用计数为 0 时，释放 ipq 实例*/
        return ret;      /*成功返回 0*/
    }
    ...
```

}

ip_defrag()函数调用关系简列如下图所示：



ip_find()函数在散列表中查找 ipq 实例，如果不存在则创建。**ip_frag_queue()**函数将数据包添加到 ipq 实例中的分段数据包队列，当分段数据包到齐时，调用 **ip_frag_reasm()**函数重组数据包。ipq_put()函数释放 ipq 实例。另外，如果重组超时定时器到期了（初始值 30s），重组还没完成，将释放 ipq 实例，向发送方发送 ICMP 消息。

ip_find()函数源代码请读者自行阅读，下面简要介绍 ip_frag_queue()函数的实现。

●添加分段数据包

ip_frag_queue()函数代码简列如下（/net/ipv4/ip_fragment.c）：

```
static int ip_frag_queue(struct ipq *qp, struct sk_buff *skb)
{
    struct sk_buff *prev, *next;
    struct net_device *dev;
    unsigned int fragsize;
    int flags, offset;
    int ihl, end;
    int err = -ENOENT;
    u8 ecn;

    if (qp->q.flags & INET_FRAG_COMPLETE)       /*重组完成*/
        goto err;

    ...
    ecn = ip4_frag_ecn(ip_hdr(skb)->tos);
    offset = ntohs(ip_hdr(skb)->frag_off);       /*偏移量*/
    flags = offset & ~IP_OFFSET;
    offset &= IP_OFFSET;
    offset <<= 3;          /* offset is in 8-byte chunks */
    ihl = ip_hdrlen(skb);         /*ip 报头长度*/

    end = offset + skb->len - skb_network_offset(skb) - ihl;         /*本数据包结束字节偏移量*/
    err = -EINVAL;

    /*是否是最后一个分段数据包*/
    if ((flags & IP_MF) == 0) {
        if (end < qp->q.len ||((qp->q.flags & INET_FRAG_LAST_IN) && end != qp->q.len))
            goto err;
        qp->q.flags |= INET_FRAG_LAST_IN;       /*收到了最后分段数据包*/
```

```
        qp->q.len = end;
    } else {        /*不是最后分段数据包*/
        ...
        if (end > qp->q.len) {
            if (qp->q.flags & INET_FRAG_LAST_IN)
                goto err;
            qp->q.len = end;        /*更新 qp->q.len 值*/
        }
    }
    ...
    /*查找分段数据包在队列中的插入位置*/
    prev = qp->q.fragments_tail;
    if (!prev || FRAG_CB(prev)->offset < offset) {
        next = NULL;
        goto found;      /*分段数据包插入到队列末尾*/
    }
    prev = NULL;
    for (next = qp->q.fragments; next != NULL; next = next->next) {     /*需要遍历队列确定插入位置*/
        if (FRAG_CB(next)->offset >= offset)
            break;    /* bingo! */
        prev = next;
    }

found:      /*找到了在队列中插入位置，在 prev 之后*/

    if (prev) {
        ...      /*检查与 prev 数据包是否有重叠*/
    }

    err = -ENOMEM;

    while (next && FRAG_CB(next)->offset < end) {
        ...      /*检查与 next 数据包是否重叠，用户数据是否有效*/
    }

    FRAG_CB(skb)->offset = offset;

    /*插入分段数据包*/
    skb->next = next;
    if (!next)
        qp->q.fragments_tail = skb;
    if (prev)
        prev->next = skb;
    else
        qp->q.fragments = skb;
```

```
        dev = skb->dev;
        if (dev) {
            qp->iif = dev->ifindex;
            skb->dev = NULL;
        }
        qp->q.stamp = skb->tstamp;
        qp->q.meat += skb->len;       /*收到数据字节数，与最后序列号不一定相同，可能有空洞*/
        qp->ecn |= ecn;
        add_frag_mem_limit(qp->q.net, skb->truesize);
        if (offset == 0)
            qp->q.flags |= INET_FRAG_FIRST_IN;       /*偏移量为 0 表示收到的是第一个分段数据包*/

        fragsize = skb->len + ihl;

        if (fragsize > qp->q.max_size)
            qp->q.max_size = fragsize;

        if (ip_hdr(skb)->frag_off & htons(IP_DF) &&fragsize > qp->max_df_size)
            qp->max_df_size = fragsize;

        if (qp->q.flags == (INET_FRAG_FIRST_IN | INET_FRAG_LAST_IN) &&
                                                qp->q.meat == qp->q.len) {       /*可以重组数据包了*/
            unsigned long orefdst = skb->_skb_refdst;

            skb->_skb_refdst = 0UL;
            err = ip_frag_reasm(qp, prev, dev);       /*重组数据包*/
            skb->_skb_refdst = orefdst;
            return err;         /*成功返回 0*/
        }
        ...
    }
```

ip_frag_queue()函数按分段数据包中偏移量值从小到大，将分段数据包插入到分段数据包队列，如果分段数据包都到齐，则调用 **ip_frag_reasm**()函数重组数据包。

●重组分段数据包

ip_frag_reasm()函数用于重组分段数据包，函数定义如下（/net/ipv4/ip_fragment.c）：

```
static int ip_frag_reasm(struct ipq *qp, struct sk_buff *prev,struct net_device *dev)
/*prev：指向在队列中最后一个接收到的分段数据包的前一个分段数据包*/
{
    struct net *net = container_of(qp->q.net, struct net, ipv4.frags);
    struct iphdr *iph;
    struct sk_buff *fp, *head = qp->q.fragments;       /*分段数据包队列头*/
    int len;
    int ihlen;
    int err;
    int sum_truesize;
```

```
u8 ecn;

ipq_kill(qp);    /*将 ipq 实例从散列表中移出，摘除定时器*/

ecn = ip_frag_ecn_table[qp->ecn];
...
/*最后收到的分段数据包复制一个副本代替原数据包，原数据包取出，作为队列第一个数据包*/
if (prev) {
    head = prev->next;    /*head 指向最后收到的分段数据包，不一定是队列中最后一个数据包*/
    fp = skb_clone(head, GFP_ATOMIC);   /*克隆最后收到的分段数据包*/
    if (!fp)
        goto out_nomem;
    /*fp 代替 head，head 从队列中移出*/
    fp->next = head->next;
    if (!fp->next)
        qp->q.fragments_tail = fp;
    prev->next = fp;
    /*head 指向最后收到的分段数据包*/
    skb_morph(head, qp->q.fragments);
                    /*将队列中第一个数据包复制到 head 指向数据包，/net/core/skbuff.c*/
    head->next = qp->q.fragments->next;
                      /*head 为第一个分段数据包，next 指向队列中第二个数据包*/
    consume_skb(qp->q.fragments);    /*释放队列第一个数据包，由 head 代替了*/
    qp->q.fragments = head;        /*head 指向分段数据包，成为了队列头*/
}

WARN_ON(!head);
WARN_ON(FRAG_CB(head)->offset != 0);    /*队列头分段数据包偏移量须为 0*/

/*分配一个新数据包*/
ihlen = ip_hdrlen(head);    /*IP 报头长度*/
len = ihlen + qp->q.len;    /*重组后数据包总长度*/

err = -E2BIG;
if (len > 65535)
    goto out_oversize;

if (skb_unclone(head, GFP_ATOMIC))      /*不是克隆数据包？ */
    goto out_nomem;

if (skb_has_frag_list(head)) {
    /*如果第一个数据包有分段数据包，则创建一个新数据包，接管分段数据包*/
    struct sk_buff *clone;
    int i, plen = 0;

    clone = alloc_skb(0, GFP_ATOMIC);
```

```
        if (!clone)
                goto out_nomem;
        clone->next = head->next;
        head->next = clone;             /*新数据包插入第一个数据包后面*/
        skb_shinfo(clone)->frag_list = skb_shinfo(head)->frag_list;     /*接管理分段数据包*/
        skb_frag_list_init(head);       /*skb_shinfo(skb)->frag_list = NULL*/
        for (i = 0; i < skb_shinfo(head)->nr_frags; i++)        /*如果存在分散数据块*/
                plen += skb_frag_size(&skb_shinfo(head)->frags[i]);       /*计算分散数据块大小*/
        clone->len = clone->data_len = head->data_len - plen;       /*分段数据包长度*/
        head->data_len -= clone->len;       /*第一个数据包中还保留了分散数据块，如果存在*/
        head->len -= clone->len;
        clone->csum = 0;
        clone->ip_summed = head->ip_summed;
        add_frag_mem_limit(qp->q.net, clone->truesize);
}

skb_push(head, head->data - skb_network_header(head));

sum_truesize = head->truesize;
for (fp = head->next; fp;) {        /*遍历分段数据包队列*/
        bool headstolen;
        int delta;
        struct sk_buff *next = fp->next;

        sum_truesize += fp->truesize;
        if (head->ip_summed != fp->ip_summed)
                head->ip_summed = CHECKSUM_NONE;
        else if (head->ip_summed == CHECKSUM_COMPLETE)
                head->csum = csum_add(head->csum, fp->csum);

        if (skb_try_coalesce(head, fp, &headstolen, &delta)) {        /*将 fp 中数据复制到 head*/
                kfree_skb_partial(fp, headstolen);        /*释放 fp 数据包*/
        } else {
                if (!skb_shinfo(head)->frag_list)
                        skb_shinfo(head)->frag_list = fp;
                head->data_len += fp->len;
                head->len += fp->len;
                head->truesize += fp->truesize;
        }
        fp = next;      /*下一个数据包*/
}       /*遍历队列中数据包结束*/

sub_frag_mem_limit(qp->q.net, sum_truesize);

/*下面是更新重组数据包中参数*/
head->next = NULL;
```

```
        head->dev = dev;
        head->tstamp = qp->q.stamp;
        IPCB(head)->frag_max_size = max(qp->max_df_size, qp->q.max_size);

        iph = ip_hdr(head);
        iph->tot_len = htons(len);
        iph->tos |= ecn;

        if (qp->max_df_size == qp->q.max_size) {
            IPCB(head)->flags |= IPSKB_FRAG_PMTU;
            iph->frag_off = htons(IP_DF);
        } else {
            iph->frag_off = 0;
        }
        ip_send_check(iph);        /*计算 IPv4 报头校验和*/

        IP_INC_STATS_BH(net, IPSTATS_MIB_REASMOKS);
        qp->q.fragments = NULL;
        qp->q.fragments_tail = NULL;
        return 0;
        ...
    }
```

ip_frag_reasm()函数中对最后收到的分段数据包克隆一个副本代替原数据包，复制队列中第一个数据包至最后收到的分段数据包中，并使其成为队列中第一个数据包。然后，遍历队列中的其它数据包，复制数据包中数据至第一个数据包（重组数据包），释放队列中数据包，最后更新重组数据包中的参数。

13.1.7 iptables

Netfilter 子系统在前面概述部分简要介绍过了，就是在数据包接收/发送路径中插入挂载点，在挂载点注册回调函数。数据包到达挂载点后，先调用挂载点注册的回调函数，然后再执行正常的处理流程。一个挂载点可同时注册多个回调函数，执行回调函数时，按优先级从高到低依次执行。有多个网络层协议在数据包处理流程中设置了 Netfilter 挂载点，如 IPv4、IPv6、ARP 等。

Netfilter 子系统提供了一个框架，它支持数据包在网络栈传输路径的各个地方（挂载点）注册回调函数，从而对数据包执行各种操作，如修改地址或端口、丢弃数据包、写入日志等。这些 Netfilter 挂载点为 Netfilter 内核模块提供了基础设施，让它能够通过注册回调函数来执行 Netfilter 子系统的各种任务。

Netfilter 子系统提供了下述功能：

- 数据包选择（iptables）
- 数据包过滤
- 网络地址转换（NAT）
- 数据包操纵
- 连接跟踪
- 网络统计信息收集

下面是一些基于 Linux 内核的 Netfilter 子系统的常见框架：

- IPVS：一种传输层负载均衡解决方案。
- IP sets：一个用户空间工具 ipset 和内核部分组成的框架。IP 集合（IP set）本质上就是一组 IP 地址。

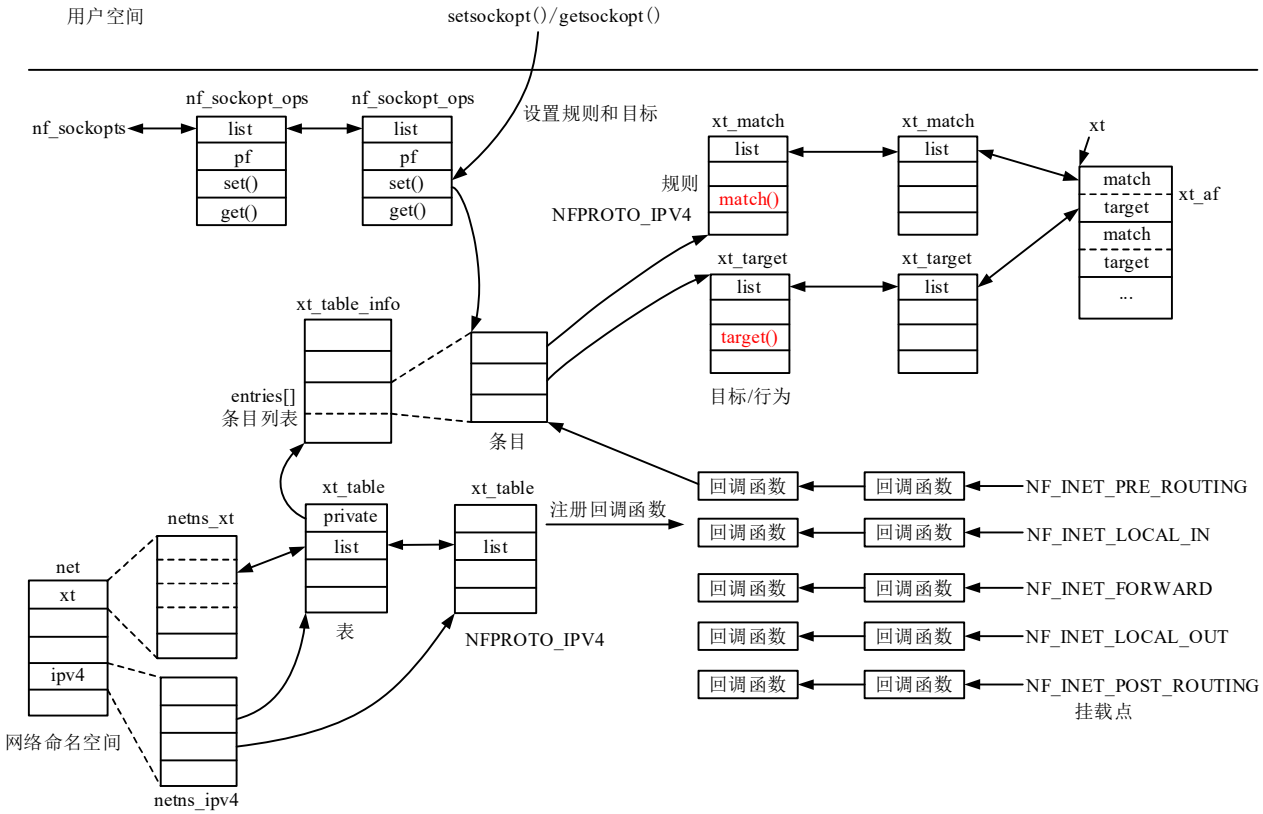
●iptables：iptables 可能是最受欢迎的 Linux 防火墙，它是 Netfilter 前端，为 Netfilter 提供了管理层，让你能够添加和删除 Netfilter 规则、显示统计信息、添加表、将表中的计数器重置为 0，等等。

IPVS、IP sets 框架代码位于/net/netfilter/目录下。

iptables 框架通用代码在/net/netfilter/x_tables.c 文件内实现，IPv4 iptables 代码文件在/net/ipv4/netfilter/目录下，IPv6 iptables 代码文件在/net/ipv6/netfilter/目录下实现。本小节以 IPv4 iptables 框架为例介绍其实现。

1 概述

内核支持 iptables 框架需要选择 IP_NF_IPTABLES 配置选项，iptables 框架如下图所示。iptables 框架适用于多种网络层协议，每种协议中定义了多个表（xt_table 表），每个表用于实现某一项功能，如数据包过滤、NAT 等。在注册 xt_table 表时，同时在相应 Netfilter 挂载点注册回调函数，xt_table 表中包含一个条目列表，每个挂载点对应列表中若干个条目，每个条目中包含规则和目标。回调函数将检查数据包是否与条目中的规则匹配，如果匹配则调用条目中目标的回调函数，执行相应的操作。内核定义了管理通用规则、目标的管理结构，用户进程可通过 setsockopt()/getsockopt()系统调用设置或获取 xt_table 表中的规则和目标信息。



2 通用规则与目标

iptables 中定义了通用规则和目标（行为），规则由 xt_match 结构体表示，目标由 xt_target 结构体表示。内核定义了 xt_af 结构体用于管理同一个网络层协议下的 xt_match 和 xt_target 结构体实例，结构体中主要包含一个 xt_match 实例双链表和一个 xt_target 实例双链表。内核创建了 xt_af 结构体数组，每个数组项对应一个网络层协议。

内核在/net/netfilter/xt_***.c 文件内定义并注册了许多 xt_match 和 xt_target 实例，注册实例，即将实例添加到 xt_af 实例中的双链表。

■数据结构

（1）规则

规则由 xt_match 结构体表示，定义如下（/include/linux/netfilter/x_tables.h）：

```
struct xt_match {
    struct list_head list;      /*双链表成员，将实例添加到 xt_af 实例中双链表*/

    const char name[XT_EXTENSION_MAXNAMELEN];    /*规则名称，查找规则时使用*/
    u_int8_t    revision;       /*版本号*/
    bool (*match)(const struct sk_buff *skb,struct xt_action_param *); /*检查数据包是否与本规则匹配*/
    int (*checkentry)(const struct xt_mtchk_param *);    /*用户向 xt_table 表插入本规则时的检查函数*/
    void (*destroy)(const struct xt_mtdtor_param *);   /*用户从 xt_table 表删除本规则时的回调函数*/
#ifdef CONFIG_COMPAT
    ...
#endif

    struct module *me;
    const char *table;              /*xt_table 表名称*/
    unsigned int matchsize;       /*匹配数据结构大小*/
#ifdef CONFIG_COMPAT
    ...
#endif
    unsigned int hooks;          /*适用的挂载点，由比特位表示*/
    unsigned short proto;
    unsigned short family;        /*网络层协议*/
};
```

xt_match 结构体主要成员简介如下：

●**list**：双链表成员，将实例添加到 xt_af 实例中双链表。

●**name**[]：名称，用于标识规则。

●**hooks**：表示规则适用的挂载点，用比特位来表示挂载点。

●**family**：网络协议类型。

●**match()**：函数指针，检查数据包是否与本规则匹配，基中参数 xt_action_param 结构体定义如下：

```
struct xt_action_param {      /*/include/linux/netfilter/x_tables.h*/
    union {
        const struct xt_match *match;          /*规则*/
        const struct xt_target *target;         /*目标*/
    };
    union {
        const void *matchinfo, *targinfo;      /*规则、目标信息*/
    };
    const struct net_device *in, *out;      /*输入输出网络设备*/
    int fragoff;              /*如果数据包是分段数据包，表示偏移量*/
    unsigned int thoff;       /*传输层报头相对于 skb->data 的偏移量*/
    unsigned int hooknum;         /*挂载点*/
    u_int8_t family;      /*网络层协议*/
    bool hotdrop;         /*是否丢弃数据包*/
```

};

（2）目标

目标或行为由 xt_target 结构体表示，定义如下（/include/linux/netfilter/x_tables.h）：

```
struct xt_target {
    struct list_head list;    /*双链表成员，将实例添加到 xt_af 中双链表*/
    const char name[XT_EXTENSION_MAXNAMELEN];    /*名称*/
    u_int8_t revision;    /*版本号*/

    unsigned int (*target)(struct sk_buff *skb,const struct xt_action_param *);  /*目标/行为回调函数*/
    int (*checkentry)(const struct xt_tgchk_param *);    /*用户向 xt_table 表插入本目标时的检查函数*/
    void (*destroy)(const struct xt_tgdtor_param *);  /*用户从 xt_table 表删除本目标时的回调函数*/
#ifdef CONFIG_COMPAT
    ...
#endif
    struct module *me;

    const char *table;    /*适用的 xt_table 实例名称*/
    unsigned int targetsize;
#ifdef CONFIG_COMPAT
    ...
#endif
    unsigned int hooks;    /*适用的挂载点，由比特位标识*/
    unsigned short proto;
    unsigned short family;    /*协议簇*/
};
```

xt_target 结构体主要成员简介如下：

●**list**：双链表成员，用于将目标添加到 xt_af 实例中双链表。

●**target**：目标执行操作的回调函数。

■接口函数

内核在/net/netfilter/x_tables.c 文件内定义了 xt_af 结构体，用于管理同一网络层协议下的 xt_target 和 xt_match 实例，xt_af 结构体定义如下：

```
struct xt_af {
    struct mutex mutex;
    struct list_head match;    /*管理 xt_match 实例双链表*/
    struct list_head target;    /*管理 xt_target 实例双链表*/
  #ifdef CONFIG_COMPAT
    ...
  #endif
};
```

内核在初始化阶段创建了 xt_af 结构体数组，每个网络层协议对应一个数组项，网络层协议标识定义如下（/include/uapi/linux/netfilter.h）：

```
enum {
```

```
    NFPROTO_UNSPEC =  0,
    NFPROTO_INET   =  1,
    NFPROTO_IPV4   =  2,      /*IPv4*/
    NFPROTO_ARP    =  3,
    NFPROTO_NETDEV =  5,
    NFPROTO_BRIDGE =  7,
    NFPROTO_IPV6   = 10,      /*IPv6*/
    NFPROTO_DECNET = 12,
    NFPROTO_NUMPROTO,
};
```

注册/注销 xt_match、xt_target 实例的接口函数如下（/net/netfilter/x_tables.c）：

●int **xt_register_match**(struct xt_match *match)：注册 xt_match 实例。

●int **xt_register_matches**(struct xt_match *match, unsigned int n)：注册多个 xt_match 实例（数组）。

●void xt_unregister_match(struct xt_match *match)：注销 xt_match 实例。

●void xt_unregister_matches(struct xt_match *match, unsigned int n)：注销多个 xt_match 实例。

●int **xt_register_target**(struct xt_target *target)：注册 xt_target 实例。

●int **xt_register_targets**(struct xt_target *target, unsigned int n)：注册多个 xt_target 实例（数组）。

●void xt_unregister_target(struct xt_target *target)：注销 xt_target 实例。

●void xt_unregister_targets(struct xt_target *target, unsigned int n)：注销多个 xt_target 实例。

注册/注销 xt_match、xt_target 实例的接口函数中主要就是将实例添加/移出 xt_af 实例中的双链表，源代码请读者自行阅读。

3 通用 xt_table 表

iptables 框架中定义了 xt_table 表，每个表用于实现某项功能。iptables 框架适用的每种网络层协议中定义并注册了若干个 xt_table 表（需选择相应的配置选项）。注册 xt_table 表时，会向 Netfilter 挂载点注册相应的回调函数。

用户进程可通过 setsockopt()/getsockopt()系统调用向 xt_table 表设置/获取规则和目标。xt_table 表注册的回调函数会检查数据包是否与表中的规则匹配，若匹配则执行目标中的回调函数，完成相应的操作。

xt_table 表公共层代码在/net/netfilter/x_tables.c 文件内实现。

■数据结构

xt_table 表的主要数据结构有 xt_table 和 xt_table_info，下面分别对其进行介绍。

●xt_table

iptables 中的表由 xt_table 结构体表示，定义如下（/include/linux/netfilter/x_tables.h）：

```
struct xt_table {
    struct list_head list;      /*双链表成员，将实例添加到网络命名空间中的双链表，见下文*/
    unsigned int valid_hooks;   /*xt_table 表中涉及的挂载点*/
                /*有效的挂载点，由比特位表示挂载点，表示表在哪些挂载点注册了回调函数*/
    struct xt_table_info *private;      /*指向 xt_table_info 结构体，主要表示匹配条目信息*/
    struct module *me;          /*模块指针*/
```

```
    u_int8_t af;          /*网络层协议类型*/
    int priority;         /*挂载点注册回调函数的优先级*/
    const char name[XT_TABLE_MAXNAMELEN];       /*表的名称，用于标识表*/
};
```

xt_table 结构体部分成员简介如下：

◎**list**：双链表成员，将 xt_table 实例添加到网络命名空间中的管理双链表。

◎**valid_hooks**：位图，每个比特位表示一个挂载点，表在此挂载点注册回调函数。

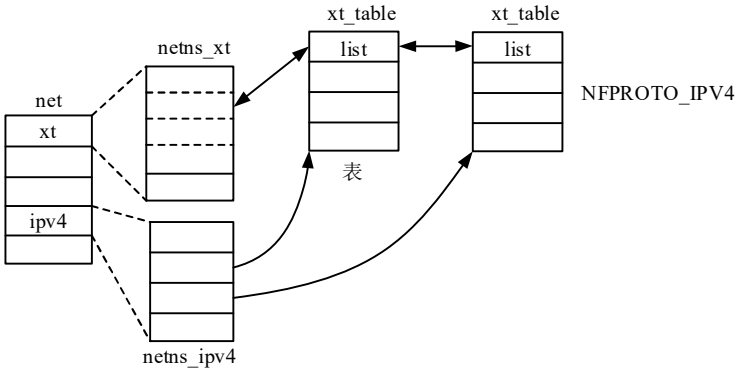
◎**af**：网络层协议。

◎**priority**：回调函数优先级。

◎**name[]**：表名称，用于标识表。

◎**private**：指向 xt_table_info 结构体，表示表的具体信息，见下文。

在网络命名空间 net 结构体中为每个网络层协议定义了一个双链表，用于管理其下 xt_table 实例。另外，net 中 netns_ipv4 结构体成员中包含指向 IPv4 中各 xt_table 实例的指针，如下图所示。



net 结构体中相关成员如下所示：

```
struct net {
    ...
    struct netns_ipv4   ipv4;      /*IPv4 网络协议资源*/
    ...
#ifdef CONFIG_NETFILTER     /*#if CONFIG_NETFILTER 开始*/
    struct netns_nf     nf;          /*proc 文件系统中的信息，/include/net/netns/netfilter.h*/
    struct netns_xt     xt;          /*管理 xt_table 实例的双链表数组，/include/net/netns/x_tables.h*/
    ...
  #if defined(CONFIG_NF_TABLES) || defined(CONFIG_NF_TABLES_MODULE)
    struct netns_nftables     nft;
  #endif
    ...
#endif     /*#if CONFIG_NETFILTER 结束*/
    ...
}
```

net 结构体中相关成员简介如下：

◎**xt**：netns_xt 结构体成员，定义如下（/include/net/netns/x_tables.h）：

```
struct netns_xt {
    struct list_head tables[NFPROTO_NUMPROTO];
                                        /*双链表数组，每个网络层协议对应一个双链表*/
    bool notrack_deprecated_warning;
```

```
    bool clusterip_deprecated_warning;
  #if defined(CONFIG_BRIDGE_NF_EBTABLES) || \
    defined(CONFIG_BRIDGE_NF_EBTABLES_MODULE)
    ...
  #endif
};
```

netns_xt 结构体中 tables[]成员是一个双链表数组，每个数组项对应一个网络层协议。

◎ipv4：netns_ipv4 结构体成员，netns_ipv4 结构体中 xt_table 表相关成员如下（/include/net/netns/ipv4.h）：

```
struct netns_ipv4 {
    ...
  #ifdef CONFIG_NETFILTER
    struct xt_table     *iptable_filter;     /*数据包过滤表*/
    struct xt_table     *iptable_mangle;     /*管理表*/
    struct xt_table     *iptable_raw;        /*原始表*/
    struct xt_table     *arptable_filter;    /*ARP 过滤表*/
    #ifdef CONFIG_SECURITY
      struct xt_table   *iptable_security;   /*安全*/
    #endif
    struct xt_table     *nat_table;      /*NAT 表*/
  #endif
    ...
}
```

IPv4 中共定义了 6 个 xt_table 表，例如：iptable_filter 表示数据包过滤表，用于实现防火墙。

●xt_table_info

xt_table 结构体中 private 成员指向 xt_table_info 结构体，定义如下（/include/linux/netfilter/x_tables.h）：

```
struct xt_table_info {
    unsigned int size;              /*条目列表（entries[]成员）大小*/
    unsigned int number;        /*条目数量，entries[]成员*/
    unsigned int initial_entries;    /*初始条目数量*/

    unsigned int hook_entry[NF_INET_NUMHOOKS];   /*各挂载点起始条目在条目列表中的偏移量*/
    unsigned int underflow[NF_INET_NUMHOOKS];
    unsigned int stacksize;
    unsigned int __percpu *stackptr;
    void ***jumpstack;

    unsigned char entries[0] __aligned(8);       /*条目列表*/
                                        /*IPv4 中为 ipt_standard 数组加上 ipt_error 实例*/
};
```

xt_table_info 结构体中主要包含一个条目数组（entries[]成员），每个条目中包含匹配规则和目标等。每个挂载点在条目列表中对应若干个条目。挂载点注册的回调函数将数据包与对应各条目中的规则匹配，若匹配成功则执行条目中目标的回调函数。

每种网络层协议定义的条目结构有所不同，条目将关联到通用的规则和目标。例如，IPv4 中条目由结构体 ipt_standard 表示，详见下文。

hook_entry[]数组表示挂载点对应起始条目要条目列表（entries[]成员）中的偏移量。

接口函数 **xt_alloc_table_info**(unsigned int size)用于分配 xt_table_info 实例，size 表示条目列表大小（字节数）。

用户进程可通过 setsockopt()/getsockopt()系统调用设置/获取 xt_table_info 实例中的条目信息。

●初始化

初始化函数 xt_init()中将创建全局 xt_af 结构体数组（管理通用规则与目标），初始化网络命名空间 net.xt 成员等，函数代码如下（/net/netfilter/x_tables.c）：

```
static int __init xt_init(void)
{
    unsigned int i;
    int rv;

    for_each_possible_cpu(i) {
        seqcount_init(&per_cpu(xt_recseq, i));
    }

    xt = kmalloc(sizeof(struct xt_af) * NFPROTO_NUMPROTO, GFP_KERNEL);
                                                    /*创建全局的 xt_af 结构体数组*/
    ...     /*错误处理*/

    for (i = 0; i < NFPROTO_NUMPROTO; i++) {     /*初始化 xt_af 结构体数组*/
        mutex_init(&xt[i].mutex);
      #ifdef CONFIG_COMPAT
        ...
      #endif
        INIT_LIST_HEAD(&xt[i].target);      /*初始化双链表，管理目标*/
        INIT_LIST_HEAD(&xt[i].match);       /*初始化双链表，管理规则*/
    }
    rv = register_pernet_subsys(&xt_net_ops);      /*初始化函数中初始化 net.xt.tables[]双链表数组*/
    ...
    return rv;
}
```

xt_init()函数中创建了全局的 xt_af 结构体数组，并进行了初始化；注册了 pernet_operations 结构体实例 xt_net_ops，其初始化函数中将初始化网络命名空间中的 net.xt.tables[]双链表数组。

■接口函数

使用 xt_table 表实现某项功能的子系统，需要定义并注册 xt_table 实例，并在挂载点注册相应的回调函数。下面介绍注册 xt_table 表及挂载点回调函数的接口函数，接口函数主要定义在/net/netfilter/x_tables.c 文件内。

●注册 xt_table 实例

在注册 xt_table 实例前，需要定义 xt_table 实例，创建并设置 xt_table_info 实例，然后调用接口函数 xt_register_table()注册 xt_table 实例，函数定义如下（/net/netfilter/x_tables.c）：

```
struct xt_table *xt_register_table(struct net *net,const struct xt_table *input_table,
                                   struct xt_table_info *bootstrap,struct xt_table_info *newinfo)
```

```
/*input_table：需要注册的 xt_table 实例，bootstrap：初始 xt_table_info 实例，
*newinfo：新 xt_table_info 实例。
*/
{
    int ret;
    struct xt_table_info *private;
    struct xt_table *t, *table;

    /*复制 input_table 实例至 table*/
    table = kmemdup(input_table, sizeof(struct xt_table), GFP_KERNEL);
    ...    /*错误处理*/

    mutex_lock(&xt[table->af].mutex);
    list_for_each_entry(t, &net->xt.tables[table->af], list) {
                                        /*在网络命名空间的双链表中查找是否有相同名称的表*/
        if (strcmp(t->name, table->name) == 0) {      /*若已有相同名称的表，返回错误码*/
            ret = -EEXIST;
            goto unlock;
        }
    }

    table->private = bootstrap;        /*指向初始 xt_table_info 实例*/

    if (!xt_replace_table(table, 0, newinfo, &ret)) /*table->private = newinfo 等，/net/netfilter/x_tables.c*/
        goto unlock;

    private = table->private;
    pr_debug("table->private->number = %u\n", private->number);

    private->initial_entries = private->number;      /*重置初始条目数*/

    list_add(&table->list, &net->xt.tables[table->af]);
                                        /*将 xt_table 实例添加到网络命名空间中双链表*/
    mutex_unlock(&xt[table->af].mutex);
    return table;
    ...
}
```

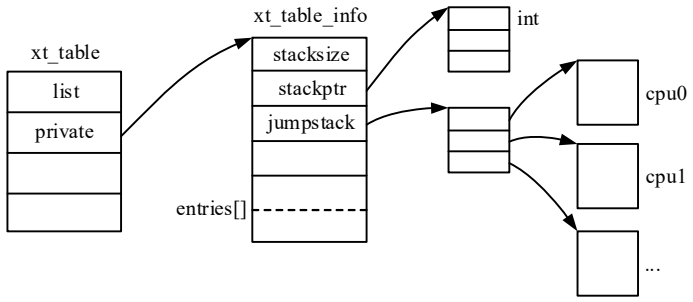
xt_register_table()函数比较简单，主要工作是复制 input_table 指向的 xt_table 实例，关联 newinfo 指向的 xt_table_info 实例，最后将 xt_table 实例添加到网络命名空间中的双链表。

xt_register_table()函数中调用 xt_replace_table()函数用 newinfo 指向的 xt_table_info 实例代替 xt_table 实例原先关联的 xt_table_info 实例，代码如下（/net/netfilter/x_tables.c）。

```
struct xt_table_info *xt_replace_table(struct xt_table *table,unsigned int num_counters,
                                                    struct xt_table_info *newinfo,int *error)
/*table：指向 xt_table 实例，num_counters：原 xt_table_info 实例中条目数，
*newinfo：指向新 xt_table_info 实例，error：保存返回码。*/
{
```

```
    struct xt_table_info *private;
    int ret;

    ret = xt_jumpstack_alloc(newinfo);        /*分配栈空间，/net/netfilter/x_tables.c*/
    ...      /*错误处理*/

    local_bh_disable();
    private = table->private;        /*原 xt_table_info 实例*/
    if (num_counters != private->number) {
        ...          /*错误处理*/
    }

    newinfo->initial_entries = private->initial_entries;     /*赋值初始条目数*/
    smp_wmb();
    table->private = newinfo;        /*关联新 xt_table_info 实例*/

    local_bh_enable();
#ifdef CONFIG_AUDIT
    ...
#endif
    return private;       /*返回原 xt_table_info 实例*/
}
```

xt_replace_table()函数中调用 xt_jumpstack_alloc()函数为 xt_table_info 实例中 stackptr、jumpstack 成员分配空间，如下图所示：



●注册回调函数

使用 xt_table 表的子系统注册 xt_table 实例后，还需要向挂载点注册回调函数，注册函数有：

（1）**xt_hook_link()**：向表适用的挂载点注册相同的回调函数。

（2）**nf_register_hooks()/nf_register_hook()**：向指定挂载点注册指定的回调函数。

nf_register_hooks()/nf_register_hook()函数在本节的概述部分已经介绍过了，用于向（多个）挂载点注册指定的回调函数。下面看一下 xt_hook_link()函数的实现。

xt_hook_link()函数代码如下（/net/netfilter/x_tables.c）：

```
struct nf_hook_ops *xt_hook_link(const struct xt_table *table, nf_hookfn *fn)
/*table：xt_table 表，fn：回调函数指针*/
{
    unsigned int hook_mask = table->valid_hooks;      /*适用挂载点，用比特位表示*/
    uint8_t i, num_hooks = hweight32(hook_mask);      /*适用挂载点数量*/
```

```
    uint8_t hooknum;
    struct nf_hook_ops *ops;
    int ret;

    ops = kmalloc(sizeof(*ops) * num_hooks, GFP_KERNEL);       /*分配 nf_hook_ops 实例数组*/
    if (ops == NULL)
        return ERR_PTR(-ENOMEM);

    for (i = 0, hooknum = 0; i < num_hooks && hook_mask != 0;hook_mask >>= 1, ++hooknum) {
        if (!(hook_mask & 1))          /*设置 nf_hook_ops 实例数组*/
            continue;
        ops[i].hook         = fn;         /*设置回调函数*/
        ops[i].owner        = table->me;
        ops[i].pf           = table->af;     /*网络层协议类型*/
        ops[i].hooknum    = hooknum;    /*挂载点编号*/
        ops[i].priority = table->priority;     /*回调函数优先级*/
        ++i;
    }

    ret = nf_register_hooks(ops, num_hooks);       /*注册 nf_hook_ops 实例数组*/
    ...        /*错误处理*/
    return ops;         /*返回 nf_hook_ops 实例数组指针*/
}
```

xt_hook_link()函数中创建 nf_hook_ops 实例数组，xt_table 表中标识的有效挂载点对应一个实例，对实例数组项（nf_hook_ops 实例）进行设置并注册。xt_hook_link()函数中在各挂载点注册的回调函数都为 **fn()**。

■用户接口

用户进程可通过 setsockopt()/getsockopt()系统调用设置或获取 xt_table 表中的规则和目标。网络层协议需要注册 nf_sockopt_ops 实例，用于处理 setsockopt()/getsockopt()系统调用。

●注册 nf_sockopt_ops 实例

nf_sockopt_ops 结构体定义如下（/include/linux/netfilter.h）：

```
struct nf_sockopt_ops {
    struct list_head list;         /*双链表成员，将实例添加到双链表 nf_sockopts*/
    u_int8_t pf;                   /*网络层协议*/

    int set_optmin;         /*设置规则操作中最小选项值*/
    int set_optmax;         /*设置规则操作中最大选项值*/
    int (*set)(struct sock *sk, int optval, void __user *user, unsigned int len);     /*设置条目*/
  #ifdef CONFIG_COMPAT
    ...
  #endif
    int get_optmin;       /*获取规则操作中最小选项值*/
    int get_optmax;       /*获取规则操作中最大选项值*/
    int (*get)(struct sock *sk, int optval, void __user *user, int *len);       /*获取条目（信息）*/
```

```
    #ifdef CONFIG_COMPAT
        ...
    #endif
        struct module *owner;
};
```

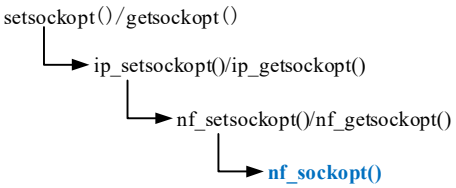
内核在/net/netfilter/nf_sockopt.c 文件内定义了全局双链表 nf_sockopts 用于管理 nf_sockopt_ops 实例，每个支持 Netfilter 的网络层协议需要定义并注册 nf_sockopt_ops 实例，注册函数为 nf_register_sockopt()。

nf_register_sockopt()函数定义如下（/net/netfilter/nf_sockopt.c）：

```
int nf_register_sockopt(struct nf_sockopt_ops *reg)
{
    struct nf_sockopt_ops *ops;
    int ret = 0;

    mutex_lock(&nf_sockopt_mutex);
    list_for_each_entry(ops, &nf_sockopts, list) {      /*遍历全局双链表 nf_sockopts*/
        if (ops->pf == reg->pf&&
            (overlap(ops->set_optmin, ops->set_optmax,reg->set_optmin, reg->set_optmax)
             || overlap(ops->get_optmin, ops->get_optmax,reg->get_optmin, reg->get_optmax))) {
                ...     /*错误处理，不能注册*/
        }
    }
    list_add(&reg->list, &nf_sockopts);      /*将实例添加到全局双链表 nf_sockopts*/
out:
    mutex_unlock(&nf_sockopt_mutex);
    return ret;
}
```

●设置/获取规则

用户进程通过 setsockopt()/getsockopt()系统调用设置/获取表中条目信息，函数调用关系如下：



设置/获取条目信息操作最终由 nf_sockopt()函数完成，定义如下（/net/netfilter/nf_sockopt.c）：

```
static int nf_sockopt(struct sock *sk, u_int8_t pf, int val,char __user *opt, int *len, int get)
{
    struct nf_sockopt_ops *ops;
    int ret;

    ops = nf_sockopt_find(sk, pf, val, get);          /*查找 nf_sockopt_ops 实例*/
    ...     /*错误处理*/

    if (get)
        ret = ops->get(sk, val, opt, len);          /*获取规则*/
```

```
        else
            ret = ops->set(sk, val, opt, *len);        /*设置规则*/
        module_put(ops->owner);
        return ret;
    }
```

nf_sockopt()函数内调用网络层协议定义的 nf_sockopt_ops 实例中的 set()/get()函数设置/获取条目信息。

4 IPv4 表

前面介绍的是 iptables 通用框架。在 iptables 通用框架适用的每种网络层协议中，包含多个表，每个表在网络层协议接收/发送数据包路径中的挂载点注册回调函数。在回调函数中，检查数据包是否匹配表中的条目，如果匹配则执行条目中目标的回调函数，以实现对数据包的某项操作。下面以 IPv4 中的 iptables 为例，介绍其实现。

Linux 内核 IPv4 网络层协议实现中，在数据包处理流程中共设置了 5 个挂载点，如下：

●**NF_INET_PRE_ROUTING**：在接收路径中，在执行路由选择前调用此挂载点注册的回调函数。

●**NF_INET_LOCAL_IN**：在接收路径中，在将数据包传递给本机传输层协议前调用此挂载点注册的回调函数。

●**NF_INET_FORWARD**：转发数据包时，在转发前调用此挂载点注册的回调函数。

●**NF_INET_LOCAL_OUT**：发送本地产生数据包时，在执行路由选择后调用此挂载点注册的回调函数。

●**NF_INET_POST_ROUTING**：发送本地数据包和转发数据包时，在完成最后发送前（到达邻居子系统）调用此挂载点注册的回调函数。

IPv4 中默认设置了 6 个 xt_table 表（需选择相应的配置选项），网络命名空间 netns_ipv4 结构体成员中包含指向这些表的指针成员，如下所示：

```
struct netns_ipv4 {
    ...
  #ifdef CONFIG_NETFILTER
    struct xt_table     *iptable_filter;     /*数据包过滤表，实现防火墙*/
    struct xt_table     *iptable_mangle;     /*管理表，可对数据包进行任何操作*/
    struct xt_table     *iptable_raw;        /*原始表，优先级最高，*/
    struct xt_table     *arptable_filter;    /*ARP 数据包过滤表*/
    #ifdef CONFIG_SECURITY
      struct xt_table   *iptable_security;    /*安全表*/
    #endif
    struct xt_table     *nat_table;       /*NAT 表，需支持连接跟踪*/
  #endif
    ...
}
```

以上各表的定义和注册，以及回调函数的注册在/net/ipv4/netfilter/iptable_***.c 文件内实现。

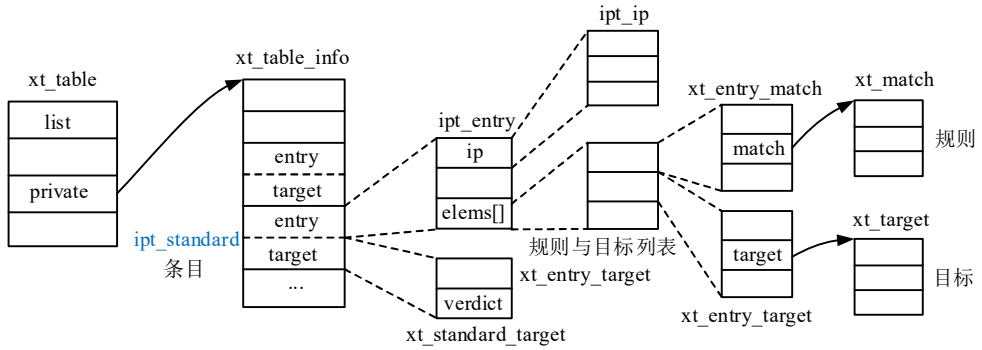
用户进程可通过 setsockopt()/getsockopt()系统调用为各表设置/获取条目。下面将介绍在 IPv4 表中条目的表示、表的注册、设置/获取条目的操作，最后以数据包过滤表为例介绍其实现和功能。

■条目表示

在 IPv4 的 xt_table 表中，条目由 ipt_standard 结构体表示。在注册 xt_table 表或设置表中规则时，通过 ipt_replace 结构体传递条目数据等参数，下面介绍这两个数据结构的定义。

●ipt_standard

xt_table 实例关联的 xt_table_info 实例中最后是一个条目列表，条目由规则和目标组成。在 IPv4 中每个条目，由 ipt_standard 结构体表示，如下图所示：



ipt_standard 结构体定义如下（/include/linux/netfilter_ipv4/ip_tables.h）：

```
struct ipt_standard {
    struct ipt_entry entry;              /*条目，/include/uapi/linux/netfilter_ipv4/ip_tables.h*/
    struct xt_standard_target target;    /*标准目标，/include/uapi/linux/netfilter/x_tables.h*/
};
```

ipt_standard 结构体成员简介如下：

●**entry**：ipt_entry 结构体成员，包含主要的条目信息，结构体定义如下：

```
struct ipt_entry {
    struct ipt_ip ip;            /*IP 报头信息，表示数据包是否匹配本条目，
                                  */include/uapi/linux/netfilter_ipv4/ip_tables.h*/
    unsigned int nfcache;
    __u16 target_offset;     /*ipt_entry 加规则（可能多个）的大小，也就是目标的起始位置*/
    __u16 next_offset;       /*ipt_entry 加规则加目标的大小*/
    unsigned int comefrom;   /* Back pointer */
    struct xt_counters counters;     /* Packet and byte counters. */
    unsigned char elems[0];          /*规则和目标列表*/
};
```

ipt_entry 结构体主要成员简介如下：

◎**ip**：ipt_ip 结构体成员，表示匹配 IP 报头信息，结构体定义如下：

```
struct ipt_ip {
    struct in_addr src, dst;        /*源、目的 IP 地址*/
    struct in_addr smsk, dmsk;      /*源、目的 IP 地址掩码*/
    char iniface[IFNAMSIZ], outiface[IFNAMSIZ];
    unsigned char iniface_mask[IFNAMSIZ], outiface_mask[IFNAMSIZ];
    __u16 proto;          /*协议类型*/
    __u8 flags;           /*标志*/
    __u8 invflags;        /*反向标志*/
};
```

◎elems[0]：由 xt_entry_match 和 xt_standard_target 结构体组成的数组，表示规则和目标列表（规则在前，目标在后），这两个结构体定义如下（/include/uapi/linux/netfilter/x_tables.h）：

```
struct xt_entry_match {
    union {      /*联合体*/
        struct {      /*由用户使用*/
            __u16 match_size;
            char name[XT_EXTENSION_MAXNAMELEN];     /*由名称标识规则*/
            __u8 revision;
        } user;

        struct {       /*由内核使用*/
            __u16 match_size;
            struct xt_match *match;       /*指向 xt_match 实例*/
        } kernel;
        __u16 match_size;     /*总大小*/
    } u;

    unsigned char data[0];
};
```

xt_entry_target 结构体定义如下（/include/uapi/linux/netfilter/x_tables.h）：

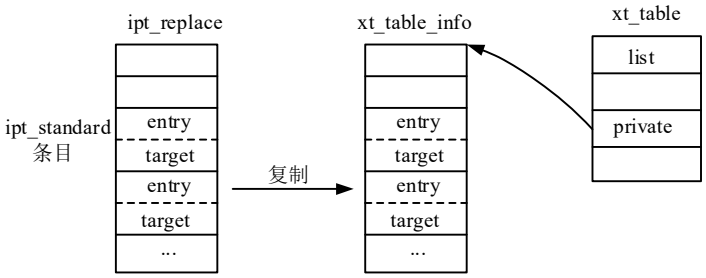
```
struct xt_entry_target {
    union {      /*联合体*/
        struct {      /*由用户使用*/
            __u16 target_size;
            char name[XT_EXTENSION_MAXNAMELEN];        /*目标名称*/
            __u8 revision;
        } user;
        struct {      /*由内核使用*/
            __u16 target_size;
            struct xt_target *target;        /*指向 xt_target 实例*/
        } kernel;
        __u16 target_size;
    } u;

    unsigned char data[0];
};
```

●target：ipt_standard 结构体 target 成员为 xt_standard_target 结构体，定义如下：

```
struct xt_standard_target {
    struct xt_entry_target target;        /*/include/uapi/linux/netfilter/x_tables.h*/
    int verdict;        /*数据包处理结果指示，如 NF_ACCEPT 等*/
};
```

●ipt_replace

内核在注册 IPv4 表及设置表规则时，通过 ipt_replace 结构体传递条目数据等参数，如下图所示：



ipt_replace 结构体定义如下（/include/uapi/linux/netfilter_ipv4/ip_tables.h）：

```
struct ipt_replace {
    char name[XT_TABLE_MAXNAMELEN];          /*关联 xt_table 实例名称*/
    unsigned int valid_hooks;      /*位图，表示适用的挂载点*/
    unsigned int num_entries;      /*条目数量*/
    unsigned int size;         /*条目列表大小，ipt_standard 结构体数组加上 ipt_error 的字节数*/

    unsigned int hook_entry[NF_INET_NUMHOOKS];
                                   /*挂载点对应起始 ipt_standard 实例，在条目列表中的偏移量*/
    unsigned int underflow[NF_INET_NUMHOOKS];      /*初始值同 hook_entry[]*/

    unsigned int num_counters;
    struct xt_counters __user *counters;
    struct ipt_entry entries[0];        /*定义防火墙规则，/include/uapi/linux/netfilter_ipv4/ip_tables.h*/
                                    /*占位符，表示条目列表，实际为 ipt_standard 结构体数组*/
};
```

ipt_replace 结构体主要成员简介如下：

●**name[]**：关联 xt_table 实例名称。

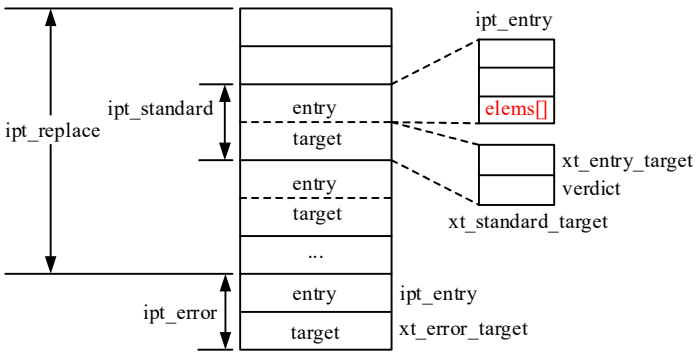
●**num_entries**：条目数量。

●**entries[]**：ipt_entry 结构体数组（空数组），在创建 ipt_replace 实例时，为 ipt_standard 结构体数组。

ipt_replace 结构体实例依据 xt_table 实例创建，接口函数为 ipt_alloc_initial_table()，定义如下：

```
void *ipt_alloc_initial_table(const struct xt_table *info)       /*/net/ipv4/netfilter/ip_tables.c*/
{
    return xt_alloc_initial_table(ipt, IPT);        /*/net/netfilter/xt_repldata.h*/
}
```

xt_alloc_initial_table()定义在/net/netfilter/xt_repldata.h 头文件，此宏内将创建并设置 ipt_replace 实例，最后返回 ipt_replace 实例指针，如下图所示：



ipt_replace 实例中最后是一个 ipt_standard 结构体数组，数组项数与 xt_table 实例适用的挂载点数量相

同，也就是说每个适有的挂载点对应一个数组项。ipt_standard 实例 entry 成员（ipt_entry 实例）中的 elems[]成员此时为空。

ipt_replace 实例后面紧接着一个 ipt_error 结构体实例。ipt_replace 实例中 size 成员表示 ipt_standard 结构体数组加上 ipt_error 结构体实例的大小（字节数）。

■注册 IPv4 表

IPv4 在注册 xt_table 实例前通常调用前面介绍的 ipt_alloc_initial_table()函数，依据 xt_table 实例创建 ipt_replace 实例，然后才调用接口函数 ipt_register_table()注册 xt_table 实例。

ipt_register_table()函数定义如下（/net/ipv4/netfilter/ip_tables.c）：

```
struct xt_table *ipt_register_table(struct net *net,const struct xt_table *table,const struct ipt_replace *repl)
/*repl：指向依据 xt_table 实例创建的 ipt_replace 实例*/
{
    int ret;
    struct xt_table_info *newinfo;
    struct xt_table_info bootstrap = {0};
    void *loc_cpu_entry;
    struct xt_table *new_table;

    newinfo = xt_alloc_table_info(repl->size);     /*分配 xt_table_info 实例，/net/netfilter/x_tables.c*/
                                                   /*大小为 xt_table_info 实例大小加 repl->size*/
                          /*repl->size 为 ipt_standard 结构体数组加上 ipt_error 结构体实例的大小*/
    ...     /*错误处理*/

    loc_cpu_entry = newinfo->entries;       /*指向 xt_table_info 实例中的条目信息，此时为空*/
    memcpy(loc_cpu_entry, repl->entries, repl->size);
                                            /*将 ipt_replace 实例中条目复制到 xt_table_info 实例*/
    ret = translate_table(net, newinfo, loc_cpu_entry, repl);
                                    /*检查或转换用户提供的条目信息，/net/ipv4/netfilter/ip_tables.c*/
    ...     /*错误处理*/

    new_table = xt_register_table(net, table, &bootstrap, newinfo);       /*注册通用 xt_table 实例*/
    ...     /*错误处理*/

    return new_table;       /*返回 xt_table 实例指针*/
    ...
}
```

ipt_register_table()函数中调用 xt_alloc_table_info()函数创建 xt_table_info 实例，实例最后保存来自实例 ipt_replace 中的条目信息（复制）。ipt_register_table()函数随后调用 xt_register_table()函数注册 xt_table 实例。

■设置/获取规则

用户进程通过 setsockopt()/getsockopt()系统调用可设置/获取表中的规则和目标信息。前面介绍过，这两个系统调用内调用网络层协议注册的 nf_sockopt_ops 实例中的相应函数，处理 Netfilter 相关的选项。

下面介绍 IPv4 中 nf_sockopt_ops 实例的实现。

●初始化

IPv4 中 iptables 初始化函数为 ip_tables_init()，定义如下（/net/ipv4/netfilter/ip_tables.c）：

```
static int __init ip_tables_init(void)
{
    int ret;
    ret = register_pernet_subsys(&ip_tables_net_ops);  /*初始化函数完成在 proc 文件系统中初始化*/
    ...
    ret = xt_register_targets(ipt_builtin_tg, ARRAY_SIZE(ipt_builtin_tg));
                                                        /*注册标准、标准错误目标*/
    ...
    ret = xt_register_matches(ipt_builtin_mt, ARRAY_SIZE(ipt_builtin_mt));    /*注册"icmp"规则*/
    ...
    ret = nf_register_sockopt(&ipt_sockopts);
                    /*注册 ipt_sockopts 实例，用于处理 setsockopt()/getsockopt()系统调用*/
    ...
    return 0;
    ...
}
```

ip_tables_init()函数中注册了两个目标和一个规则，最后注册了 IPv4 网络协议的 nf_sockopt_ops 结构体实例 ipt_sockopts，用于处理 setsockopt()/getsockopt()系统调用。

IPv4 中与 iptables 相关的套接字选项及选项值表示定义在/include/uapi/linux/netfilter_ipv4/ip_tables.h 头文件，选项如下：

```
#define   IPT_BASE_CTL          64

#define   IPT_SO_SET_REPLACE            (IPT_BASE_CTL)   /*设置 ipt_replace 实例，设置规则*/
#define   IPT_SO_SET_ADD_COUNTERS       (IPT_BASE_CTL + 1)
#define   IPT_SO_SET_MAX                IPT_SO_SET_ADD_COUNTERS

#define   IPT_SO_GET_INFO               (IPT_BASE_CTL)       /*获取信息*/
#define   IPT_SO_GET_ENTRIES            (IPT_BASE_CTL + 1)
#define   IPT_SO_GET_REVISION_MATCH     (IPT_BASE_CTL + 2)
#define   IPT_SO_GET_REVISION_TARGET    (IPT_BASE_CTL + 3)
#define   IPT_SO_GET_MAX                IPT_SO_GET_REVISION_TARGET
```

以上每个选项需要通过一个数据结构来传递选项值，例如：IPT_SO_SET_REPLACE 选项由 ipt_replace 结构体传递选项值，IPT_SO_GET_INFO 选项由 ipt_getinfo 结构体传递选项值等。

IPv4 网络层协议中的 nf_sockopt_ops 结构体实例 **ipt_sockopts** 定义如下(/net/ipv4/netfilter/ip_tables.c)：

```
static struct nf_sockopt_ops ipt_sockopts = {
    .pf         = PF_INET,
    .set_optmin     = IPT_BASE_CTL,
    .set_optmax     = IPT_SO_SET_MAX+1,
    .set        = do_ipt_set_ctl,       /*设置规则（参数）函数*/
#ifdef CONFIG_COMPAT
    .compat_set     = compat_do_ipt_set_ctl,
#endif
```

```
    .get_optmin    = IPT_BASE_CTL,
    .get_optmax    = IPT_SO_GET_MAX+1,
    .get           = do_ipt_get_ctl,        /*获取规则（参数）函数*/
#ifdef CONFIG_COMPAT
    .compat_get    = compat_do_ipt_get_ctl,
#endif
    .owner         = THIS_MODULE,
};
```

其中 do_ipt_set_ctl()函数用于处理 setsockopt()系统调用，do_ipt_get_ctl()函数用于处理 getsockopt()系统调用。

●设置规则

setsockopt()系统调用将最终调用 ipt_sockopts 实例中 set()函数，即 do_ipt_set_ctl()函数，完成设置规则/参数的操作，函数定义如下（/net/ipv4/netfilter/ip_tables.c）：

```
static int do_ipt_set_ctl(struct sock *sk, int cmd, void __user *user, unsigned int len)
/*user：指向用户传递的 ipt_replace 实例*/
{
    int ret;

    if (!ns_capable(sock_net(sk)->user_ns, CAP_NET_ADMIN))
        return -EPERM;

    switch (cmd) {
    case IPT_SO_SET_REPLACE:        /*选项名称*/
        ret = do_replace(sock_net(sk), user, len);      /*重置规则，/net/ipv4/netfilter/ip_tables.c*/
        break;

    case IPT_SO_SET_ADD_COUNTERS:
        ret = do_add_counters(sock_net(sk), user, len, 0);
        break;

    default:
        ...
    }
    return ret;
}
```

do_ipt_set_ctl()函数在处理 **IPT_SO_SET_REPLACE** 选项时，调用 do_replace()函数。do_replace()函数内前半部分工作与注册 xt_table 实例时类似，依 ipt_replace 实例创建 xt_table_info 实例，然后判断是否要替换 xt_table 实例中原 xt_table_info 实例。

●获取参数

getsockopt()系统调用用于获取参数值，处理函数为 do_ipt_get_ctl()，定义如下：

```
static int do_ipt_get_ctl(struct sock *sk, int cmd, void __user *user, int *len)
/*/net/ipv4/netfilter/ip_tables.c*/
```

```
{
    int ret;

    if (!ns_capable(sock_net(sk)->user_ns, CAP_NET_ADMIN))
        return -EPERM;

    switch (cmd) {
    case IPT_SO_GET_INFO:
        ret = get_info(sock_net(sk), user, len, 0);        /*获取信息由 ipt_getinfo 结构体表示*/
        break;

    case IPT_SO_GET_ENTRIES:
        ret = get_entries(sock_net(sk), user, len);   /*获取条目信息，由 ipt_get_entries 结构体表示*/
        break;

    case IPT_SO_GET_REVISION_MATCH:
    case IPT_SO_GET_REVISION_TARGET: {
        ...
    }
    return ret;
}
```

do_ipt_get_ctl()函数内针对不同的选项，调用不同的处理函数，具体函数源代码请读者自行阅读。

■数据包过滤表

IPv4 中默认设置了 6 个 xt_table 表（需选择相应的配置选项），分别在/net/ipv4/netfilter/iptable_***.c 文件内实现。下面以数据包过滤表为例，介绍其实现。

数据包过滤表（防火墙）实现在/net/ipv4/netfilter/iptable_filter.c 文件内，需选择 IP_NF_FILTER 配置选项。

●注册过滤表

数据包过滤表定义如下：

```
#define FILTER_VALID_HOOKS     ((1 << NF_INET_LOCAL_IN) | \
                                (1 << NF_INET_FORWARD) | \
                                (1 << NF_INET_LOCAL_OUT))        /*注册回调函数的挂载点*/

static const struct xt_table packet_filter = {        /*数据包过滤表实例*/
    .name           = "filter",            /*名称*/
    .valid_hooks    = FILTER_VALID_HOOKS,
    .me             = THIS_MODULE,
    .af             = NFPROTO_IPV4,          /*网络层协议*/
    .priority       = NF_IP_PRI_FILTER,     /*回调函数优先级（0）*/
};
```

数据包过滤表初始化函数 iptable_filter_init()定义如下：

```
static int __init iptable_filter_init(void)
```

```
{
    int ret;

    ret = register_pernet_subsys(&iptable_filter_net_ops);      /*注册 pernet_operations 实例*/
    ...    /*错误处理*/

    /*注册各挂载点的回调函数*/
    filter_ops = xt_hook_link(&packet_filter, iptable_filter_hook);
                                                    /*回调函数都为 iptable_filter_hook()*/
    ...    /*错误处理*/
    return ret;
}
```

iptable_filter_init()函数中注册了 pernet_operations 实例 iptable_filter_net_ops，注册相关挂载点的回调函数为 iptable_filter_hook()。回调函数后面将介绍，下面看一下 pernet_operations 实例的定义：

```
static struct pernet_operations iptable_filter_net_ops = {
    .init = iptable_filter_net_init,      /*初始化函数，注册过滤表*/
    .exit = iptable_filter_net_exit,
};
```

初始化函数 iptable_filter_net_init()主要工作是注册数据包过滤表，代码如下：

```
static int __net_init iptable_filter_net_init(struct net *net)
{
    struct ipt_replace *repl;

    repl = ipt_alloc_initial_table(&packet_filter);      /*创建 ipt_replace 实例*/
    ...    /*错误处理*/

    ((struct ipt_standard *)repl->entries)[1].target.verdict =
        forward ? -NF_ACCEPT - 1 : -NF_DROP - 1;

    net->ipv4.iptable_filter =ipt_register_table(net, &packet_filter, repl);
                                                    /*注册 IPv4 数据包过滤表*/
    kfree(repl);
    return PTR_ERR_OR_ZERO(net->ipv4.iptable_filter);
}
```

●回调函数

下面看一下数据包过滤表在挂载点注册的回调函数 **iptable_filter_hook()**的实现，函数代码如下：

```
static unsigned int iptable_filter_hook(const struct nf_hook_ops *ops, struct sk_buff *skb,
                                        const struct nf_hook_state *state)
{
    const struct net *net;

    if (ops->hooknum == NF_INET_LOCAL_OUT &&(skb->len < sizeof(struct iphdr) ||
                                             ip_hdrlen(skb) < sizeof(struct iphdr)))
```

```
        return NF_ACCEPT;

    net = dev_net(state->in ? state->in : state->out);        /*网络命名空间*/
    return ipt_do_table(skb, ops->hooknum, state, net->ipv4.iptable_filter);      /*通用接口函数*/
}
```

iptable_filter_hook()函数中调用 ipt_do_table()函数，函数代码如下（/net/ipv4/netfilter/ip_tables.c）

```
unsigned int ipt_do_table(struct sk_buff *skb,
                          unsigned int hook,const struct nf_hook_state *state,struct xt_table *table)
/*table：此处指向 IPv4 中数据包过滤表 packet_filter*/
{
    static const char nulldevname[IFNAMSIZ] __attribute__((aligned(sizeof(long))));
    const struct iphdr *ip;
    unsigned int verdict = NF_DROP;
    const char *indev, *outdev;
    const void *table_base;
    struct ipt_entry *e, **jumpstack;
    unsigned int *stackptr, origptr, cpu;
    const struct xt_table_info *private;
    struct xt_action_param acpar;
    unsigned int addend;

    /*初始化*/
    ip = ip_hdr(skb);        /*指向数据包 IP 报头*/
    indev = state->in ? state->in->name : nulldevname;        /*输入网络设备*/
    outdev = state->out ? state->out->name : nulldevname;     /*输出网络设备*/

    acpar.fragoff = ntohs(ip->frag_off) & IP_OFFSET;
                                    /*0 表示是第一个分段数据包，非 0 表示是其它分段数据包*/
    acpar.thoff     = ip_hdrlen(skb);       /*IP 报头长度*/
    acpar.hotdrop = false;
    acpar.in          = state->in;
    acpar.out         = state->out;
    acpar.family    = NFPROTO_IPV4;        /*网络层协议类型*/
    acpar.hooknum = hook;        /*挂载点编号*/

    IP_NF_ASSERT(table->valid_hooks & (1 << hook));
    local_bh_disable();
    addend = xt_write_recseq_begin();
    private = table->private;          /*指向 xt_table_info 实例*/
    cpu     = smp_processor_id();       /*当前 CPU 核*/

    smp_read_barrier_depends();
    table_base = private->entries;        /*指向条目开始*/
    jumpstack   = (struct ipt_entry **)private->jumpstack[cpu];
    stackptr     = per_cpu_ptr(private->stackptr, cpu);
```

```
    origptr      = *stackptr;

    e = get_entry(table_base, private->hook_entry[hook]);      /*指向挂载点起始 ipt_entry 实例*/
    ...

    do {      /*遍历挂载点对应的 ipt_entry 实例列表*/
        const struct xt_entry_target *t;
        const struct xt_entry_match *ematch;
        struct xt_counters *counter;

        IP_NF_ASSERT(e);
        if (!ip_packet_match(ip, indev, outdev,&e->ip, acpar.fragoff)) {
                /*检查数据包是否与当前条目匹配，/net/ipv4/netfilter/ip_tables.c*/
no_match:
            e = ipt_next_entry(e);        /*下一个条目，ipt_entry 实例*/
            continue;
        }

        xt_ematch_foreach(ematch, e) {      /*遍历 ipt_entry 实例中的规则*/
            acpar.match        = ematch->u.kernel.match;
            acpar.matchinfo = ematch->data;
            if (!acpar.match->match(skb, &acpar))
                                                /*调用匹配函数，检查数据包与规则是否匹配*/
                goto no_match;
        }

        /*数据包与规则匹配*/
        counter = xt_get_this_cpu_counter(&e->counters);
        ADD_COUNTER(*counter, skb->len, 1);

        t = ipt_get_target(e);
                /*获取 xt_entry_target 实例，/include/uapi/linux/netfilter_ipv4/ip_tables.h*/
        IP_NF_ASSERT(t->u.kernel.target);

    #if IS_ENABLED(CONFIG_NETFILTER_XT_TARGET_TRACE)
        if (unlikely(skb->nf_trace))
            trace_packet(skb, hook, state->in, state->out,table->name, private, e);
    #endif

        /*是否是标准目标*/
        if (!t->u.kernel.target->target) {      /*不是标准目标，执行以下代码*/
            int v;

            v = ((struct xt_standard_target *)t)->verdict;
            if (v < 0) {
                if (v != XT_RETURN) {
```

```
                    verdict = (unsigned int)(-v) - 1;
                    break;
                }
                if (*stackptr <= origptr) {
                    e = get_entry(table_base,private->underflow[hook]);
                    ...
                } else {
                    e = jumpstack[--*stackptr];
                    ...
                    e = ipt_next_entry(e);
                }
                continue;
            }

            /*v 大于等于 0*/
            if (table_base + v != ipt_next_entry(e) &&!(e->ip.flags & IPT_F_GOTO)) {
                if (*stackptr >= private->stacksize) {
                    verdict = NF_DROP;
                    break;
                }
                jumpstack[(*stackptr)++] = e;
                ...
            }

            e = get_entry(table_base, v);
            continue;
        }     /*不是标准目标处理完成*/

        acpar.target     = t->u.kernel.target;
        acpar.targinfo = t->data;

        verdict = t->u.kernel.target->target(skb, &acpar);       /*调用目标回调函数*/
        ip = ip_hdr(skb);
        if (verdict == XT_CONTINUE)
            e = ipt_next_entry(e);
        else
            break;
    } while (!acpar.hotdrop);       /*遍历挂载点对应的 ipt_entry 实例列表结束*/

    ...
    *stackptr = origptr;
    xt_write_recseq_end(addend);
    local_bh_enable();

#ifdef DEBUG_ALLOW_ALL
    return NF_ACCEPT;
```

```
    #else
        if (acpar.hotdrop)
            return NF_DROP;
        else return verdict;
    #endif
}
```

ipt_do_table()函数遍历挂载点在条目列表中对应的 ipt_entry 实例，检查数据包是否与 ipt_entry 实例匹配（ip 成员），然后再检查是否与其中的规则匹配，若匹配，则调用目标中的 target()函数。

13.2 数据链路层协议实现

网络层协议发送数据包时，对其填充数据链路层报头后，调用 dev_queue_xmit(skb)函数将数据包传递给数据链路层。内核建立了发送和接收数据包的缓存队列，用于向数据链路层发送数据包或从数据链路层接收数据包。

网络层传递的数据包，先添加到发送缓存队列中，由发送数据包软中断分批将数据包发送到数据链路层，通过网络设备驱动最终发送到物理链路上。

网络设备从物理链路上接收到数据包后，将数据包提交到接收数据包缓存队列中，由接收数据包软中断分批将数据包传递给网络层。

13.2.1 概述

1 实现框架

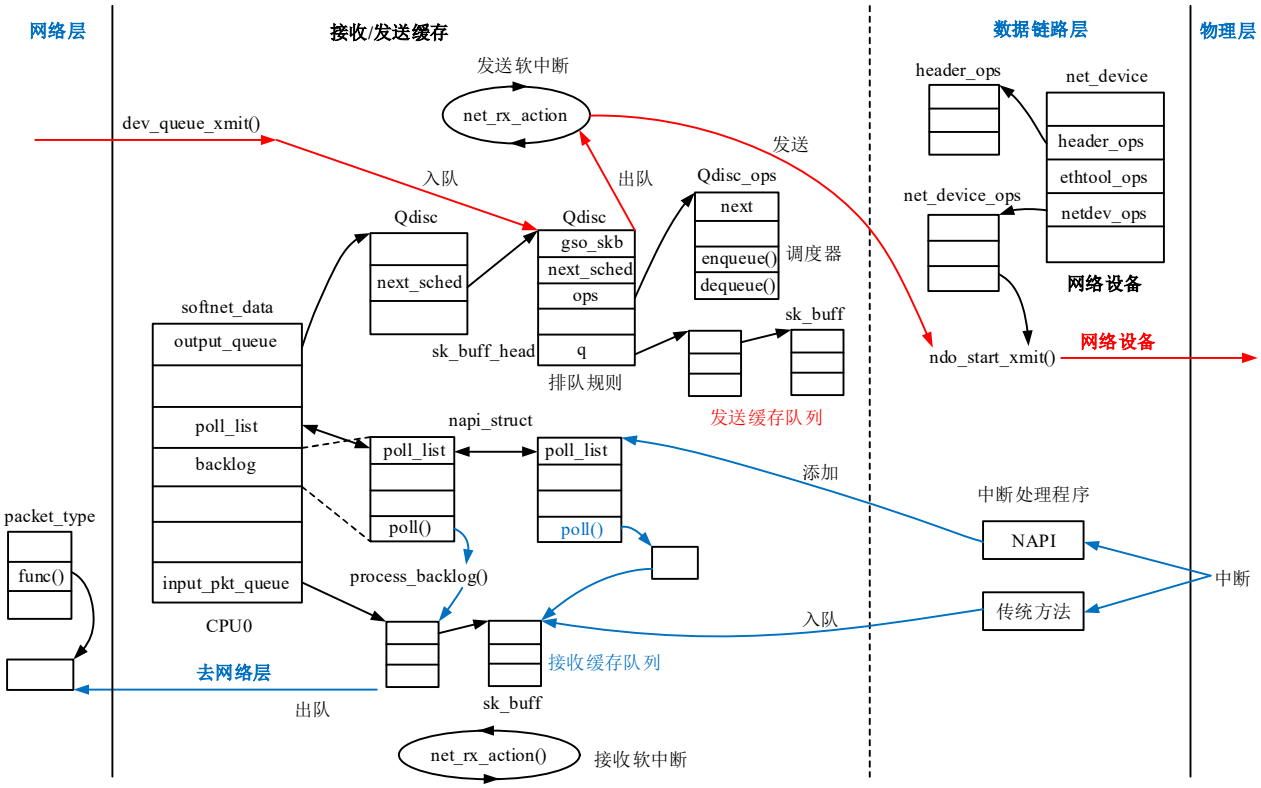
内核中数据链路层的实现框架如下图所示。内核在网络层与数据链路层之间建立了发送/接收数据包的缓存队列，分别由软中断将缓存队列中数据包发往数据链路层和网络层。

内核通过 softnet_data 结构体实例（percpu 变量）管理发送和接收数据包缓存队列，每个 CPU 核对应一个 softnet_data 实例。

发往某个网络设备的数据包由排队规则 Qdisc 结构体实例缓存，Qdisc 实例绑定到某个网络设备，只接收发往此设备的数据包，并添加到 softnet_data 实例中的双链表。Qdisc 实例还关联一个调度器，负责发送缓存队列中数据包的管理，包括数据包的入队/出队操作等，以实现发送数据包的调度。

网络层传递过来的数据包添加到输出网络设备关联 Qdisc 实例的发送队列。发送数据包软中断循环遍历 softnet_data 实例中 Qdisc 实例链表，从各实例发送队列中取出数据包，最终调用网络设备操作结构（网络设备驱动）中的发送函数将数据包传递给网络设备。网络设备通常由硬件将数据包中数据发送到物理链路。

softnet_data 实例中还包含一个接收数据包缓存队列。网络设备接收到数据包后，在驱动程序中构建数据包 sk_buff 实例，并将其添加到接收缓存队列。接收数据包软中断负责从接收缓存队列中取出数据包，传递到网络层。



数据链路层协议实现网络中相邻节点之间的数据传输。数据链路层协议主要由网络设备（驱动程序）实现。网络设备在驱动程序中由 net_device 结构体表示，net_device 实例关联网络设备操作 net_device_ops 结构体实例，数据链路层报头操作结构 header_ops 实例等。net_device_ops 实例实现特定于网络设备的操作，header_ops 实例实现特定于数据链路层协议的报头操作。

net_device_ops 结构体中的 ndo_start_xmit()函数指针成员，负责将数据包传递给网络设备，并由网络设备最终将其发送到物理链路上。

网络设备在接收数据包时，传统的方法是在每个数据包到来时，都产生一个中断，在中断处理程序中构建 sk_buff 实例，并将其提交到接收缓存队列。

但是，随着网络速度的提高，传统方法在每个数据包到达时都要产生中断，开销较大，不能适应高速的要求。因此，内核实现了被称为 NAPI（New API）的方法。NAPI 方法下，网络设备驱动程序需要定义 napi_struct 结构体实例，其中的 poll()函数负责接收数据包。softnet_data 实例中包含 napi_struct 实例双链表，称为轮询表。网络设备在第一个数据包到达时进入中断处理程序，在处理程序中关闭网络设备中断，然后将其 napi_struct 实例添加到轮询表，触发接收数据包软中断，中断返回（不开中断）。接收数据包软中断处理程序将不断轮询 napi_struct 实例链表，调用实例的 poll()函数接收数据包。直到没有数据包到达时，将 napi_struct 实例从轮询表中移出，开启网络设备中断。

表示网络设备的 net_device 实例，由驱动程序创建和注册，并由网络命名空间管理。net_device 实例关联的网络设备操作结构 net_device_ops 实例网络设备驱动程序实现。net_device 实例关联的 header_ops 实例表示数据链路层报头操作结构，由数据链路层协议实现，如以太网协议定义了其 header_ops 实例 eth_header_ops。数据链路层协议定义的数据发送接收等操作主要由硬件实现，驱动程只要把数据交给网络设备即可，接收数据时从网络设备中读取数据即可。

网络设备 net_device 实例归属于网络命名空间，由网络命名空间管理。网络命名空间 net 结构体中包含一个双链表，两个散列表（名称散列表和索引值散列表），用于管理网络设备。同一个网络设备在注册时同时加入到这三个链表中。

网络设备驱动程序的主要工作是选择正确的数据链路层协议，调用内核接口函数分配网络设备 net_device 实例；实现网络设备操作 net_device_ops 实例（主要是操作函数），并赋予 net_device 实例；实现并注册网络设备中断处理程序，在中断处理程序中将 napi_struct 实例添加到轮询表；最后调用内核接口

函数注册 net_device 实例。

2 初始化

下面来看一下数据链路层实现的初始化函数 net_dev_init()，定义如下（/net/core/dev.c）：

```
static int __init net_dev_init(void)
{
    int i, rc = -ENOMEM;

    BUG_ON(!dev_boot_phase);

    if (dev_proc_init())       /*完成在 procfs 中的初始化，创建文件等，/net/core/net-procfs.c*/
        goto out;
    if (netdev_kobject_init())     /*注册设备类 net_class 等，/net/core/net-sysfs.c*/
        goto out;

    INIT_LIST_HEAD(&ptype_all);        /*初始化 ptype_all 双链表*/
                            /*用于管理适用于所有网络层协议、未绑定到网络设备的 packet_type 实例*/
    for (i = 0; i < PTYPE_HASH_SIZE; i++)
        INIT_LIST_HEAD(&ptype_base[i]);
            /*初始化散列表，用于管理特定于网络层协议、未绑定到网络设备的 packet_type 实例*/

    INIT_LIST_HEAD(&offload_base);

    if (register_pernet_subsys(&netdev_net_ops))
                                    /*初始化网络命名空间中管理 net_device 实例的链表*/
        goto out;

    /*初始化各 CPU 核的 softnet_data 实例*/
    for_each_possible_cpu(i) {
        struct softnet_data *sd = &per_cpu(softnet_data, i);

        skb_queue_head_init(&sd->input_pkt_queue);     /*初始化数据包队列*/
        skb_queue_head_init(&sd->process_queue);
        INIT_LIST_HEAD(&sd->poll_list);       /*初始化 napi_struct 实例双链表*/
        sd->output_queue_tailp = &sd->output_queue;
    #ifdef CONFIG_RPS
        sd->csd.func = rps_trigger_softirq;
        sd->csd.info = sd;
        sd->cpu = i;
    #endif

        sd->backlog.poll = process_backlog;    /*赋值 backlog 实例的 poll()函数*/
        sd->backlog.weight = weight_p;       /*配额，默认值为 64*/
    }
```

```
    dev_boot_phase = 0;

    if (register_pernet_device(&loopback_net_ops))
                            /*初始化函数中注册环回设备，/drivers/net/loopback.c*/
        goto out;

    if (register_pernet_device(&default_device_ops))    /*/net/core/dev.c*/
        goto out;

    open_softirq(NET_TX_SOFTIRQ, net_tx_action);     /*注册发送数据包软中断*/
    open_softirq(NET_RX_SOFTIRQ, net_rx_action);     /*注册接收数据包软中断*/

    hotcpu_notifier(dev_cpu_callback, 0);     /*注册 CPU 热插拔通知，/net/core/dev.c*/
    dst_init();   /*向 netdev_chain 通知链注册通知 dst_dev_notifier 实例，/net/core/dst.c*/
    rc = 0;
out:
    return rc;
}
subsys_initcall(net_dev_init);      /*内核初始化阶段调用此函数*/
```

net_dev_init()函数主要完成以下工作：

（1）完成网络设备驱动在 procfs、sysfs 文件系统中的初始化，创建目录、文件等。

（2）初始化 ptype_all 双链表和 ptype_base[]散列表，用于管理未绑定到网络设备的 packet_type 实例。

（3）注册 netdev_net_ops 实例，其初始函数中将初始化网络命名空间中管理 net_device 实例的各链表。

（4）初始化各 CPU 核对应的 softnet_data 实例，结构体定义见 13.2.3 小节。

（5）注册 loopback_net_ops 实例，在其初始化函数中定义并注册了环回网络设备，设备名称为 lo，函数代码位于/drivers/net/loopback.c 文件内，可视此文件为一个简单网络设备驱动程序的模板。

（6）注册发送数据包的 NET_TX_SOFTIRQ 软中断，注册接收数据包的 NET_RX_SOFTIRQ 软中断，详见下文。

（7）向 CPU 热插拔通知链注册通知，通知处理函数为 dev_cpu_callback()，函数内主要完成 softnet_data 实例中接收队列的迁移等工作。

（8）向 netdev_chain 通知链注册通知 dst_dev_notifier 实例，见下文。

下面对 sysfs 中的初始化以及向 netdev_chain 通知链注册的 dst_dev_notifier 通知做简要介绍。

■ sysfs 初始化

内核通过网络设备 net_device、发送队列 netdev_queue，以及接收队列 netdev_rx_queue 结构体中内嵌的 kobject 结构体成员，将网络设备、发送队列和接收队列及其属性导出到 sysfs 文件系统。内核创建了 net_class 设备类，网络设备归于此类。发送队列、接收队列位于网络设备下，相关代码位于/net/core/net-sysfs.c 文件内。用户可通过对网络设备、队列属性的读写，获取/设置设备、队列属性。

初始化函数 netdev_kobject_init()用于注册 net_class 设备类等。向 sysfs 文件系统注册网络设备及队列的接口函数如下：

●int **netdev_register_kobject**(struct net_device *)：向 sysfs 注册网络设备，同时注册队列。

●int **net_rx_queue_update_kobjects**(struct net_device *, int old_num, int new_num)：更新接收队列。

●int **netdev_queue_update_kobjects**(struct net_device *net,int old_num, int new_num)：更新发送队列。

■注册 dst_dev_notifier 通知

函数 dst_init()向 netdev_chain 通知链注册通知 dst_dev_notifier 实例，定义如下（/net/core/dst.c）：

```
void __init dst_init(void)
{
    register_netdevice_notifier(&dst_dev_notifier);
}

static struct notifier_block dst_dev_notifier = {
    .notifier_call   = dst_dev_event,     /*通知回调函数*/
    .priority = -10, /* must be called after other network notifiers */
};
```

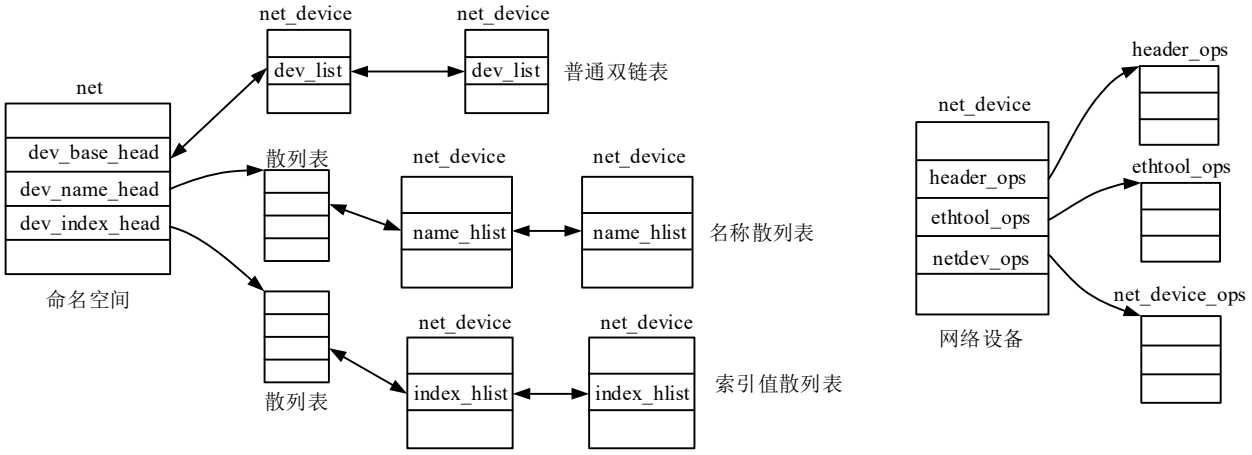
dst_dev_notifier 通知回调函数为 dst_dev_event()，函数根据网络设备事件调用 dst_entry 实例 dst_ops 结构体中的相应函数，完成网络设备事件在 dst_entry 实例中的处理工作，函数定义如下：

```
static int dst_dev_event(struct notifier_block *this, unsigned long event,void *ptr)
{
    struct net_device *dev = netdev_notifier_info_to_dev(ptr);
    struct dst_entry *dst, *last = NULL;

    switch (event) {
    case NETDEV_UNREGISTER_FINAL:
    case NETDEV_DOWN:       /*设备关闭*/
        mutex_lock(&dst_gc_mutex);
        for (dst = dst_busy_list; dst; dst = dst->next) {
            last = dst;
            dst_ifdown(dst, dev, event != NETDEV_DOWN);    /*调用 dst_ops 实例中 ifdown 函数*/
        }

        spin_lock_bh(&dst_garbage.lock);
        dst = dst_garbage.list;
        dst_garbage.list = NULL;
        spin_unlock_bh(&dst_garbage.lock);

        if (last)
            last->next = dst;
        else
            dst_busy_list = dst;
        for (; dst; dst = dst->next)
            dst_ifdown(dst, dev, event != NETDEV_DOWN);
        mutex_unlock(&dst_gc_mutex);
        break;
    }
    return NOTIFY_DONE;
}
```

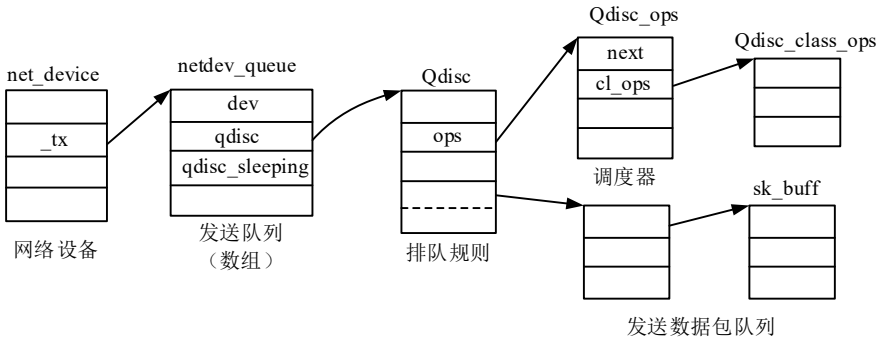
13.2.2 网络设备

网络设备在驱动程序中由 net_device 结构体表示，内核提供了创建、注册、管理 net_device 实例的接口函数供网络设备驱动程序调用。

net_device 实例由网络命名空间管理，如下图所示，网络命名空间 net 结构中包含一个双链表和两个散列表，用于管理 net_device 实例，同一个实例同时添加到双链表和两个散列表。net_device 实例需要关联 header_ops、net_device_ops、ethtool_ops 结构体实例等，其中 header_ops 实例由数据链路层协议实现，其它两个实例由网络设备驱动程序实现。



另外，网络设备还包含发送数据包队列（数组），如下图所示，用于缓存发送到本网络设备的数据包。



net_device 结构体中还包含其它许多网络设备相关的参数，下面先介绍相关数据结构的定义，然后介绍分配、注册网络设备接口函数的实现。

1 数据结构

网络设备驱动程序中主要的数据结构有 net_device、net_device_ops、ethtool_ops、header_ops 等，下面简要介绍其定义。

■net_device

net_device 结构体表示网络设备，它是一个庞大的数据结构（内核评论整个结构就是一个巨大的错误），结构体成员简列如下（/include/linux/netdevice.h）：

```
struct net_device {
    char            name[IFNAMSIZ];     /*网络设备名称，如 eth0、eth1 等*/
    struct hlist_node   name_hlist; /*将 net_device 实例插入到网络命名空间中按名称排列的散列表*/
    char            *ifalias;       /*接口的 SNMP 别名*/
    /*I/O 相关字段*/
    unsigned long       mem_end;    /*共享内存结束位置*/
```

```
    unsigned long       mem_start;      /*共享内存起始位置*/
    unsigned long       base_addr;      /*设备 IO 基地址*/
    int                 irq;            /*网络设备中断编号*/
    atomic_t            carrier_changes;
    unsigned long       state;      /*设备状态*/

    struct list_head    dev_list;       /*将实例添加到全局网络设备双链表*/
    struct list_head    napi_list;      /*链接 napi_struct 实例，用于轮询操作方式*/
    struct list_head    unreg_list;     /*将实例添加到已注销网络设备链表*/
    struct list_head    close_list;     /*关闭设备时使用的链表*/
    struct list_head    ptype_all;      /*链接绑定到此设备适用所有网络层协议的 packet_type 实例*/
    struct list_head    ptype_specific; /*链接绑定到此设备特定于网络层协议的 packet_type 实例*/
    struct {
        struct list_head upper;
        struct list_head lower;
    } adj_list;

    struct {
        struct list_head upper;
        struct list_head lower;
    } all_adj_list;

    netdev_features_t   features;           /*当前已启用的设备功能集*/
    netdev_features_t   hw_features;        /*可修改功能集*/
    netdev_features_t   wanted_features;        /*用户请求的功能集*/
    netdev_features_t   vlan_features;      /*其状态将被 VLAN 子系统继承的功能集*/
    netdev_features_t   hw_enc_features;    /*指出封装设备将继承哪些功能的掩码*/
    netdev_features_t   mpls_features;

    int    ifindex;     /*设备唯一标识符（索引值），在网络命名空间中分配*/
    int    group;
    struct net_device_stats    stats;
    atomic_long_t       rx_dropped;     /*接收路径中丢弃的数据包数量*/
    atomic_long_t       tx_dropped;     /*发送路径中丢弃的数据包数量*/
#ifdef CONFIG_WIRELESS_EXT
    const struct iw_handler_def * wireless_handlers;
    struct iw_public_data *   wireless_data;
#endif
    const struct net_device_ops *netdev_ops;    /*网络设备操作结构，见下文*/
    const struct ethtool_ops *ethtool_ops;      /*获取/设置各种网络参数，获取统计信息等操作结构*/
#ifdef CONFIG_NET_SWITCHDEV
    const struct switchdev_ops *switchdev_ops;
#endif
    const struct header_ops *header_ops;    /*指向数据链路层报头操作结构*/

    unsigned int        flags;      /*可从用户空间查看的网络设备的接口标志，见下文*/
```

```
    unsigned int        priv_flags;    /*用户空间不可见的接口标志*/

    unsigned short      gflags;
    unsigned short      padded;    /*alloc_netdev()分配实例时为满足对齐要求预留的字节*/
    unsigned char       operstate;
    unsigned char       link_mode;   /*操作状态映射策略*/

    unsigned char       if_port;
    unsigned char       dma;
    unsigned int        mtu;       /*网络接口的 MTU（最大传输单元）值，最大帧长度*/
    unsigned short      type;      /*网络接口的硬件类型，以太网接口为 ARPHRD_ETHER*/
    unsigned short      hard_header_len;      /*数据链路层报头长度*/
    unsigned short      needed_headroom;
    unsigned short      needed_tailroom;

    unsigned char       perm_addr[MAX_ADDR_LEN];      /*设备永久硬件地址（MAC 地址）*/
    unsigned char       addr_assign_type;    /*分配的硬件地址类型，如 NET_ADDR_PERM*/
    unsigned char       addr_len;       /*硬件地址长度，字节数*/
    unsigned short      neigh_priv_len;
    unsigned short      dev_id;
    unsigned short      dev_port;
    spinlock_t          addr_list_lock;
    unsigned char       name_assign_type;
    bool            uc_promisc;
    struct netdev_hw_addr_list uc;    /*单播 MAC 地址列表*/
    struct netdev_hw_addr_list mc;    /*组播 MAC 地址列表*/
    struct netdev_hw_addr_list dev_addrs;      /*设备硬件地址列表*/
#ifdef CONFIG_SYSFS
    struct kset     *queues_kset;
#endif
    unsigned int        promiscuity;   /*计数器，表示网络接口卡被命令在混杂模式下工作的次数*/
    unsigned int        allmulti;      /*计数器，可启用或禁用所有组播模式*/
    /*协议相关指针*/
    ...
    void                *atalk_ptr;
    struct in_device __rcu  *ip_ptr;   /*网络设备在 IPv4 网络层协议中的表示，指向 in_device 实例*/
    struct dn_dev __rcu     *dn_ptr;
    struct inet6_dev __rcu  *ip6_ptr;  /*指向网络设备在 IPv6 中的表示，inet6_dev 实例*/
    void            *ax25_ptr;
    struct wireless_dev     *ieee80211_ptr;
    struct wpan_dev         *ieee802154_ptr;
    ...
    unsigned long       last_rx;       /*收到最后一个数据包的时间*/
    unsigned char       *dev_addr;     /*网络接口的物理地址，随机分配的 MAC 地址*/

#ifdef CONFIG_SYSFS
```

```
    struct netdev_rx_queue  *_rx;       /*接收数据包缓存队列数组*/
    unsigned int        num_rx_queues;   /*注册网络设备时分配的接收队列数*/
    unsigned int        real_num_rx_queues;   /*活动中当前处于活动状态的接收队列数*/
#endif
    unsigned long       gro_flush_timeout;
    rx_handler_func_t __rcu  *rx_handler;     /*网络设备定义的将数据包传递给网络层的函数*/
    void __rcu          *rx_handler_data;

#ifdef CONFIG_NET_CLS_ACT
    struct tcf_proto __rcu    *ingress_cl_list;
#endif
    struct netdev_queue __rcu *ingress_queue;
#ifdef CONFIG_NETFILTER_INGRESS
    struct list_head    nf_hooks_ingress;
#endif
    unsigned char       broadcast[MAX_ADDR_LEN];     /*硬件广播地址*/
    ...
    struct hlist_node   index_hlist;     /*将实例添加到以索引值查找的散列表*/
    struct netdev_queue       *_tx ____cacheline_aligned_in_smp;       /*发送数据包缓存队列数组*/
    unsigned int        num_tx_queues;     /*队列数*/
    unsigned int        real_num_tx_queues;   /*活跃状态队列数*/
    struct Qdisc        *qdisc;   /*指向排队规则列表，实现对 Linux 内核的流量管理*/
    unsigned long       tx_queue_len;     /*每个队列可存储的最大数据包数*/
    spinlock_t          tx_global_lock;
    int             watchdog_timeo;

#ifdef CONFIG_XPS
    struct xps_dev_maps __rcu *xps_maps;
#endif
    unsigned long       trans_start;     /*最后一次传输时间*/
    struct timer_list   watchdog_timer;
    int __percpu        *pcpu_refcnt;     /*每个 CPU 的网络设备引用计数器*/
    struct list_head    todo_list;
    struct list_head    link_watch_list;
    ...
    bool dismantle;

    enum {
        RTNL_LINK_INITIALIZED,
        RTNL_LINK_INITIALIZING,
    } rtnl_link_state:16;
    void (*destructor)(struct net_device *dev);       /*在注销网络设备时调用的函数，析构函数*/

#ifdef CONFIG_NETPOLL
    struct netpoll_info __rcu  *npinfo;
#endif
```

```
    possible_net_t      nd_net;     /*指向所属网络命名空间的指针*/
    union {
        void            *ml_priv;
        struct pcpu_lstats __percpu         *lstats;
        struct pcpu_sw_netstats __percpu    *tstats;
        struct pcpu_dstats __percpu         *dstats;
        struct pcpu_vstats __percpu         *vstats;
    };

    struct garp_port __rcu  *garp_port;
    struct mrp_port __rcu   *mrp_port;
    struct device   dev;    /*device 实例，将网络设备添加到通用驱动模型*/
    const struct attribute_group *sysfs_groups[4];
    const struct attribute_group *sysfs_rx_queue_group;
    const struct rtnl_link_ops *rtnl_link_ops;    /*rtnetlink 消息类型处理实例（数组）指针*/
    #define GSO_MAX_SIZE        65536
    unsigned int        gso_max_size;
#define GSO_MAX_SEGS        65535
    u16             gso_max_segs;
    u16             gso_min_segs;
#ifdef CONFIG_DCB
    const struct dcbnl_rtnl_ops *dcbnl_ops;
#endif
    u8 num_tc;    /*网络设备中的流量类别数*/
    struct netdev_tc_txq tc_to_txq[TC_MAX_QUEUE];
    u8 prio_tc_map[TC_BITMASK + 1];

#if IS_ENABLED(CONFIG_FCOE)
    unsigned int        fcoe_ddp_xid;
#endif
#if IS_ENABLED(CONFIG_CGROUP_NET_PRIO)
    struct netprio_map __rcu *priomap;
#endif
    struct phy_device *phydev;      /*关联的物理设备，phy_device 表示物理层设备*/
    ...
};
```

net_device 结构体部分成员简介如下：

●**features**：netdev_features_t 结构体成员，这是一个 64 位无符号整数，每个比特位标记网络设备的一个设备功能，由驱动程序开发人员设置初始值。设备功能集如下（/include/linux/netdev_features.h）：

```
enum {
    NETIF_F_SG_BIT,             /* Scatter/gather IO. */
    NETIF_F_IP_CSUM_BIT,        /* Can checksum TCP/UDP over IPv4. */
    __UNUSED_NETIF_F_1,
    NETIF_F_HW_CSUM_BIT,        /* Can checksum all the packets. */
    NETIF_F_IPV6_CSUM_BIT,      /* Can checksum TCP/UDP over IPV6 */
    NETIF_F_HIGHDMA_BIT,        /* Can DMA to high memory. */
```

```
    NETIF_F_FRAGLIST_BIT,        /* Scatter/gather IO. 分散/聚集*/
    NETIF_F_HW_VLAN_CTAG_TX_BIT,    /* Transmit VLAN CTAG HW acceleration */
    NETIF_F_HW_VLAN_CTAG_RX_BIT,    /* Receive VLAN CTAG HW acceleration */
    NETIF_F_HW_VLAN_CTAG_FILTER_BIT,/* Receive filtering on VLAN CTAGs */
    NETIF_F_VLAN_CHALLENGED_BIT,    /* Device cannot handle VLAN packets */
    NETIF_F_GSO_BIT,          /* Enable software GSO. */
    NETIF_F_LLTX_BIT,         /* LockLess TX - deprecated. Please */
                              /* do not use LLTX in new drivers */
    NETIF_F_NETNS_LOCAL_BIT, /* Does not change network namespaces */
    NETIF_F_GRO_BIT,          /* Generic receive offload */
    NETIF_F_LRO_BIT,          /* large receive offload */

    NETIF_F_GSO_SHIFT,       /* keep the order of SKB_GSO_* bits */
    NETIF_F_TSO_BIT= NETIF_F_GSO_SHIFT,   /* ... TCPv4 segmentation */
    NETIF_F_UFO_BIT,          /* ... UDPv4 fragmentation */
    NETIF_F_GSO_ROBUST_BIT,          /* ... ->SKB_GSO_DODGY */
    NETIF_F_TSO_ECN_BIT,          /* ... TCP ECN support */
    NETIF_F_TSO6_BIT,         /* ... TCPv6 segmentation */
    NETIF_F_FSO_BIT,          /* ... FCoE segmentation */
    NETIF_F_GSO_GRE_BIT,          /* ... GRE with TSO */
    NETIF_F_GSO_GRE_CSUM_BIT,     /* ... GRE with csum with TSO */
    NETIF_F_GSO_IPIP_BIT,         /* ... IPIP tunnel with TSO */
    NETIF_F_GSO_SIT_BIT,          /* ... SIT tunnel with TSO */
    NETIF_F_GSO_UDP_TUNNEL_BIT,        /* ... UDP TUNNEL with TSO */
    NETIF_F_GSO_UDP_TUNNEL_CSUM_BIT, /* ... UDP TUNNEL with TSO & CSUM */
    NETIF_F_GSO_TUNNEL_REMCSUM_BIT,   /* ... TUNNEL with TSO & REMCSUM */
    NETIF_F_GSO_LAST =NETIF_F_GSO_TUNNEL_REMCSUM_BIT,
                                          /* last bit, see GSO_MASK */
    NETIF_F_FCOE_CRC_BIT,             /* FCoE CRC32 */
    NETIF_F_SCTP_CSUM_BIT,            /* SCTP checksum offload */
    NETIF_F_FCOE_MTU_BIT,             /* Supports max FCoE MTU, 2158 bytes*/
    NETIF_F_NTUPLE_BIT,           /* N-tuple filters supported */
    NETIF_F_RXHASH_BIT,           /* Receive hashing offload */
    NETIF_F_RXCSUM_BIT,           /* Receive checksumming offload */
    NETIF_F_NOCACHE_COPY_BIT,     /* Use no-cache copyfromuser */
    NETIF_F_LOOPBACK_BIT,             /* Enable loopback */
    NETIF_F_RXFCS_BIT,        /* Append FCS to skb pkt data */
    NETIF_F_RXALL_BIT,            /* Receive errored frames too */
    NETIF_F_HW_VLAN_STAG_TX_BIT,     /* Transmit VLAN STAG HW acceleration */
    NETIF_F_HW_VLAN_STAG_RX_BIT,     /* Receive VLAN STAG HW acceleration */
    NETIF_F_HW_VLAN_STAG_FILTER_BIT,/* Receive filtering on VLAN STAGs */
    NETIF_F_HW_L2FW_DOFFLOAD_BIT,  /* Allow L2 Forwarding in Hardware */
    NETIF_F_BUSY_POLL_BIT,            /* Busy poll */

    NETDEV_FEATURE_COUNT
};
```

GSO（Generic Segmentation Offload）：可以使大型数据包只经过网络栈一次，从而优化性能。要使用 GSO 必须使网络设备处于 Scatter/Gatter 模式。GSO 取代了只能用于 TCP/IPv4 的 TSO。

GRO（Generic Receive Offload）：在使用 GRO 时，收到的数据包将被合并。要使用 GRO，驱动程序必须在接收路径中调用 napi_gro_receive()方法。GRO 功能可改善网络性能，它取代了只能用于 TCP/IPv4 的 LRO。

●**flags**：可从用户空间查看的网络设备的接口标志，标志取值定义如下（/include/uapi/linux/if.h）：

```
enum net_device_flags {
    IFF_UP              = 1<<0,  /* sysfs，接口状态从关闭变为开启状态*/
    IFF_BROADCAST       = 1<<1,  /* volatile，接口处于混杂模式（接收数据包）时被设置*/
    IFF_DEBUG           = 1<<2,  /* sysfs */
    IFF_LOOPBACK        = 1<<3,  /* volatile */
    IFF_POINTOPOINT     = 1<<4,  /* volatile，设备为 PPP 设备时设置*/
    IFF_NOTRAILERS      = 1<<5,  /* sysfs，*/
    IFF_RUNNING         = 1<<6,  /* volatile */
    IFF_NOARP           = 1<<7,  /* sysfs，设备不使用 ARP 时设置*/
    IFF_PROMISC         = 1<<8,  /* sysfs */
    IFF_ALLMULTI        = 1<<9,  /* sysfs */
    IFF_MASTER          = 1<<10, /* volatile，设备为主设备时被设置*/
    IFF_SLAVE           = 1<<11, /* volatile，设备为从设备时被设置*/
    IFF_MULTICAST       = 1<<12, /* sysfs */
    IFF_PORTSEL         = 1<<13, /* sysfs */
    IFF_AUTOMEDIA       = 1<<14, /* sysfs */
    IFF_DYNAMIC         = 1<<15, /* sysfs */
    IFF_LOWER_UP        = 1<<16, /* volatile */
    IFF_DORMANT         = 1<<17, /* volatile */
    IFF_ECHO            = 1<<18, /* volatile */
};
```

■net_device_ops

net_device_ops 结构体成员全部是表示网络设备底层操作的函数指针，因此结构体实例由网络设备驱动程序实现，并赋予 net_device 实例。net_device_ops 结构体成员简列如下（/include/linux/netdevice.h）：

```
struct net_device_ops {
    int   (*ndo_init)(struct net_device *dev);     /*初始化函数，注册网络设备时调用*/
    void  (*ndo_uninit)(struct net_device *dev);    /*注销网络设备或注册失败时调用*/
    int   (*ndo_open)(struct net_device *dev);      /*网络设备从关闭变为开启时被调用*/
    int   (*ndo_stop)(struct net_device *dev);     /*网络设备状态变为关闭时被调用*/
    netdev_tx_t    (*ndo_start_xmit) (struct sk_buff *skb,struct net_device *dev);
                                        /*将数据包发送到物理链路，不能为 NULL*/
    u16  (*ndo_select_queue)(struct net_device *dev,struct sk_buff *skb,
                             void *accel_priv,select_queue_fallback_t fallback);
                                        /*在使用多个队列时，用于选择传输队列*/
    void  (*ndo_change_rx_flags)(struct net_device *dev,int flags);
    void  (*ndo_set_rx_mode)(struct net_device *dev);
    int   (*ndo_set_mac_address)(struct net_device *dev,void *addr);   /*设置网络设备物理地址*/
```

```
    int    (*ndo_validate_addr)(struct net_device *dev);       /*检查物理地址是否有效*/
    int    (*ndo_do_ioctl)(struct net_device *dev,struct ifreq *ifr, int cmd); /*处理网络设备私有 IO 命令*/
    int    (*ndo_set_config)(struct net_device *dev,struct ifmap *map);   /*设置网络设备总线接口参数*/
    int    (*ndo_change_mtu)(struct net_device *dev,int new_mtu);     /*负责处理 MTU 变更*/
    int    (*ndo_neigh_setup)(struct net_device *dev,struct neigh_parms *);
    void   (*ndo_tx_timeout) (struct net_device *dev);   /*在传输器空闲很长时间时被调用*/
    struct rtnl_link_stats64* (*ndo_get_stats64)(struct net_device *dev,struct rtnl_link_stats64 *storage);
    struct net_device_stats* (*ndo_get_stats)(struct net_device *dev);
    int    (*ndo_vlan_rx_add_vid)(struct net_device *dev,__be16 proto, u16 vid);
    int    (*ndo_vlan_rx_kill_vid)(struct net_device *dev,__be16 proto, u16 vid);
#ifdef CONFIG_NET_POLL_CONTROLLER
    void       (*ndo_poll_controller)(struct net_device *dev);
    int    (*ndo_netpoll_setup)(struct net_device *dev,struct netpoll_info *info);
    void   (*ndo_netpoll_cleanup)(struct net_device *dev);
#endif
#ifdef CONFIG_NET_RX_BUSY_POLL
    int    (*ndo_busy_poll)(struct napi_struct *dev);
#endif
    int    (*ndo_set_vf_mac)(struct net_device *dev,int queue, u8 *mac);
    int    (*ndo_set_vf_vlan)(struct net_device *dev,int queue, u16 vlan, u8 qos);
    int    (*ndo_set_vf_rate)(struct net_device *dev,int vf, int min_tx_rate,int max_tx_rate);
    int    (*ndo_set_vf_spoofchk)(struct net_device *dev,int vf, bool setting);
    int    (*ndo_get_vf_config)(struct net_device *dev,int vf,struct ifla_vf_info *ivf);
    int    (*ndo_set_vf_link_state)(struct net_device *dev,int vf, int link_state);
    int    (*ndo_get_vf_stats)(struct net_device *dev,int vf,struct ifla_vf_stats*vf_stats);
    int    (*ndo_set_vf_port)(struct net_device *dev,int vf,struct nlattr *port[]);
    int    (*ndo_get_vf_port)(struct net_device *dev,int vf, struct sk_buff *skb);
    int    (*ndo_set_vf_rss_query_en)(struct net_device *dev,int vf, bool setting);
    int    (*ndo_setup_tc)(struct net_device *dev, u8 tc);
#if IS_ENABLED(CONFIG_FCOE)
    ...
#endif

#if IS_ENABLED(CONFIG_LIBFCOE)
    ...
#endif

#ifdef CONFIG_RFS_ACCEL
    ...
#endif
    int    (*ndo_add_slave)(struct net_device *dev,struct net_device *slave_dev);
                                        /*用于将一个网络设备添加为另一个网络设备的从设备*/
    int    (*ndo_del_slave)(struct net_device *dev,struct net_device *slave_dev);   /*删除从设备*/
    netdev_features_t   (*ndo_fix_features)(struct net_device *dev,netdev_features_t features);
    int    (*ndo_set_features)(struct net_device *dev,netdev_features_t features); /*修改网络设备的功能*/
    int    (*ndo_neigh_construct)(struct neighbour *n);
```

```
    void   (*ndo_neigh_destroy)(struct neighbour *n);

    int    (*ndo_fdb_add)(struct ndmsg *ndm,struct nlattr *tb[],struct net_device *dev,
                          const unsigned char *addr,u16 vid,u16 flags);
    int    (*ndo_fdb_del)(struct ndmsg *ndm,struct nlattr *tb[],struct net_device *dev,
                          const unsigned char *addr,u16 vid);
    int    (*ndo_fdb_dump)(struct sk_buff *skb,struct netlink_callback *cb,struct net_device *dev,
                                                  struct net_device *filter_dev,int idx);
    int    (*ndo_bridge_setlink)(struct net_device *dev,struct nlmsghdr *nlh,u16 flags);
    int    (*ndo_bridge_getlink)(struct sk_buff *skb,u32 pid, u32 seq,struct net_device *dev,
                                                    u32 filter_mask,int nlflags);
    int    (*ndo_bridge_dellink)(struct net_device *dev,struct nlmsghdr *nlh,u16 flags);
    int    (*ndo_change_carrier)(struct net_device *dev,bool new_carrier);
    int    (*ndo_get_phys_port_id)(struct net_device *dev,struct netdev_phys_item_id *ppid);
    int    (*ndo_get_phys_port_name)(struct net_device *dev,char *name, size_t len);
    void   (*ndo_add_vxlan_port)(struct    net_device *dev,sa_family_t sa_family,__be16 port);
    void   (*ndo_del_vxlan_port)(struct    net_device *dev,sa_family_t sa_family,__be16 port);
    void* (*ndo_dfwd_add_station)(struct net_device *pdev,struct net_device *dev);
    void   (*ndo_dfwd_del_station)(struct net_device *pdev,void *priv);
    netdev_tx_t     (*ndo_dfwd_start_xmit) (struct sk_buff *skb,struct net_device *dev,void *priv);
    int          (*ndo_get_lock_subclass)(struct net_device *dev);
    netdev_features_t   (*ndo_features_check) (struct sk_buff *skb,struct net_device *dev,
                                                                netdev_features_t features);
    int    (*ndo_set_tx_maxrate)(struct net_device *dev,int queue_index,u32 maxrate);
    int    (*ndo_get_iflink)(const struct net_device *dev);
};
```

net_device_ops 结构体实例由网络设备驱动程序实现，其中 **ndo_start_xmit()**函数用于将数据包发送到物理链路。

■ethtool_ops

ethtool_ops 结构体包含许多函数指针，这些函数用于处理减负、获取和设置各种网络设置、获取统计信息、读取接收流方向散列表和局域网唤醒参数等。如果网络驱动程序没有初始化 net_device 实例中的 ethtool_ops 对象，内核网络核心将提供一个默认的空 ethtool_ops 实例 default_ethtool_ops。ethtool_ops 实例也需要由网络设备驱动程序实现。

ethtool_ops 结构体定义如下（/include/linux/ethtool.h）：

```
struct ethtool_ops {
    int    (*get_settings)(struct net_device *, struct ethtool_cmd *);
    int    (*set_settings)(struct net_device *, struct ethtool_cmd *);
    void (*get_drvinfo)(struct net_device *, struct ethtool_drvinfo *);
    int    (*get_regs_len)(struct net_device *);
    void (*get_regs)(struct net_device *, struct ethtool_regs *, void *);
    void (*get_wol)(struct net_device *, struct ethtool_wolinfo *);
    int    (*set_wol)(struct net_device *, struct ethtool_wolinfo *);
    u32  (*get_msglevel)(struct net_device *);
    void (*set_msglevel)(struct net_device *, u32);
```

```
    int    (*nway_reset)(struct net_device *);
    u32    (*get_link)(struct net_device *);
    int    (*get_eeprom_len)(struct net_device *);
    int    (*get_eeprom)(struct net_device *,struct ethtool_eeprom *, u8 *);
    int    (*set_eeprom)(struct net_device *,struct ethtool_eeprom *, u8 *);
    int    (*get_coalesce)(struct net_device *, struct ethtool_coalesce *);
    int    (*set_coalesce)(struct net_device *, struct ethtool_coalesce *);
    void (*get_ringparam)(struct net_device *,struct ethtool_ringparam *);
    int    (*set_ringparam)(struct net_device *,struct ethtool_ringparam *);
    void (*get_pauseparam)(struct net_device *,struct ethtool_pauseparam*);
    int    (*set_pauseparam)(struct net_device *,struct ethtool_pauseparam*);
    void (*self_test)(struct net_device *, struct ethtool_test *, u64 *);
    void (*get_strings)(struct net_device *, u32 stringset, u8 *);
    int    (*set_phys_id)(struct net_device *, enum ethtool_phys_id_state);
    void (*get_ethtool_stats)(struct net_device *,struct ethtool_stats *, u64 *);
    int    (*begin)(struct net_device *);
    void (*complete)(struct net_device *);
    u32    (*get_priv_flags)(struct net_device *);
    int    (*set_priv_flags)(struct net_device *, u32);
    int    (*get_sset_count)(struct net_device *, int);
    int    (*get_rxnfc)(struct net_device *,struct ethtool_rxnfc *, u32 *rule_locs);
    int    (*set_rxnfc)(struct net_device *, struct ethtool_rxnfc *);
    int    (*flash_device)(struct net_device *, struct ethtool_flash *);
    int    (*reset)(struct net_device *, u32 *);
    u32    (*get_rxfh_key_size)(struct net_device *);
    u32    (*get_rxfh_indir_size)(struct net_device *);
    int    (*get_rxfh)(struct net_device *, u32 *indir, u8 *key,u8 *hfunc);
    int    (*set_rxfh)(struct net_device *, const u32 *indir,const u8 *key, const u8 hfunc);
    void (*get_channels)(struct net_device *, struct ethtool_channels *);
    int    (*set_channels)(struct net_device *, struct ethtool_channels *);
    int    (*get_dump_flag)(struct net_device *, struct ethtool_dump *);
    int    (*get_dump_data)(struct net_device *,struct ethtool_dump *, void *);
    int    (*set_dump)(struct net_device *, struct ethtool_dump *);
    int    (*get_ts_info)(struct net_device *, struct ethtool_ts_info *);
    int    (*get_module_info)(struct net_device *,struct ethtool_modinfo *);
    int    (*get_module_eeprom)(struct net_device *,struct ethtool_eeprom *, u8 *);
    int    (*get_eee)(struct net_device *, struct ethtool_eee *);
    int    (*set_eee)(struct net_device *, struct ethtool_eee *);
    int    (*get_tunable)(struct net_device *,const struct ethtool_tunable *, void *);
    int    (*set_tunable)(struct net_device *,const struct ethtool_tunable *, const void *);
};
```

内核在/net/core/ethtool.c 文件内定义了部分成员函数的通用实现，以及部分成员函数的封装函数。

■header_ops

header_ops 结构体中包含数据链路层报头的操作函数指针，结构体定义如下（/include/linux/netdevice.h）：

```
struct header_ops {
    int   (*create) (struct sk_buff *skb, struct net_device *dev,
                unsigned short type, const void *daddr,const void *saddr, unsigned int len);
                                                    /*生成数据链路层报头，写入数据包报头中*/
    int   (*parse)(const struct sk_buff *skb, unsigned char *haddr);      /*获取源硬件地址*/
    int   (*cache)(const struct neighbour *neigh, struct hh_cache *hh, __be16 type);
                                                    /*将 L2 报头写入邻居实例*/
    void (*cache_update)(struct hh_cache *hh,const struct net_device *dev,const unsigned char *haddr);
                                                    /*更新 hh 指向的 L2 报头缓存*/
};
```

header_ops 结构体实例由数据链路层协议实现，对同类型的网卡都是相同的，不需要网络设备驱动程序实现，例如，以太网卡的默认实现在/net/ethernet/eth.c 文件内。

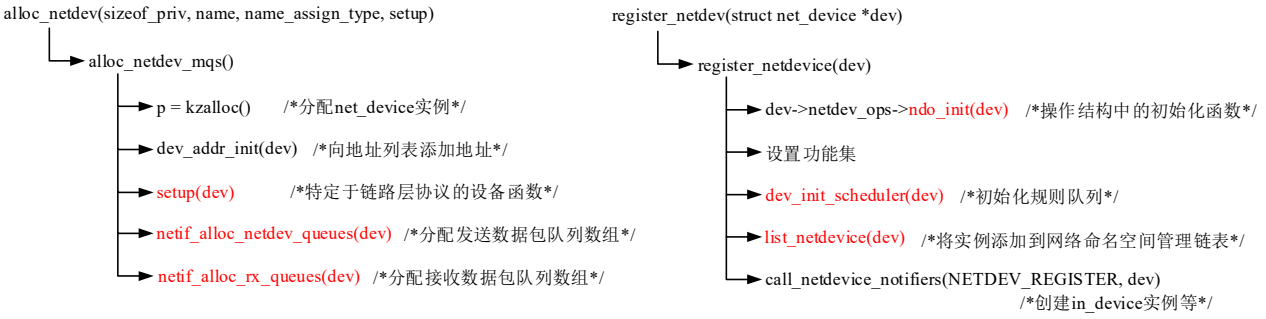
网络设备在发送数据包时，在邻居子系统中查找到邻居，且解析了物理地址后，将调用 header_ops 实例中的 cache()函数生成 L2 层报头缓存（下次发送数据包时直接使用），并调用 create()函数生成链路层报头写入数据包。

2 接口函数

网络设备驱动程序需要创建、填充和注册 net_device 实例。接口函数 alloc_netdev()用于创建 net_device 实例，并调用参数指定的特定于数据链路层协议的设置函数设置此实例，例如：以太网设备的设置函数为 ether_setup()。注册网络设备 net_device 实例的函数为 register_netdev()。

网络设备驱动程序的主要工作是：调用 **alloc_netdev()**函数创建 net_device 实例；对 net_device 实例各成员赋值，尤其是 net_device_ops、ethtool_ops 结构体成员等；定义并注册网络设备中断处理程序；最后调用 **register_netdev()**函数注册 net_device 实例。

下面主要介绍分配和注册 net_device 实例的函数，下图简要示意了这两个函数主要调用的函数，后面将详细介绍。



■分配 net_device 实例

通用的分配 net_device 实例的接口函数为 **alloc_netdev()**，定义如下（/include/linux/netdevice.h）：

```
#define alloc_netdev(sizeof_priv, name, name_assign_type, setup) \      /*设置函数 setup()*/
    alloc_netdev_mqs(sizeof_priv, name, name_assign_type, setup, 1, 1)
```

分配以太网网络设备 net_device 实例的接口函数如下（/include/linux/etherdevice.h）：

```
#define   alloc_etherdev(sizeof_priv)   alloc_etherdev_mq(sizeof_priv, 1)
#define   alloc_etherdev_mq(sizeof_priv, count)   alloc_etherdev_mqs(sizeof_priv, count, count)
```

alloc_etherdev_mqs()函数定义在/net/ethernet/eth.c 文件内：

```
struct net_device *alloc_etherdev_mqs(int sizeof_priv, unsigned int txqs,unsigned int rxqs)
/*sizeof_priv：私有数据大小，txqs：发送队列数量，此处为 1，rxqs：接收队列数量，此处为 1。*/
{
    return alloc_netdev_mqs(sizeof_priv, "eth%d", NET_NAME_UNKNOWN,
                                                    ether_setup, txqs, rxqs);
}
```

alloc_etherdev_mqs()函数内也是调用 alloc_netdev_mqs()函数分配 net_device 实例，网络设备文件名称为 ethX，设置 net_device 实例的函数为 **ether_setup()**，后面将详细介绍此函数。

alloc_netdev_mqs()函数定义在/net/core/dev.c 文件内，代码如下：

```
struct net_device *alloc_netdev_mqs(int sizeof_priv, const char *name,unsigned char name_assign_type,
        void (*setup)(struct net_device *),unsigned int txqs, unsigned int rxqs)
/*
*sizeof_priv：私有数据结构大小，附在 net_device 实例后，name：网络设备文件名称，
*name_assign_type：设备名称分配的类型，setup()：设置 net_device 实例的函数，
*txqs：发送队列数量，rxqs：接收队列数量。
*/
{
    struct net_device *dev;
    size_t alloc_size;
    struct net_device *p;

    BUG_ON(strlen(name) >= sizeof(dev->name));
    ...
    alloc_size = sizeof(struct net_device);     /*net_device 实例大小*/
    if (sizeof_priv) {     /*私有数据大小*/
        alloc_size = ALIGN(alloc_size, NETDEV_ALIGN);       /*32 字节对齐*/
        alloc_size += sizeof_priv;     /*分配空间大小，net_device 加私有数据*/
    }
    alloc_size += NETDEV_ALIGN - 1;     /*确保 32 字节对齐*/

    p = kzalloc(alloc_size, GFP_KERNEL | __GFP_NOWARN | __GFP_REPEAT);
                                                        /*分配 net_device 实例加私有数据*/
    if (!p)
        p = vzalloc(alloc_size);
    if (!p)
        return NULL;

    dev = PTR_ALIGN(p, NETDEV_ALIGN);     /*指向 net_device 实例*/
    dev->padded = (char *)dev - (char *)p;       /*指向私有数据结构*/

    dev->pcpu_refcnt = alloc_percpu(int);     /*分配 percpu 变量*/
    ...
    if (dev_addr_init(dev))       /*初始化网络设备物理地址列表，/net/core/dev_addr_lists.c*/
        goto free_pcpu;
```

```
    dev_mc_init(dev);       /*初始化组播地址列表，/net/core/dev_addr_lists.c*/
    dev_uc_init(dev);       /*初始化单播地址列表，/net/core/dev_addr_lists.c*/

    dev_net_set(dev, &init_net);     /*网络命名空间设为初始网络命名空间*/

    dev->gso_max_size = GSO_MAX_SIZE;       /*64KB*/
    dev->gso_max_segs = GSO_MAX_SEGS;     /*64KB*/
    dev->gso_min_segs = 0;
    /*初始化双链表成员*/
    INIT_LIST_HEAD(&dev->napi_list);
    INIT_LIST_HEAD(&dev->unreg_list);
    INIT_LIST_HEAD(&dev->close_list);
    INIT_LIST_HEAD(&dev->link_watch_list);
    INIT_LIST_HEAD(&dev->adj_list.upper);
    INIT_LIST_HEAD(&dev->adj_list.lower);
    INIT_LIST_HEAD(&dev->all_adj_list.upper);
    INIT_LIST_HEAD(&dev->all_adj_list.lower);
    INIT_LIST_HEAD(&dev->ptype_all);
    INIT_LIST_HEAD(&dev->ptype_specific);
    dev->priv_flags = IFF_XMIT_DST_RELEASE | IFF_XMIT_DST_RELEASE_PERM;
    setup(dev);       /*调用数据链路层协议的设置函数*/

    dev->num_tx_queues = txqs;       /*发送队列数量，通常为 1*/
    dev->real_num_tx_queues = txqs;
    if (netif_alloc_netdev_queues(dev))     /*分配发送缓存队列数组，/net/core/dev.c*/
        goto free_all;

#ifdef CONFIG_SYSFS
    dev->num_rx_queues = rxqs;     /*接收队列数量，通常为 1*/
    dev->real_num_rx_queues = rxqs;
    if (netif_alloc_rx_queues(dev))   /*分配接收缓存队列数组，/net/core/dev.c*/
        goto free_all;
#endif

    strcpy(dev->name, name);     /*复制网络设备名称*/
    dev->name_assign_type = name_assign_type;
    dev->group = INIT_NETDEV_GROUP;
    if (!dev->ethtool_ops)   /*如果 ethtool_ops 为 NULL，赋予默认实例 default_ethtool_ops（全空）*/
        dev->ethtool_ops = &default_ethtool_ops;

    nf_hook_ingress_init(dev);   /*初始化 dev->nf_hooks_ingress 成员，需配置 NETFILTER_INGRESS*/
                    /*没有配置 NETFILTER_INGRESS 为空，/include/linux/netfilter_ingress.h*/
    return dev;     /*返回 net_device 实例指针*/
    ...
}
```

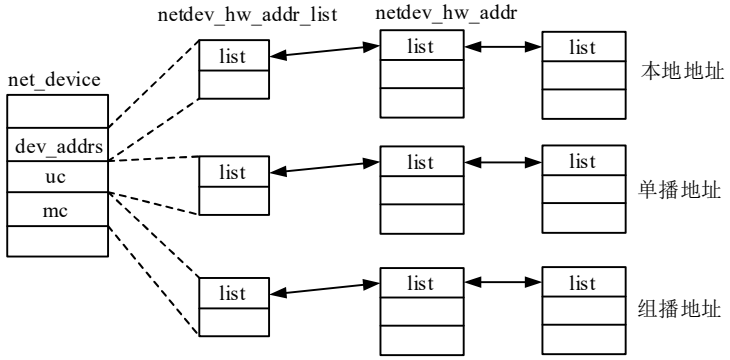
alloc_netdev()函数主要工作是分配 net_device 结构体实例（含私数据结构），调用参数 setup()函数设置实例，初始化各地址列表，创建发送、接收缓存队列等。下面对部分工作做详细介绍。

●初始化地址列表

net_device 结构体中有三个成员记录了网络设备、单播、组播的物理地址，如下所示：

```
net_device{
    ...
    struct netdev_hw_addr_list    uc;     /*单播 MAC 地址列表*/
    struct netdev_hw_addr_list    mc;     /*组播 MAC 地址列表*/
    struct netdev_hw_addr_list    dev_addrs;      /*设备硬件地址列表*/
    ...
}
```

以上三个成员都是 netdev_hw_addr_list 结构体实例，结构体中包含一个双链表，管理 netdev_hw_addr 结构体实例，如下图所示：



netdev_hw_addr_list 和 netdev_hw_addr 结构体定义如下（/include/linux/netdevice.h）：

```
struct netdev_hw_addr_list {        /*/include/linux/netdevice.h*/
    struct list_head    list;       /*双链表成员*/
    int                 count;      /*双链表中成员数量*/
};

struct netdev_hw_addr {             /*物理地址表示*/
    struct list_head    list;       /*双链表成员*/
    unsigned char       addr[MAX_ADDR_LEN];     /*地址值*/
    unsigned char       type;    /*地址类型，本地地址、单播地址、组播地址等*/
#define   NETDEV_HW_ADDR_T_LAN          1     /*本地地址，地址类型定义*/
#define   NETDEV_HW_ADDR_T_SAN          2
#define   NETDEV_HW_ADDR_T_SLAVE        3
#define   NETDEV_HW_ADDR_T_UNICAST      4     /*单播地址*/
#define   NETDEV_HW_ADDR_T_MULTICAST    5     /*组播地址*/
    bool                global_use;
    int                 sync_cnt;
    int                 refcount;    /*引用计数*/
    int                 synced;
    struct rcu_head             rcu_head;
};
```

地址列表相关的操作函数在/net/core/dev_addr_lists.c 文件内实现。在注册 net_device 实例的函数中调用**dev_addr_init(dev)**函数为 dev_addrs 链表创建并添加地址值为 0，类型为 NETDEV_HW_ADDR_T_LAN 的 netdev_hw_addr 实例。

dev_uc_init(dev)和 dev_mc_init(dev)函数初始化 net_devcie 实例 uc、mc 成员，所做的工作就是初始化其中的 list 和 count 成员，而并没有创建和添加 netdev_hw_addr 实例。

在/net/core/dev_addr_lists.c 文件内定义了向网络设备添加/删除各类型地址的接口函数，例如：

◎int **dev_addr_add**(struct net_device *dev, const unsigned char *addr,unsigned char addr_type)：向网络设备 dev_addrs 链表创建并添加 netdev_hw_addr 实例，如果地址实例已存在则增加其引用计数。另外此函数内还将调用执行 netdev_chain 通知链通知（NETDEV_CHANGEADDR 事件）。

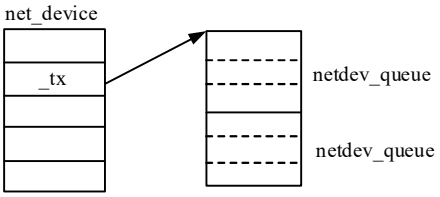
◎int **dev_addr_del**(struct net_device *dev, const unsigned char *addr,unsigned char addr_type)：释放 dev_addrs 链表指定地址值、地址类型的 netdev_hw_addr 实例，并调用执行 netdev_chain 通知链通知（NETDEV_CHANGEADDR 事件）。

◎int **dev_uc_add**(struct net_device *dev, const unsigned char *addr)：向 dev->uc 添加单播地址。

◎int **dev_mc_add**(struct net_device *dev, const unsigned char *addr)：向 dev->mc 添加组播地址。

●分配发送缓存队列数组

alloc_netdev()函数中调用 **netif_alloc_netdev_queues(dev)**函数为网络设备分配发送数据包缓存队列数组，队列由 netdev_queue 结构体表示，如下图所示：



在 alloc_netdev()函数中，实际只为网络设备分配了一个发送缓存队列。

netdev_queue 结构体定义如下（/include/linux/netdevice.h）：

```
struct netdev_queue {
    struct net_device    *dev;      /*指向网络设备*/
    struct Qdisc __rcu   *qdisc;    /*指向排队规则，见下文*/
    struct Qdisc         *qdisc_sleeping;
#ifdef CONFIG_SYSFS
    struct kobject       kobj;     /*跟踪队列的 kobject 实例*/
#endif
#if defined(CONFIG_XPS) && defined(CONFIG_NUMA)
    int          numa_node;
#endif

    spinlock_t      _xmit_lock    ____cacheline_aligned_in_smp;
    int             xmit_lock_owner;
    unsigned long       trans_start;
    unsigned long       trans_timeout;
    unsigned long       state;    /*状态*/

#ifdef CONFIG_BQL
    struct dql      dql;
#endif
```

```
    unsigned long       tx_maxrate;
} ____cacheline_aligned_in_smp;
```

netdev_queue 结构体部分成员简介如下：

●**state**：状态，取值定义如下（/include/linux/netdevice.h）：

```
enum netdev_queue_state_t {
    __QUEUE_STATE_DRV_XOFF,      /*用于驱动程序关闭发送缓存队列*/
    __QUEUE_STATE_STACK_XOFF,    /*用于 stack 关闭发送缓存队列*/
    __QUEUE_STATE_FROZEN,
};
/*定义标记位*/
#define   QUEUE_STATE_DRV_XOFF      (1 << __QUEUE_STATE_DRV_XOFF)
#define   QUEUE_STATE_STACK_XOFF      (1 << __QUEUE_STATE_STACK_XOFF)
#define   QUEUE_STATE_FROZEN    (1 << __QUEUE_STATE_FROZEN)

#define   QUEUE_STATE_ANY_XOFF
                (QUEUE_STATE_DRV_XOFF | QUEUE_STATE_STACK_XOFF)
#define   QUEUE_STATE_ANY_XOFF_OR_FROZEN
                            (QUEUE_STATE_ANY_XOFF | QUEUE_STATE_FROZEN)
#define   QUEUE_STATE_DRV_XOFF_OR_FROZEN
                            (QUEUE_STATE_DRV_XOFF | QUEUE_STATE_FROZEN)
```

netif_alloc_netdev_queues(dev)函数为网络设备分配发送数据包缓存队列，定义如下（/net/core/dev.c）：

```
static int netif_alloc_netdev_queues(struct net_device *dev)
{
    unsigned int count = dev->num_tx_queues;    /*发送缓存队列数量，通常为 1*/
    struct netdev_queue *tx;        /*指向缓存队列 netdev_queue 实例*/
    size_t sz = count * sizeof(*tx);

    if (count < 1 || count > 0xffff)
        return -EINVAL;

    tx = kzalloc(sz, GFP_KERNEL | __GFP_NOWARN | __GFP_REPEAT);   /*分配队列数组*/
    if (!tx) {
        tx = vzalloc(sz);
        if (!tx)
            return -ENOMEM;
    }
    dev->_tx = tx;    /*指向队列数组*/

    netdev_for_each_tx_queue(dev, netdev_init_one_queue, NULL);
                    /*对每个队列实例调用 netdev_init_one_queue()初始化函数*/
    spin_lock_init(&dev->tx_global_lock);
    return 0;
}
```

以上函数内为网络设备分配 netdev_queue 实例数组，并赋予_tx 成员，对每个队列实例调用初始化函

数 netdev_init_one_queue()（/net/core/dev.c），函数内只是对队列实例进行简单的初始化，源代码请读者自行阅读。

●分配接收缓存队列数组

网络设备接收缓存队列由 netdev_rx_queue 结构体表示（需选择 SYSFS 配置选项），结构定义如下：

```
struct netdev_rx_queue {      /*/include/linux/netdevice.h*/
  #ifdef CONFIG_RPS
      struct rps_map __rcu          *rps_map;
      struct rps_dev_flow_table __rcu  *rps_flow_table;
  #endif
      struct kobject            kobj;     /*跟踪队列的 kobject 实例*/
      struct net_device         *dev;     /*指向网络设备*/
} ____cacheline_aligned_in_smp;
```

在 alloc_netdev()函数内调用 netif_alloc_rx_queues()函数为网络设备分配接收缓存队列，函数定义如下：

```
static int netif_alloc_rx_queues(struct net_device *dev)      /*/net/core/dev.c*/
{
    unsigned int i, count = dev->num_rx_queues;     /*接收队列数量，通常为 1*/
    struct netdev_rx_queue *rx;
    size_t sz = count * sizeof(*rx);

    BUG_ON(count < 1);

    rx = kzalloc(sz, GFP_KERNEL | __GFP_NOWARN | __GFP_REPEAT);     /*分配队列数组*/
    if (!rx) {
        rx = vzalloc(sz);
        if (!rx)
            return -ENOMEM;
    }
    dev->_rx = rx;     /*队列指针赋予_rx 成员*/
    for (i = 0; i < count; i++)
        rx[i].dev = dev;       /*指向网络设备*/
    return 0;
}
```

●设置 net_device 实例

alloc_netdev()函数将调用参数 setup()传递的函数设置 net_device 实例。setup()函数通常由数据链路层协议实现。对于以太网设备，setup()参数为 **ether_setup()**函数指针。

ether_setup()函数定义如下（/net/ethernet/eth.c）：

```
void ether_setup(struct net_device *dev)
{
    dev->header_ops    = &eth_header_ops;       /*数据链路层报头操作结构实例*/
    dev->type          = ARPHRD_ETHER;     /*网络设备类型*/
    dev->hard_header_len   = ETH_HLEN;     /*以太网报头长度（14 字节）*/
    dev->mtu       = ETH_DATA_LEN;         /*MTU 值，1500 字节*/
```

```
    dev->addr_len       = ETH_ALEN;        /*物理地址长度，6 字节*/
    dev->tx_queue_len = 1000;          /*发送队列长度*/
    dev->flags          = IFF_BROADCAST|IFF_MULTICAST;    /*网络设备标志*/
    dev->priv_flags     |= IFF_TX_SKB_SHARING;

    eth_broadcast_addr(dev->broadcast);      /*设置组播地址，FF.FF.FF.FF.FF.FF*/
}
```

以太网报头操作结构实例为 eth_header_ops，用于生成以太网报头等操作，详见 13.2.5 小节。

■注册 net_device 实例

网络设备驱动程序在调用 alloc_netdev()函数创建 net_device 实例后，还需要对实例进行设置，最后调用 **register_netdev**(struct net_device *dev)函数注册 net_device 实例，函数定义在/net/core/dev.c 文件内，代码如下：

```
int register_netdev(struct net_device *dev)
{
    int err;
    rtnl_lock();
    err = register_netdevice(dev);    /*持有锁情况下注册实例，/net/core/dev.c*/
    rtnl_unlock();
    return err;
}
```

register_netdevice()函数定义如下：

```
int register_netdevice(struct net_device *dev)
{
    int ret;
    struct net *net = dev_net(dev);        /*网络命名空间*/
    ...
    might_sleep();
    ...
    spin_lock_init(&dev->addr_list_lock);
    netdev_set_addr_lockdep_class(dev);

    ret = dev_get_valid_name(net, dev, dev->name); /*将有效设备名称写入 dev->name，/net/core/dev.c*/
    ...
    if (dev->netdev_ops->ndo_init) {
        ret = dev->netdev_ops->ndo_init(dev);    /*调用网络设备操作结构中的初始化函数*/
        ...
    }

    if (((dev->hw_features | dev->features) &NETIF_F_HW_VLAN_CTAG_FILTER) &&
        (!dev->netdev_ops->ndo_vlan_rx_add_vid ||!dev->netdev_ops->ndo_vlan_rx_kill_vid)) {
        netdev_WARN(dev, "Buggy VLAN acceleration in driver!\n");
        ret = -EINVAL;
        goto err_uninit;
```

```
}

ret = -EBUSY;
if (!dev->ifindex)
    dev->ifindex = dev_new_index(net);
                        /*为网络设备分配索引值，每个网络设备赋予一个编号*/
else if (__dev_get_by_index(net, dev->ifindex))
                        /*如果已指定了索引值，查找是否存在相同值的设备*/
    goto err_uninit;

dev->hw_features |= NETIF_F_SOFT_FEATURES;    /*设置功能集*/
dev->features |= NETIF_F_SOFT_FEATURES;
dev->wanted_features = dev->features & dev->hw_features;

if (!(dev->flags & IFF_LOOPBACK)) {    /*不是环回设备*/
    dev->hw_features |= NETIF_F_NOCACHE_COPY;
}
dev->vlan_features |= NETIF_F_HIGHDMA;
dev->hw_enc_features |= NETIF_F_SG;
dev->mpls_features |= NETIF_F_SG;

ret = call_netdevice_notifiers(NETDEV_POST_INIT, dev);   /*执行 netdev_chain 通知链中通知*/
ret = notifier_to_errno(ret);
...
ret = netdev_register_kobject(dev);    /*注册 net_device 实例内嵌 device 实例，/net/core/net-sysfs.c*/
...
dev->reg_state = NETREG_REGISTERED;    /*注册状态*/

__netdev_update_features(dev);    /*更新设备功能集，/net/core/dev.c*/

set_bit(__LINK_STATE_PRESENT, &dev->state);        /*设置状态标志位*/

linkwatch_init_dev(dev);    /*/net/core/link_watch.c*/

dev_init_scheduler(dev);    /*初始化排队规则，见 13.2.3 小节，/net/sched/sch_generic.c*/
dev_hold(dev);    /*增加引用计数，/include/linux/netdevice.h*/
list_netdevice(dev);    /*将 net_device 实例添加到管理结构，见下文*/
add_device_randomness(dev->dev_addr, dev->addr_len);

if (dev->addr_assign_type == NET_ADDR_PERM)
    memcpy(dev->perm_addr, dev->dev_addr, dev->addr_len);

/* Notify protocols, that a new device appeared. */
ret = call_netdevice_notifiers(NETDEV_REGISTER, dev);
                        /*再次执行 netdev_chain 通知链中通知，如创建 in_device 实例等。*/
ret = notifier_to_errno(ret);
```

```
    if (ret) {
        rollback_registered(dev);
        dev->reg_state = NETREG_UNREGISTERED;
    }

    if (!dev->rtnl_link_ops ||dev->rtnl_link_state == RTNL_LINK_INITIALIZED)
        rtmsg_ifinfo(RTM_NEWLINK, dev, ~0U, GFP_KERNEL);
                                        /*向用户套接字发送新网络接口消息*/
out:
    return ret;
    ...
}
```

注册网络设备完成的主要工作有：设置网络设备名称；赋予设备编号；执行 netdev_chain 通知链中通知；设置功能集标志；初始化排队规则；将 net_device 实例添加到管理结构等。

下面分别介绍将实例添加到管理结构和执行 netdev_chain 通知链中通知所做的工作。

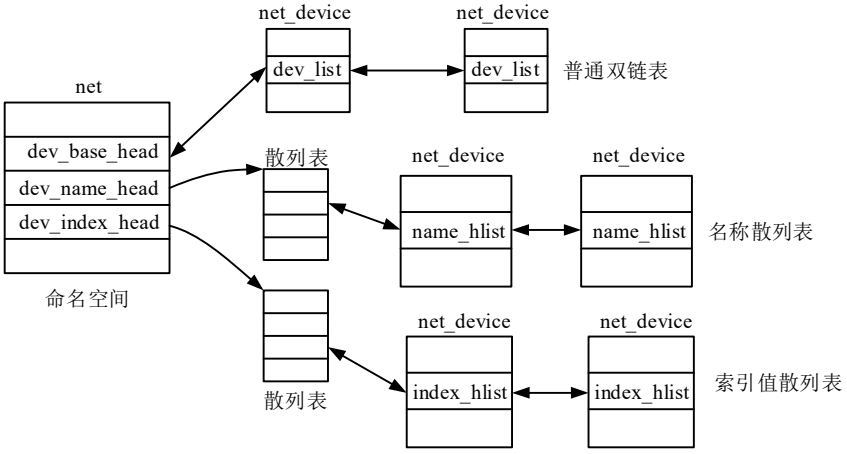
●网络设备管理

网络设备归属于网络命名空间，由网络命名空间管理。网络命名空间 net 结构体中建立一个双链表，两个散列表（名称散列表和索引值散列表），用于管理网络设备。同一个网络设备在注册时同时加入到这三个链表中。

网络命名空间 net 结构体中与管理 net_device 实例相关的成员如下所示：

```
struct net{
    ...
    struct list_head      dev_base_head;    /*双链表成员*/
    struct hlist_head     *dev_name_head;   /*指向名称散列表*/
    struct hlist_head     *dev_index_head;  /*指向索引值散列表*/
    ...
}
```

网络命名空间管理 net_device 实例的结构如下图所示：



内核在网络设备初始化函数 net_dev_init()中注册了 pernet_operations 结构体实例 netdev_net_ops，在此实例的初始化函数中将为网络命名空间初始化管理 net_device 实例的双链表、散列表。

```
static int __init net_dev_init(void)
{
    ...
    if (register_pernet_subsys(&netdev_net_ops))     /*注册 netdev_net_ops 实例*/
```

```
            goto out;
        ...
}
subsys_initcall(net_dev_init);
```

netdev_net_ops 实例定义如下：

```
static struct pernet_operations __net_initdata netdev_net_ops = {
    .init = netdev_init,     /*创建并初始化管理 net_device 实例的双链表、散列表，/net/core/dev.c*/
    .exit = netdev_exit,
};
```

netdev_init()函数将初始化网络命名空间中 dev_base_head 双链表成员，为指针成员 dev_name_head 和 dev_index_head 分配并初始化散列表实例，函数源代码请读者自行阅读。

在注册 net_device 实例的 register_netdevice()函数中调用 **list_netdevice(dev)**函数将 net_device 实例添加到网络命名空间网络设备管理结构中，函数定义如下（/net/core/dev.c）：

```
static void list_netdevice(struct net_device *dev)
{
    struct net *net = dev_net(dev);       /*网络命名空间*/

    ASSERT_RTNL();

    write_lock_bh(&dev_base_lock);
    list_add_tail_rcu(&dev->dev_list, &net->dev_base_head);     /*插入网络命名空间双链表*/
    hlist_add_head_rcu(&dev->name_hlist, dev_name_hash(net, dev->name));
                                                                /*添加到按名称排列的散列表*/
    hlist_add_head_rcu(&dev->index_hlist,dev_index_hash(net, dev->ifindex));
                                                                /*添加到按索引值排列的散列表*/
    write_unlock_bh(&dev_base_lock);
    dev_base_seq_inc(net);
}
```

以下接口函数用于在指定网络命名空间中查找 net_device 实例：

●struct net_device ***dev_get_by_index**(struct net *net, int ifindex)：在按索引值管理的散列表中，按索引值 ifindex 查找网络设备。

●struct net_device ***dev_get_by_name**(struct net *net, const char *name)：在按名称管理的散列表中，按网络设备名称查找网络设备。

●创建 in_device 实例

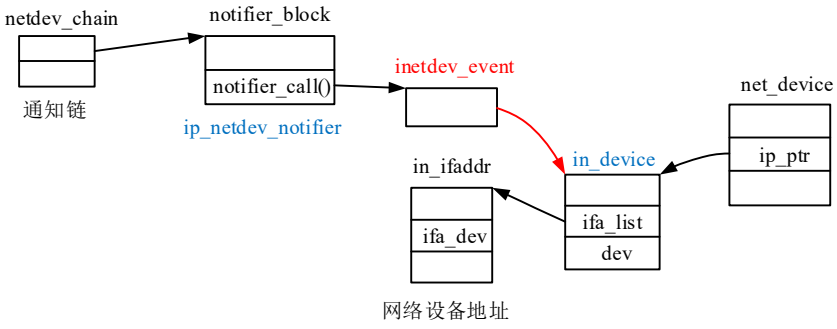
在 13.1.2 小节中介绍的本地地址管理时，在 devinet_init()函数中向 netdev_chain 通知链中注册了通知实例 **ip_netdev_notifier**，通知实例如下（/net/ipv4/devinet.c）：

```
static struct notifier_block ip_netdev_notifier = {
    .notifier_call = inetdev_event,
};
```

在注册 net_device 实例的 register_netdevice()函数中将执行 netdev_chain 通知链中注册的通知，设备事件为 NETDEV_REGISTER。其中 ip_netdev_notifier 通知的回调函数定义如下（/net/ipv4/devinet.c）：

```
static int inetdev_event(struct notifier_block *this, unsigned long event,void *ptr)
/*this：指向通知实例，event：NETDEV_REGISTER，ptr：net_device 实例指针*/
{
    struct net_device *dev = netdev_notifier_info_to_dev(ptr);
    struct in_device *in_dev = __in_dev_get_rtnl(dev);      /*在网络层表示网络设备的 in_device 实例*/

    ASSERT_RTNL();

    if (!in_dev) {      /*in_device 实例为 NULL，则创建 in_device 实例*/
        if (event == NETDEV_REGISTER) {
            in_dev = inetdev_init(dev);     /*创建 in_device 实例，/net/ipv4/devinet.c*/
            if (IS_ERR(in_dev))
                return notifier_from_errno(PTR_ERR(in_dev));
            if (dev->flags & IFF_LOOPBACK) {
                IN_DEV_CONF_SET(in_dev, NOXFRM, 1);
                IN_DEV_CONF_SET(in_dev, NOPOLICY, 1);
            }
        } else if (event == NETDEV_CHANGEMTU) {
            ...
        }
        goto out;
    }
    ...
out:
    return NOTIFY_DONE;
}
```

inetdev_init(dev)函数用于为 net_device 实例创建在 IPv4 中表示网络设备的 in_device 实例，创建结果如下图所示：



用户通过 netlink 套接字为网络设备（接口）添加本地地址时，将为 in_device 实例创建并关联 in_ifaddr 实例。由 IP 地址查找网络设备的接口函数为 ip_dev_find()，定义如下（/include/linux/inetdevice.h）：

```
static inline struct net_device *ip_dev_find(struct net *net, __be32 addr)
/*net：网络命名空间，addr：IP 地址*/
{
    return __ip_dev_find(net, addr, true);
}
```

__ip_dev_find()函数定义如下（/net/ipv4/devinet.c）：

```
struct net_device *__ip_dev_find(struct net *net, __be32 addr, bool devref)
{
    u32 hash = inet_addr_hash(net, addr); /*计算 inet_addr_lst 散列表散列值，用于查找 in_ifaddr 实例*/
    struct net_device *result = NULL;
    struct in_ifaddr *ifa;

    rcu_read_lock();
    hlist_for_each_entry_rcu(ifa, &inet_addr_lst[hash], hash) {     /*遍历散列链表*/
        if (ifa->ifa_local == addr) {       /*地址相同的 in_ifaddr 实例*/
            struct net_device *dev = ifa->ifa_dev->dev;       /*net_device 实例*/
            if (!net_eq(dev_net(dev), net))
                continue;
            result = dev;
            break;
        }
    }
    if (!result) {     /*未找到 in_ifaddr 实例，则将 addr 作为本地 IP 地址执行路由选择查找*/
        struct flowi4 fl4 = { .daddr = addr };       /*设为目的地址*/
        struct fib_result res = { 0 };
        struct fib_table *local;
        local = fib_get_table(net, RT_TABLE_LOCAL);
        if (local &&!fib_table_lookup(local, &fl4, &res, FIB_LOOKUP_NOREF) &&
                                                        res.type == RTN_LOCAL)
            result = FIB_RES_DEV(res);       /*网络设备*/
    }
    if (result && devref)
        dev_hold(result);
    rcu_read_unlock();
    return result;
}
```

3 用户接口

用户进程可通过 NETLINK_ROUTE 套接字的消息类型，设置网络设备参数，消息类型如下：

内核在头文件定义了 NETLINK_ROUTE 套接字的消息类型，如下所示：

```
enum {                      /*/include/uapi/linux/rtnetlink.h*/
    RTM_BASE   = 16,            /*消息类型基数，小于 16 的消息类型为控制消息保留*/

    RTM_NEWLINK  = 16,     /*设置网络接口（网络设备）参数，如 MAC 地址等*/
    RTM_DELLINK,
    RTM_GETLINK,
    RTM_SETLINK,
    ...
}
```

消息类型处理函数在/net/core/rtnetlink.c 文件内定义，用户进程通过 ifinfomsg 结构体传递网络接口参

数（/include/uapi/linux/rtnetlink.h）。消息类型处理函数在 rtnetlink_init()函数内注册，各处理函数源代码请读者自行阅读。

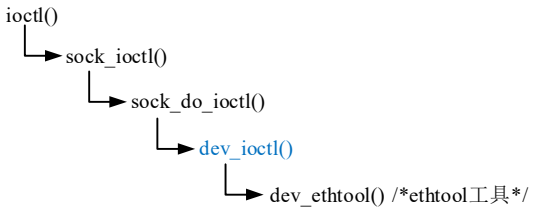
用户进程还可通过 ioctl()系统调用，ifreq 结构体实现内核与用户之间的交互网络设备信息。
ifreq 结构体定义如下（/include/uapi/linux/if.h）：

```
struct ifreq {
  #define IFHWADDRLEN  6
    union
    {
        char    ifrn_name[IFNAMSIZ];        /*网络设备文件名称*/
    } ifr_ifrn;

    union {       /*联合体*/
        struct   sockaddr ifru_addr;
        struct   sockaddr ifru_dstaddr;
        struct   sockaddr ifru_broadaddr;
        struct   sockaddr ifru_netmask;
        struct    sockaddr ifru_hwaddr;
        short    ifru_flags;
        int      ifru_ivalue;
        int      ifru_mtu;
        struct    ifmap ifru_map;
        char     ifru_slave[IFNAMSIZ]; /* Just fits the size */
        char     ifru_newname[IFNAMSIZ];
        void __user *     ifru_data;       /*ethtool 工具接口，用户数据开头是命令，后面是参数*/
        struct   if_settings ifru_settings;
  } ifr_ifru;
};
```

ioctl()系统调用网络设备配置相关命令定义在/include/uapi/linux/sockios.h 头文件，例如：

```
#define   SIOCGIFNAME     0x8910        /* get iface name*/
#define   SIOCSIFLINK     0x8911        /* set iface channel */
#define   SIOCGIFCONF     0x8912        /* get iface list*/
#define   SIOCGIFFLAGS    0x8913        /* get flags*/
#define   SIOCSIFFLAGS    0x8914        /* set flags*/
#define   SIOCGIFADDR     0x8915        /* get PA address*/
...
```

ioctl()系统调用函数调用关系简列如下图所示：



dev_ioctl()函数用于处理网络设备相关的命令，函数代码简列如下（/net/core/dev_ioctl.c）：

```
int dev_ioctl(struct net *net, unsigned int cmd, void __user *arg)
{
    struct ifreq    ifr;        /*ifreq 结构体实例*/
    int ret;
    char *colon;

    if (cmd == SIOCGIFCONF) {
        rtnl_lock();
        ret = dev_ifconf(net, (char __user *) arg);
        rtnl_unlock();
        return ret;
    }
    if (cmd == SIOCGIFNAME)
        return dev_ifname(net, (struct ifreq __user *)arg);

    if (copy_from_user(&ifr, arg, sizeof(struct ifreq)))     /*复制用户数据至 ifreq 实例*/
        return -EFAULT;

    ifr.ifr_name[IFNAMSIZ-1] = 0;

    colon = strchr(ifr.ifr_name, ':');
    if (colon)
        *colon = 0;
    switch (cmd) {      /*根据命令调用不同的处理函数*/
    case SIOCGIFFLAGS:
    case SIOCGIFMETRIC:
    case SIOCGIFMTU:
    case SIOCGIFHWADDR:
    case SIOCGIFSLAVE:
    case SIOCGIFMAP:
    case SIOCGIFINDEX:
    case SIOCGIFTXQLEN:
        dev_load(net, ifr.ifr_name);
        rcu_read_lock();
        ret = dev_ifsioc_locked(net, &ifr, cmd);
        rcu_read_unlock();
        ...
        return ret;

    case SIOCETHTOOL:       /*用于 ethtool 工具*/
        dev_load(net, ifr.ifr_name);
        rtnl_lock();
        ret = dev_ethtool(net, &ifr);       /*/net/core/ethtool.c*/
        rtnl_unlock();
        ...
        return ret;
```

```
        ...     /*处理其它命令*/
        }
    }
```

dev_ioctl()函数根据命令值调用不同的处理函数。这里需要说明的是 SIOCETHTOOL 命令，它通过 ifreq 结构体中 ifru_data 用户数据指针成员传递参数。SIOCETHTOOL 命令下包含许多子命令，用户数据第一个成员是子命令，随后是参数。子命令及参数形式定义在/include/uapi/linux/ethtool.h 头文件。

dev_ethtool()函数内根据 SIOCETHTOOL 子命令调用不同的处理函数，这些函数中常调用 ethtool_ops 实例中的函数完成操作，函数源代码请读者自行阅读。

13.2.3 发送数据包

由 13.1.3 小节介绍的数据包发送流程可知，IPv4 中发送函数 ip_finish_output2()将在邻居表中查找邻居实例（如果没有查找到则创建），如果邻居中缓存了 L2 层报头，则将报头写入数据包，调用 dev_queue_xmit()函数将数据包发往数据链路层。如果邻居中没有缓存 L2 层报头，则发送 ARP 请求，接收邻居发回的 ARP 应答，解析邻居物理地址后，生成 L2 层报头写入邻居实例缓存，并生成 L2 层报头写入数据包，最后调用 dev_queue_xmit()函数发送数据包。总之，dev_queue_xmit()函数是网络层向数据链路层发送数据包的接口函数。

1 概述

如下图所示，网络设备中包含发送数据包队列，由 netdev_queue 结构体表示。通常网络设备具有一个队列，也可以包含多个队列，本节只考虑单个队列的情形。netdev_queue 结构体关联排队规则 Qdisc 实例，此实例添加到网络设备 qdisc 链表。

Qdisc 实例中包含发送数据包缓存队列，Qdisc 实例关联网络调度器 Qdisc_ops 实例，主要负责数据包的入队、出队等操作。内核定义并注册了多个调度器实例，这些实例由全局单链表 qdisc_base 管理。

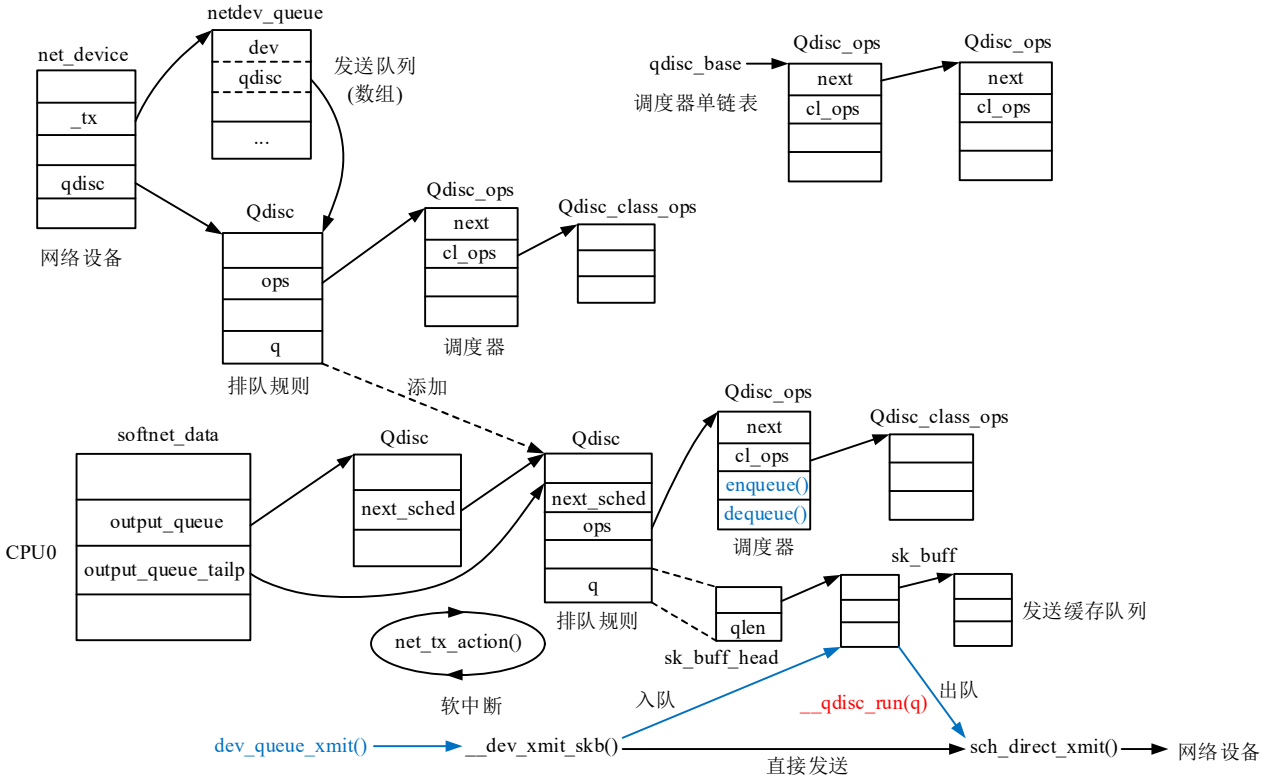
在注册网络设备时，网络设备关联一个空的排队规则及调度器，在激活（打开）网络设备时将为其创建新的排队规则，并关联默认的 pfifo_fast_ops 调度器实例。用户进程可通过用户接口设置系统默认的调度器实例以及网络设备的调度器实例等。

网络层调用 dev_queue_xmit()接口函数发送数据包至数据链路层。dev_queue_xmit()函数判断网络设备的排队规则（调度器）是否定义了入队函数，如果没有定义则直接将数据包立即发送到网络设备。

如果定义了入队函数，再判断排队规则中队列是否为空且排队规则允许被旁路，是则直接发送数据包至网络设备。否则，调用入队函数将数据包添加到排队规则中的缓存队列，然后调用**__qdisc_run(q)**函数从排队规则队列中取出数据包，逐个发送到网络设备。如果__qdisc_run(q)函数中发送的数据包数量达到了配额或需要重调度，则将 Qdisc 实例添加到 softnet_data 结构体实例中的链表，并触发 NET_TX_SOFTIRQ 软中断，__qdisc_run(q)函数返回。

softnet_data 结构体实例是一个 percpu 变量，每个 CPU 核对应一个实例，用于缓存发送到网络设备和从网络设备接收的数据包，它相当于一个数据包的中转站。

在 NET_TX_SOFTIRQ 软中断处理函数 net_tx_action()中，将遍历 softnet_data 实例中 Qdisc 实例链表，对每个实例调用 qdisc_run()函数（调用__qdisc_run(q)函数），此函数从排队规则队列中取出数据包，发送到网络设备。当发送的数据包达到配额或需要执行重调度时，qdisc_run()函数将 Qdisc 实例重新添加到 softnet_data 实例中 Qdisc 实例链表，并触发软中断，依此循环。



2 网络调度器

当网络设备有许多数据包要发送时，需要确定哪些优先级比较高，需要先发送，哪些可延迟发送，哪些可以丢弃。这项工作由网络调度器完成，网络设备发送队列关联到一个排队规则，排队规则最终通过关联的网络调度器来实现数据包的优先级管理。这类似于进程管理和块设备驱动中的调度器。排队规则还是内核实现流量控制的基础。

排队规则由 Qdisc 结构体表示，网络调度器由 Qdisc_ops 结构体表示。内核定义并注册了许多网络调度器实例。在注册/开启网络设备时，为发送队列创建排队规则，并关联默认的网络调度器，用户可修改网络设备使用的调度器。

相关代码在/net/sched/目录下文件内实现。

■数据结构

网络设备 net_device 实例可关联多个排队规则，由 qdisc 成员指向 Qdisc 实例链表。排队规则包含发送数据包队列，表示网络调度器的 Qdisc_ops 结构体实例指针等成员。调度器 Qdisc_ops 结构体中主要是函数指针成员，包含数据包的入队、出队、选择等操作，以及对排队规则 Qdisc 实例的操作等。

●softnet_data

softnet_data 结构体用于在网络层和数据链路层之间管理发送和接收数据包的一个数据结构，可以理解成数据包的中转站。softnet_data 结构体定义如下（/include/linux/netdevice.h）：

```
struct softnet_data {
    struct list_head      poll_list;      /*napi_struct 实例链表*/
    struct sk_buff_head   process_queue;

    unsigned int          processed;
```

```
    unsigned int        time_squeeze;
    unsigned int        cpu_collision;
    unsigned int        received_rps;
#ifdef CONFIG_RPS
    struct softnet_data  *rps_ipi_list;
#endif
#ifdef CONFIG_NET_FLOW_LIMIT
    struct sd_flow_limit __rcu *flow_limit;
#endif
    struct Qdisc        *output_queue;   /*指向排队规则链表*/
    struct Qdisc        **output_queue_tailp;      /*指链表中末尾 Qdisc 实例 next_sched 成员*/
    struct sk_buff      *completion_queue;     /*完成队列*/

#ifdef CONFIG_RPS
    struct call_single_data   csd ____cacheline_aligned_in_smp;
    struct softnet_data  *rps_ipi_next;
    unsigned int        cpu;
    unsigned int        input_queue_head;
    unsigned int        input_queue_tail;
#endif
    unsigned int        dropped;
    struct sk_buff_head     input_pkt_queue;      /*接收数据包缓存队列*/
    struct napi_struct backlog;      /*其 poll()函数，将接收缓存队列中数据包发往网络层*/
};
```

softnet_data 结构体主要成员简介如下：

●**poll_list**：双链表，管理网络设备添加的 napi_struct 实例，用于接收数据包。

●**output_queue**：指向排队规则 Qdisc 实例，用于发送数据包，详见下文。

●**completion_queue**：完成数据包队列。

●**input_pkt_queue**：接收数据包缓存队列。

●**backlog**：napi_struct 结构体实例，将添加到 poll_list 双链表，其 poll()函数（**process_backlog()**）将 input_pkt_queue 队列中数据包传递到网络层。

内核在/include/linux/netdevice.h 头文件中定义了 softnet_data 结构体实例（percpu 变量）：

```
DECLARE_PER_CPU_ALIGNED(struct softnet_data, softnet_data);   /*每个 CPU 对应一个实例*/
```

在初始化函数 net_dev_init()中将初始化 softnet_data 实例，见 13.2.1 小节。

●Qdisc

排队规则 Qdisc 结构体定义如下（/include/net/sch_generic.h）：

```
struct Qdisc {
    int         (*enqueue)(struct sk_buff *skb, struct Qdisc *dev);     /*数据包入队函数*/
    struct sk_buff *     (*dequeue)(struct Qdisc *dev);        /*数据包出队函数*/
    unsigned int        flags;   /*标志*/
    ...      /*标志位定义，见下文*/
    u32             limit;
    const struct Qdisc_ops *ops;   /*指向网络调度器，排队规则关联一个网络调度器，见下文*/
```

```
    struct qdisc_size_table __rcu *stab;
    struct list_head    list;    /*在多队列网络设备中，将实例链入网络设备 net_device->qdisc 链表*/
    u32             handle;    /*句柄，排队规则唯一的标识，网络设备初始排队规则此值为 0*/
    u32             parent;    /*父节点句柄值*/
    int             (*reshape_fail)(struct sk_buff *skb,struct Qdisc *q);

    void            *u32_node;

    struct Qdisc        *__parent;      /*排队规则父节点*/
    struct netdev_queue     *dev_queue;    /*指向发送数据包缓存队列*/

    struct gnet_stats_rate_est64    rate_est;      /*统计信息*/
    struct gnet_stats_basic_cpu __percpu *cpu_bstats;
    struct gnet_stats_queue  __percpu *cpu_qstats;

    struct Qdisc        *next_sched;   /*下一个实例，将实例链入 softnet_data->output_queue 链表*/
    struct sk_buff      *gso_skb;      /*数据包暂存队列*/

    unsigned long       state;      /*调度状态*/
    struct sk_buff_head  q;         /*缓存数据包队列头*/
    struct gnet_stats_basic_packed bstats;
    unsigned int    __state;      /*排队规则状态*/
    struct gnet_stats_queue  qstats;
    struct rcu_head         rcu_head;
    int             padded;
    atomic_t        refcnt;

    spinlock_t          busylock ____cacheline_aligned_in_smp;
};
```

Qdisc 结构体部分成员简介如下：

●**enqueue()**：数据包入队函数指针，来源于 Qdisc_ops 实例。

●**dequeue()**：数据包出队函数指针，来源于 Qdisc_ops 实例。

●**flags**：标志，取值在结构体内定义，如下所示：

```
#define   TCQ_F_BUILTIN          1
#define   TCQ_F_INGRESS          2
#define   TCQ_F_CAN_BYPASS       4    /*可以旁路本排队规则*/
#define   TCQ_F_MQROOT           8
#define   TCQ_F_ONETXQUEUE       0x10      /*网络设备只有一个发送缓存队列*/
#define   TCQ_F_WARN_NONWC       (1 << 16)
#define   TCQ_F_CPUSTATS         0x20    /* run using percpu statistics */
```

●**ops**：指向 Qdisc_ops 实例，表示网络调度器，详见下文。

●**state**：调度状态，这是一个位图，每个位的语义由以下枚举类型定义：

```
enum qdisc_state_t {
    __QDISC_STATE_SCHED,   /*bit0，表示队列规则是否接收软中断的调度*/
    __QDISC_STATE_DEACTIVATED,    /*bit1，不活跃*/
```

```
    __QDISC_STATE_THROTTLED,        /**/
};
```

●**__state**：排队规则状态，由枚举类型表示：

```
enum qdisc___state_t {
    __QDISC___STATE_RUNNING = 1,    /*启用*/
};
```

●**stab**：指向 qdisc_size_table 结构体，结构体定义如下：

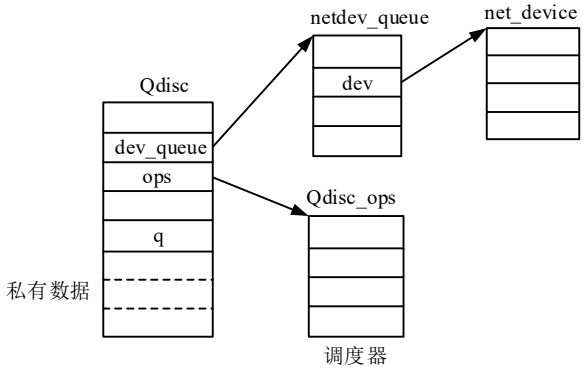
```
struct qdisc_size_table {
    struct rcu_head     rcu;
    struct list_head    list;
    struct tc_sizespec  szopts;
    int                 refcnt;
    u16                 data[];
};
```

●**list**：双链表成员，在多队列网络设备中，将 Qdisc 实例链入网络设备 net_device->qdisc 链表。

●**next_sched**：指向下一个 Qdisc 实例，用于将实例链入 **softnet_data->output_queue** 链表。

●**q**：发送缓存数据包队列头。

函数 struct Qdisc ***qdisc_alloc**(struct netdev_queue *dev_queue,const struct Qdisc_ops *ops)用于分配排队规则 Qdisc 实例，参数需要指定发送缓存队列 netdev_queue 实例和关联的网络调度器 Qdisc_ops 实例。函数执行结果如下图所示（/net/sched/sch_generic.c）：



在 qdisc_alloc()函数中会将 Qdisc_ops 实例中的 enqueue()和 dequeue()函数指针成员赋予 Qdisc 实例。

●**Qdisc_ops**

Qdisc_ops 结构体表示网络调度器，定义如下（/include/net/sch_generic.h）：

```
struct Qdisc_ops {
    struct Qdisc_ops    *next;    /*指向下一个实例，将实例链入全局 qdisc_base 双链表*/
    const struct Qdisc_class_ops  *cl_ops;      /*排队规则类操作结构*/
    char                id[IFNAMSIZ];       /*名称，用于标识调度器，用户套接字查找调度器*/
    int                 priv_size;              /*私有数据，分配 Qdisc 实例时附在其后的空间大小*/

    int                 (*enqueue)(struct sk_buff *, struct Qdisc *);       /*入队函数，赋予 Qdisc 实例*/
    struct sk_buff *    (*dequeue)(struct Qdisc *);       /*出队函数，赋予 Qdisc 实例*/
    struct sk_buff *    (*peek)(struct Qdisc *);     /*选取数据包*/
    unsigned int        (*drop)(struct Qdisc *);
```

```
    int             (*init)(struct Qdisc *, struct nlattr *arg);      /*初始化函数，初始化 Qdisc 实例*/
    void            (*reset)(struct Qdisc *);      /*复位函数，复位 Qdisc 实例*/
    void            (*destroy)(struct Qdisc *);    /*释放函数，释放 Qdisc 实例资源*/
    int             (*change)(struct Qdisc *, struct nlattr *arg);    /*更新函数，修改 Qdisc 实例参数*/
    void            (*attach)(struct Qdisc *);
                        /*多队列情况下，用于绑定排队规则，将实例添加到 net_device->qdisc 链表等*/
    int             (*dump)(struct Qdisc *, struct sk_buff *);      /*输出函数*/
    int             (*dump_stats)(struct Qdisc *, struct gnet_dump *);

    struct module       *owner;
};
```

Qdisc_ops 结构体中主要包含数据包的出入队列函数，Qdisc 实例的操作函数指针成员等，其中 cl_ops 成员指向 Qdisc_class_ops 结构体，结构体定义如下：

```
struct Qdisc_class_ops {
    /* Child qdisc manipulation */
    struct netdev_queue *    (*select_queue)(struct Qdisc *, struct tcmsg *);
    int         (*graft)(struct Qdisc *, unsigned long cl,struct Qdisc *, struct Qdisc **);
    struct Qdisc *      (*leaf)(struct Qdisc *, unsigned long cl);
    void        (*qlen_notify)(struct Qdisc *, unsigned long);

    /* Class manipulation routines */
    unsigned long       (*get)(struct Qdisc *, u32 classid);
    void        (*put)(struct Qdisc *, unsigned long);
    int         (*change)(struct Qdisc *, u32, u32,struct nlattr **, unsigned long *);
    int         (*delete)(struct Qdisc *, unsigned long);
    void        (*walk)(struct Qdisc *, struct qdisc_walker * arg);

    /* Filter manipulation */
    struct tcf_proto __rcu ** (*tcf_chain)(struct Qdisc *, unsigned long);
    unsigned long       (*bind_tcf)(struct Qdisc *, unsigned long,u32 classid);
    void        (*unbind_tcf)(struct Qdisc *, unsigned long);

    /* rtnetlink specific */
    int         (*dump)(struct Qdisc *, unsigned long,struct sk_buff *skb, struct tcmsg*);
    int         (*dump_stats)(struct Qdisc *, unsigned long,struct gnet_dump *);
};
```

内核定义并注册了许多调度器 Qdisc_ops 实例，接口函数 **register_qdisc**(struct Qdisc_ops *qops)用于注册 Qdisc_ops 实例，主要工作是将 Qdisc_ops 实例添加到全局单链表 qdisc_base 中。

注册函数 register_qdisc()定义在/net/sched/sch_api.c 文件内，源代码请读者自行阅读。各网络调度器的定义及注册在/net/sched/目录下的文件内实现。

■初始化

网络调度器初始化函数 pktsched_init()定义在/net/sched/sch_api.c 文件内，代码如下：

```
static int __init pktsched_init(void)
{
    int err;

    err = register_pernet_subsys(&psched_net_ops);   /*初始化函数中在 procfs 中创建"psched"文件*/
    ...    /*错误处理*/
    /*注册网络调度器实例*/
    register_qdisc(&pfifo_fast_ops);
    register_qdisc(&pfifo_qdisc_ops);
    register_qdisc(&bfifo_qdisc_ops);
    register_qdisc(&pfifo_head_drop_qdisc_ops);
    register_qdisc(&mq_qdisc_ops);

    /*注册 netlink 网络调度器相关消息类型处理函数*/
    rtnl_register(PF_UNSPEC, RTM_NEWQDISC, tc_modify_qdisc, NULL, NULL);
                                                        /*修改排队规则*/
    rtnl_register(PF_UNSPEC, RTM_DELQDISC, tc_get_qdisc, NULL, NULL);
    rtnl_register(PF_UNSPEC, RTM_GETQDISC, tc_get_qdisc, tc_dump_qdisc, NULL);
    rtnl_register(PF_UNSPEC, RTM_NEWTCLASS, tc_ctl_tclass, NULL, NULL);
    rtnl_register(PF_UNSPEC, RTM_DELTCLASS, tc_ctl_tclass, NULL, NULL);
    rtnl_register(PF_UNSPEC, RTM_GETTCLASS, tc_ctl_tclass, tc_dump_tclass, NULL);

    return 0;
}
subsys_initcall(pktsched_init);        /*内核初始化阶段调用此函数*/
```

初始化函数中主要工作是注册一些网络调度器 Qdisc_ops 实例，注册 NETLINK_ROUTE 套接字中与网络调度器相关消息类型的处理函数，以上函数都定义在/net/sched/sch_api.c 文件内，后面将简要介绍消息处理函数。

3 设置排队规则

在注册网络设备时，网络设备的发送缓存队列将关联空的排队规则。当激活网络设备时，将创建新的排队规则，并关联默认的网络调度器。另外，用户也可以通过 NETLINK_ROUTE 套接字设置/获取网络调度器信息。

■初始化排队规则

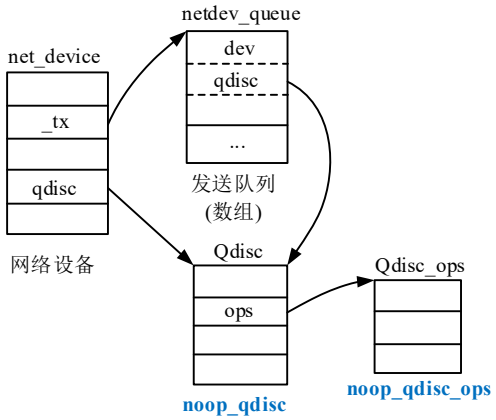
在分配网络设备实例时将为其分配发送数据包缓存队列（数组）netdev_queue 实例，在注册 net_device 实例的 register_netdevice(dev)函数中将调用 **dev_init_scheduler(dev)**函数设置初始的排队规则（调度器）。

dev_init_scheduler()函数定义如下（/net/sched/sch_generic.c）：

```
void dev_init_scheduler(struct net_device *dev)
{
    dev->qdisc = &noop_qdisc;       /*/net/sched/sch_generic.c*/
```

```
    netdev_for_each_tx_queue(dev, dev_init_scheduler_queue, &noop_qdisc);
    if (dev_ingress_queue(dev))
        dev_init_scheduler_queue(dev, dev_ingress_queue(dev), &noop_qdisc);

    setup_timer(&dev->watchdog_timer, dev_watchdog, (unsigned long)dev);
                                                    /*设置定时器，/net/sched/sch_generic.c*/
}
```

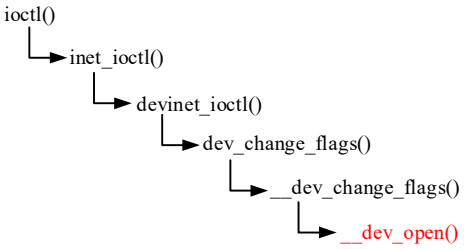
dev_init_scheduler()函数将网络设备关联排队规则 **noop_qdisc** 实例，然后遍历网络设备 net_device 实例中发送缓存队列数组中 netdev_queue 实例，调用函数 dev_init_scheduler_queue()对每个 netdev_queue 实例进行设置，主要工作是使队列 dev_queue->qdisc 和 dev_queue->qdisc_sleeping 成员都指向 **noop_qdisc** 实例，如下图所示。noop_qdisc 排队规则关联 noop_qdisc_ops 调度器，此调度器是一个空调度器，入队函数直接将数据包释放，出队函数直接返回 NULL。



dev_init_scheduler()函数还需要设置网络设备看门狗定时器，定时器回调函数为 **dev_watchdog()**。

■设置默认排队规则

用户在连接网络前需要通过 ifconfig 等命令配置网络设备 IP 地址，然后执行形如 ifconfig eth0 up 的命令启动网络设备。这时会执行 ioctl()系统调用，函数调用关系简列如下：



用户通过 NETLINK_ROUTE 套接字 RTM_NEWLINK 消息类型打开网络设备时，消息类型处理函数 rtnl_newlink()中最终也调用__dev_change_flags()（/net/core/rtnetlink.c）函数打开网络设备，源代码请读者自行阅读。

__dev_change_flags()函数根据网络设备新的标志值，调用不同的处理函数。对于 IFF_UP（打开）标志，将调用__dev_open()函数。

__dev_open()函数用于执行打开网络设备操作，函数定义如下（/net/core/dev.c）：

```
static int __dev_open(struct net_device *dev)
{
    const struct net_device_ops *ops = dev->netdev_ops;      /*网络设备操作结构*/
    int ret;
```

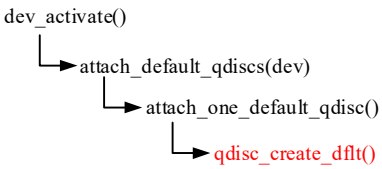
```
    ASSERT_RTNL();

    if (!netif_device_present(dev))      /*网络设备是否存在，/include/linux/netdevice.h*/
        return -ENODEV;

    netpoll_poll_disable(dev);

    ret = call_netdevice_notifiers(NETDEV_PRE_UP, dev);      /*执行 netdev_chain 通知链*/
    ret = notifier_to_errno(ret);
    if (ret)
        return ret;

    set_bit(__LINK_STATE_START, &dev->state);

    if (ops->ndo_validate_addr)
        ret = ops->ndo_validate_addr(dev);

    if (!ret && ops->ndo_open)
        ret = ops->ndo_open(dev);      /*调用网络设备操作结构中的打开函数*/

    netpoll_poll_enable(dev);

    if (ret)
        clear_bit(__LINK_STATE_START, &dev->state);
    else {
        dev->flags |= IFF_UP;   /*设置 IFF_UP 标志位*/
        dev_set_rx_mode(dev);       /*调用 ops->ndo_set_rx_mode(dev)函数等，/net/core/dev.c*/
        dev_activate(dev);          /*激活排队规则，/net/sched/sch_generic.c*/
        add_device_randomness(dev->dev_addr, dev->addr_len);
    }
    return ret;
}
```

在__dev_open()函数函数中，如果网络设备操作结构中定义了 ndo_open()函数，将会调用执行此函数，然后调用 **dev_activate()**函数激活排队规则。dev_activate()函数，函数定义如下（/net/sched/sch_generic.c）：

```
void dev_activate(struct net_device *dev)
{
    int need_watchdog;

    if (dev->qdisc == &noop_qdisc)      /*排队规则还是注册网络设备时设置的空排队规则*/
        attach_default_qdiscs(dev);      /*创建默认的新排队规则*/

    if (!netif_carrier_ok(dev))     /*如果需要延迟激活*/
        return;
```

```
    need_watchdog = 0;
    netdev_for_each_tx_queue(dev, transition_one_qdisc, &need_watchdog);
    if (dev_ingress_queue(dev))
        transition_one_qdisc(dev, dev_ingress_queue(dev), NULL);

    if (need_watchdog) {
        dev->trans_start = jiffies;
        dev_watchdog_up(dev);
    }
}
```

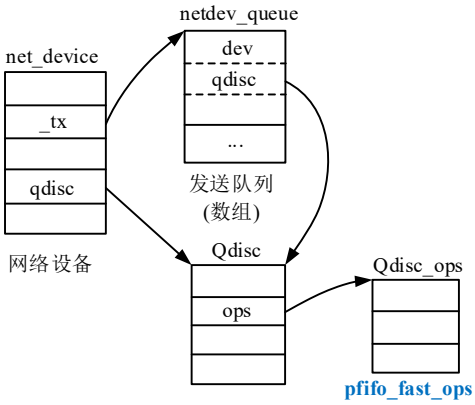
在注册网络设备时 dev->qdisc 成员被设置成 noop_qdisc 实例，因此此处调用 attach_default_qdiscs(dev)函数为网络设备创建并初始化新的默认排队规则，函数调用关系如下：



attach_default_qdiscs()函数定义如下（/net/sched/sch_generic.c）：

```
static void attach_default_qdiscs(struct net_device *dev)
{
    struct netdev_queue *txq;
    struct Qdisc *qdisc;

    txq = netdev_get_tx_queue(dev, 0);   /*指向网络设备发送缓存队列（数组）*/

    if (!netif_is_multiqueue(dev) || dev->tx_queue_len == 0) {     /*只有一个发送缓存队列*/
        netdev_for_each_tx_queue(dev, attach_one_default_qdisc, NULL);
                                                                   /*/net/sched/sch_generic.c*/
        dev->qdisc = txq->qdisc_sleeping;
        atomic_inc(&dev->qdisc->refcnt);
    } else {     /*多发送缓存队列或 dev->tx_queue_len 不为 0*/
        qdisc = qdisc_create_dflt(txq, &mq_qdisc_ops, TC_H_ROOT);
        if (qdisc) {
            dev->qdisc = qdisc;
            qdisc->ops->attach(qdisc);
        }
    }
}
```

对于只有一个发送缓存队列的情况，attach_default_qdiscs()函数内调用 attach_one_default_qdisc()函数为队列创建排队规则 Qdisc 实例，其关联的网络调度器为 default_qdisc_ops 全局指针指向的默认调度器实例，初始默认为 **pfifo_fast_ops**，如下图所示。



另外，在创建排队规则 Qdisc 实例后还将调用调度器初始化函数 ops->init()。**pfifo_fast_ops** 实例是一个数据包按优先级分类的先进先出的调度器，实现在/net/sched/sch_generic.c 文件内，源代码请读者自行阅读。

■用户接口

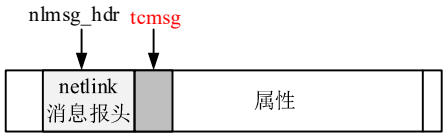
在打开网络设备时，内核为网络设备创建排队规则，并关联默认的网络调度器 **pfifo_fast_ops** 实例，用户可通过"default_qdisc"系统控制参数（proc 文件系统），设置默认的网络调度器（以名称标识）。

用户可通过 NETLINK_ROUTE 套接字消息设置/获取网络设备排队规则及其参数，相关的消息类型有：

- **RTM_NEWQDISC** /*设置排队规则新参数，处理函数 **tc_modify_qdisc()***/
- **RTM_DELQDISC** /*删除排队规则参数，处理函数 tc_get_qdisc()*/
- **RTM_GETQDISC** /*获取排队规则参数，处理函数 tc_get_qdisc()*/

在初始化函数 pktsched_init()中注册了以上类型消息的处理函数，例如 **RTM_NEWQDISC** 消息类型处理函数为 tc_modify_qdisc()。

以上消息类型中消息结构如下图所示：



消息类型报头由 tcmsg 结构体表示，定义如下（/include/uapi/linux/rtnetlink.h）：

```
struct tcmsg {
    unsigned char   tcm_family;    /*协议类型*/
    unsigned char   tcm__pad1;
    unsigned short      tcm__pad2;
    int         tcm_ifindex;     /*网络设备索引值*/
    __u32           tcm_handle;    /*需要修改或创建排队规则句柄值*/
    __u32           tcm_parent;    /*父排队规则句柄值*/
    __u32           tcm_info;
};
```

属性类型定义如下：

```
enum {
    TCA_UNSPEC,
```

```
    TCA_KIND,     /*排队规则关联的调度器名称*/
    TCA_OPTIONS,  /*选项*/
    TCA_STATS,
    TCA_XSTATS,
    TCA_RATE,
    TCA_FCNT,
    TCA_STATS2,
    TCA_STAB,
    __TCA_MAX
};
```

下面看一下设置网络设备排队规则新参数的 RTM_NEWQDISC 消息处理函数 **tc_modify_qdisc()**的实现，此函数用于创建或修改排队规则，其它函数源代码请读者自行阅读。

tc_modify_qdisc()函数定义如下（/net/sched/sch_api.c）：

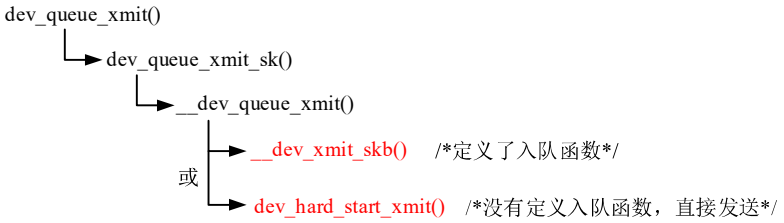
```
static int tc_modify_qdisc(struct sk_buff *skb, struct nlmsghdr *n)
{
    struct net *net = sock_net(skb->sk);
    struct tcmsg *tcm;
    struct nlattr *tca[TCA_MAX + 1];
    struct net_device *dev;
    u32 clid;
    struct Qdisc *q, *p;
    int err;

    if (!netlink_ns_capable(skb, net->user_ns, CAP_NET_ADMIN))
        return -EPERM;

replay:
    err = nlmsg_parse(n, sizeof(*tcm), tca, TCA_MAX, NULL);      /*解析属性*/
    if (err < 0)
        return err;

    tcm = nlmsg_data(n);     /*指向 tcmsg 结构体实例*/
    clid = tcm->tcm_parent;        /*父排队规则句柄*/
    q = p = NULL;

    dev = __dev_get_by_index(net, tcm->tcm_ifindex);      /*由索引值查找网络设备*/
    if (!dev)
        return -ENODEV;

    if (clid) {        /*clid 非零*/
        if (clid != TC_H_ROOT) {
            if (clid != TC_H_INGRESS) {
                p = qdisc_lookup(dev, TC_H_MAJ(clid));
                if (!p)
                    return -ENOENT;
```

```
                q = qdisc_leaf(p, clid);
            } else if (dev_ingress_queue_create(dev)) {
                q = dev_ingress_queue(dev)->qdisc_sleeping;
            }
        } else {
            q = dev->qdisc;
        }

        /*忽略默认的排队规则*/
        if (q && q->handle == 0)
            q = NULL;

        if (!q || !tcm->tcm_handle || q->handle != tcm->tcm_handle) {
            if (tcm->tcm_handle) {
                if (q && !(n->nlmsg_flags & NLM_F_REPLACE))
                    return -EEXIST;
                if (TC_H_MIN(tcm->tcm_handle))
                    return -EINVAL;
                q = qdisc_lookup(dev, tcm->tcm_handle);      /*查找指定句柄的排队规则*/
                if (!q)
                    goto create_n_graft;        /*没有找到则创建*/
                if (n->nlmsg_flags & NLM_F_EXCL)
                    return -EEXIST;
                if (tca[TCA_KIND] && nla_strcmp(tca[TCA_KIND], q->ops->id))
                    return -EINVAL;
                if (q == p ||
                    (p && check_loop(q, p, 0)))
                    return -ELOOP;
                atomic_inc(&q->refcnt);
                goto graft;
            } else {
                if (!q)
                    goto create_n_graft;

                if ((n->nlmsg_flags & NLM_F_CREATE) &&
                    (n->nlmsg_flags & NLM_F_REPLACE) &&
                    ((n->nlmsg_flags & NLM_F_EXCL) ||
                      (tca[TCA_KIND] &&
                        nla_strcmp(tca[TCA_KIND], q->ops->id))))
                    goto create_n_graft;
            }
        }
    } else {       /*clid 为零*/
        if (!tcm->tcm_handle)
            return -EINVAL;
        q = qdisc_lookup(dev, tcm->tcm_handle);      /*查找指定排队规则*/
```

```
    }

    /*修改排队规则参数*/
    if (q == NULL)
        return -ENOENT;
    if (n->nlmsg_flags & NLM_F_EXCL)
        return -EEXIST;
    if (tca[TCA_KIND] && nla_strcmp(tca[TCA_KIND], q->ops->id))
        return -EINVAL;
    err = qdisc_change(q, tca);
                    /*调用 q->ops->change()函数修改排队规则参数等，/net/sched/sch_api.c*/
    if (err == 0)
        qdisc_notify(net, skb, n, clid, NULL, q);      /*向用户套接字发送消息*/
    return err;

create_n_graft:     /*创建新排队规则，关联指定的网络调度器等*/
    if (!(n->nlmsg_flags & NLM_F_CREATE))
        return -ENOENT;
    if (clid == TC_H_INGRESS) {
        if (dev_ingress_queue(dev))
            q = qdisc_create(dev, dev_ingress_queue(dev), p,
                        tcm->tcm_parent, tcm->tcm_parent, tca, &err);
        else
            err = -ENOENT;
    } else {
        struct netdev_queue *dev_queue;

        if (p && p->ops->cl_ops && p->ops->cl_ops->select_queue)
            dev_queue = p->ops->cl_ops->select_queue(p, tcm);
        else if (p)
            dev_queue = p->dev_queue;
        else
            dev_queue = netdev_get_tx_queue(dev, 0);

        q = qdisc_create(dev, dev_queue, p,tcm->tcm_parent, tcm->tcm_handle,tca, &err);
                        /*创建新排队规则，tca[TCA_KIND]属性指定了调度器，/net/sched/sch_api.c*/
    }
    ...    /*错误处理*/

graft:
    err = qdisc_graft(dev, p, skb, n, clid, q, NULL);
                            /*排队规则关联到队列，重新激活网络设备等，/net/sched/sch_api.c*/
    ...    /*错误处理*/

    return 0;
}
```

4 发送数据包函数

由前一节的介绍可知，网络层协议调用接口函数 **dev_queue_xmit()**将数据包发送给数据链路层，函数调用关系简列如下图所示：



dev_queue_xmit()最终调用__dev_queue_xmit()函数执行发送数据包操作。如果网络设备排队规则（调度器）未定义入队函数，则调用 **dev_hard_start_xmit**()函数立即发送数据包，此函数内将最终调用网络设备操作结构中的 net_device_ops->**ndo_start_xmit**(skb, dev)函数将数据包发送到网络设备。

如果网络设备排队规则（调度器）定义入队函数，则调用**__dev_xmit_skb()**函数发送数据包，详见下文。

■接口函数

dev_queue_xmit()函数定义如下（/include/linux/netdevice.h）：

```
static inline int dev_queue_xmit(struct sk_buff *skb)
/*skb：数据包（队列）*/
{
    return dev_queue_xmit_sk(skb->sk, skb);      /*/net/core/dev.c*/
}
```

dev_queue_xmit_sk()函数定义如下：

```
int dev_queue_xmit_sk(struct sock *sk, struct sk_buff *skb)
{
    return __dev_queue_xmit(skb, NULL);      /*发送数据包，/net/core/dev.c*/
}
```

数据包最终由__dev_queue_xmit()函数发送，代码如下：

```
static int __dev_queue_xmit(struct sk_buff *skb, void *accel_priv)
/*skb：指向要发送的数据包（队列），accel_priv：指向私有数据，此处为 NULL*/
{
    struct net_device *dev = skb->dev;      /*网络设务*/
    struct netdev_queue *txq;      /*网络设备发送数据包缓存队列*/
    struct Qdisc *q;   /*排队规则*/
    int rc = -ENOMEM;

    skb_reset_mac_header(skb);

    if (unlikely(skb_shinfo(skb)->tx_flags & SKBTX_SCHED_TSTAMP))
        __skb_tstamp_tx(skb, NULL, skb->sk, SCM_TSTAMP_SCHED);

    rcu_read_lock_bh();
```

```
    skb_update_prio(skb);    /*更新数据包优先级（需配置 CGROUP_NET_PRIO），/net/core/dev.c*/

    if (dev->priv_flags & IFF_XMIT_DST_RELEASE)
        skb_dst_drop(skb);
    else
        skb_dst_force(skb);    /*确保 skb->dst 被引用，/include/net/dst.h*/

    txq = netdev_pick_tx(dev, skb, accel_priv);    /*选择发送缓存队列，/net/core/dev.c*/
    q = rcu_dereference_bh(txq->qdisc);    /*指向队列排队规则*/

#ifdef CONFIG_NET_CLS_ACT
    skb->tc_verd = SET_TC_AT(skb->tc_verd, AT_EGRESS);
#endif
    trace_net_dev_queue(skb);
    if (q->enqueue) {        /*如果排队规则（调度器）定义了入队函数，一般都会定义*/
        rc = __dev_xmit_skb(skb, q, dev, txq);
                    /*将数据包添加到排队规则中数据包队列再发送，见下文，/net/core/dev.c*/
        goto out;       /*跳转到 out 处执行*/
    }

    /*如果排队规则（调度器）没有定义入队函数*/
    if (dev->flags & IFF_UP) {        /*如果网络设备已启用*/
        int cpu = smp_processor_id();       /*当前 CPU 编号*/
        if (txq->xmit_lock_owner != cpu) {
            if (__this_cpu_read(xmit_recursion) > RECURSION_LIMIT)
                goto recursion_alert;

            skb = validate_xmit_skb(skb, dev);    /*数据包有效性检查，/net/core/dev.c*/
            if (!skb)
                goto drop;

            HARD_TX_LOCK(dev, txq, cpu);

            if (!netif_xmit_stopped(txq)) {    /*发送队列是否关闭，/include/linux/netdevice.h*/
                __this_cpu_inc(xmit_recursion);
                skb = dev_hard_start_xmit(skb, dev, txq, &rc);    /*/net/core/dev.c*/
                            /*最终调用 net_device_ops->ndo_start_xmit(skb, dev)发送数据包*/
                __this_cpu_dec(xmit_recursion);
                if (dev_xmit_complete(rc)) {       /*发送是否完成，/include/linux/netdevice.h*/
                    HARD_TX_UNLOCK(dev, txq);
                    goto out;
                }
            }
            HARD_TX_UNLOCK(dev, txq);
            ...
```

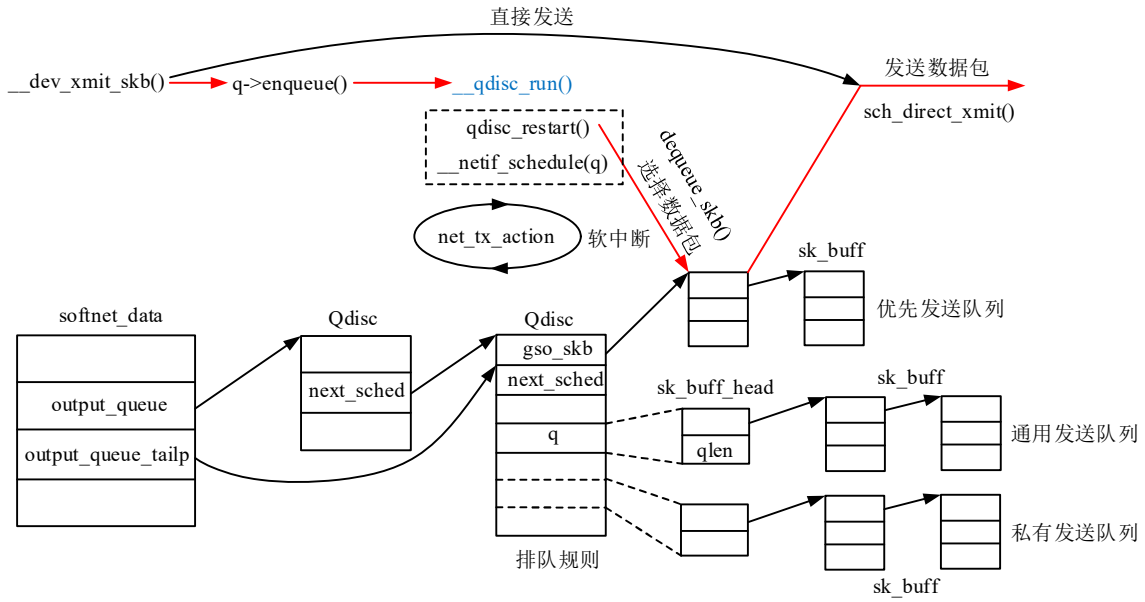
```
            } else {
                ...
            }
        }    /*如果排队规则（调度器）没有定义入队函数情形处理完毕*/
        ...
out:
        rcu_read_unlock_bh();
        return rc;
}
```

__dev_queue_xmit()函数用于发送数据包，也可以是一个数据包列表。如果排队规则（调度器）没有定义入队函数，则调用 dev_hard_start_xmit()函数立即发送数据包，此函数将对数据包列表中每个数据包调用网络设备操作结构中的 net_device_ops->**ndo_start_xmit**(skb, dev)函数将数据包发送至网络设备，源代码请读者自行阅读。

如果排队规则（调度器）定义入队函数，__dev_queue_xmit()函数将调用**__dev_xmit_skb**()函数发送数据包，下面将详细介绍__dev_xmit_skb()函数的实现。

■通过调度器发送

排队规则 Qdisc 结构体中的 q 成员是一个 sk_buff_head 结构体成员，表示通用的发送数据包缓存队列，q.qlen 成员记录了队列中数据包的数量。但是，具体网络调度器可以不将发送数据包添加到 q 队列，而是建立自己的私有队列（pfifo_fast_ops 调度器就定义了按优先级分类的私有队列数组），如下图所示。由于数据包入队/出队操作是由调度器类定义的，因此数据包具体在哪个队列是由调度器决定，而不是排队规则决定的。不管数据包在哪个队列，其数量依然记录在 Qdisc->q.qlen 成员中，对排队规则假装数据包添加在 q 队列。内核通过检测 Qdisc->q.qlen 成员值确定发送队列长度（是否为空）。



__dev_xmit_skb()函数发送数据包时，如果调度器允许旁路，且排队规则发送队列长度为 0，则直接发送数据包，由 sch_direct_xmit()函数将数据包发往网络设备。如果由于网络设备忙等原因导致发送失败，则将数据包添加到 gso_skb 队列。gso_skb 队列中数据包数量也累加到 Qdisc->q.qlen 成员。

如果发送队列不为空或调度器不允许旁路，则将数据包插入发送队列，调用__qdisc_run()函数发送队列中的数据包。

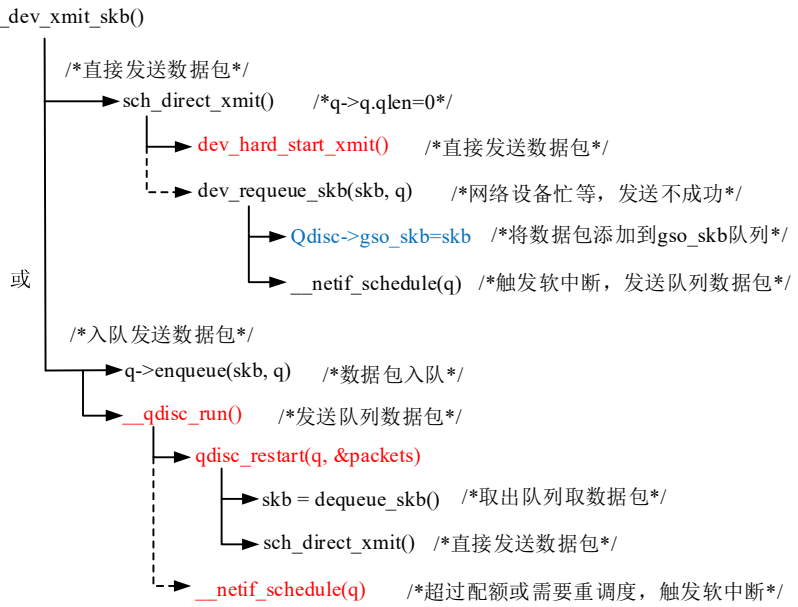
__qdisc_run()函数主要工作是分批从排队规则发送队列中取出数据包（优先从 gso_skb 队列取数据包），

发送到网络设备。如果发送不成功（如网络设备忙），数据包将会添加到 Qdisc->gso_skb 队列。下次发送数据包时将优先从此队列取数据包发送，发送完本队列数据包后再从 q 或私有队列取数据包发送。

如果__qdisc_run()函数发送的数据包数量达到了配额或需要执行进程重调度，则将调用函数**__netif_schedule(q)**将 Qdisc 实例添加到 softnet_data 实例 Qdisc 实例链表中，并触发 NET_TX_SOFTIRQ 软中断，__qdisc_run()函数返回。

在软中断处理函数中将遍历 softnet_data 实例 Qdisc 实例链表，对每个实例调用__qdisc_run()函数分批发送排队规则队列中的数据包。如果队列中数据包未发送完，Qdisc 实例将被再次添加到 softnet_data 实例中的队列，触发软中断，重复分批发送的操作。

__dev_xmit_skb()函数调用关系简列如下图所示：



__dev_xmit_skb()函数定义如下（/net/core/dev.c）：

```
static inline int __dev_xmit_skb(struct sk_buff *skb, struct Qdisc *q,struct net_device *dev,
                                                                    struct netdev_queue *txq)
{
    spinlock_t *root_lock = qdisc_lock(q);
    bool contended;
    int rc;

    qdisc_pkt_len_init(skb);    /*数据包长度初始化，/net/core/dev.c*/
    qdisc_calculate_pkt_len(skb, q);    /*/include/net/sch_generic.h*/

    contended = qdisc_is_running(q);    /*排队规则是否已启用，/include/net/sch_generic.h*/
    if (unlikely(contended))
        spin_lock(&q->busylock);

    spin_lock(root_lock);
    if (unlikely(test_bit(__QDISC_STATE_DEACTIVATED, &q->state))) {
        kfree_skb(skb);                /*排队规则未激活，释放数据包*/
        rc = NET_XMIT_DROP;
    } else if ((q->flags & TCQ_F_CAN_BYPASS) && !qdisc_qlen(q) &&qdisc_run_begin(q)) {
```

```
        /*排队规则设置了 TCQ_F_CAN_BYPASS 标志位（pfifo_fast_ops 初始化函数中置位，允许旁路），
        *排队规则中发送数据包队列为空，
        *设置排队规则启用标志。*/
            qdisc_bstats_update(q, skb);       /*更新统计量*/
            if (sch_direct_xmit(skb, q, dev, txq, root_lock, true)) {     /*/net/sched/sch_generic.c*/
                                        /*直接发送数据包，返回 q->q.qlen 值*/
                if (unlikely(contended)) {
                    spin_unlock(&q->busylock);
                    contended = false;
                }
                __qdisc_run(q);       /*直接发送后，如果发送队列不为空，则发送队列数据包*/
            } else     /*如果发送队列为空，q->q.qlen=0*/
                qdisc_run_end(q);     /*清除排队规则__QDISC___STATE_RUNNING 标记*/

            rc = NET_XMIT_SUCCESS;
        } else {        /*排队规则数据包队列非空，或没有设置 TCQ_F_CAN_BYPASS 标志位等情况*/
            rc = q->enqueue(skb, q) & NET_XMIT_MASK;     /*调用入队函数*/
            if (qdisc_run_begin(q)) {     /*设置排队规则标志位（启用）*/
                if (unlikely(contended)) {
                    spin_unlock(&q->busylock);
                    contended = false;
                }
                __qdisc_run(q);       /*发送队列数据包*/
            }
        }
        spin_unlock(root_lock);
        if (unlikely(contended))
            spin_unlock(&q->busylock);
        return rc;
    }
```

__dev_xmit_skb()函数内判断如果排队规则发送数据包队列为空，且排队规则允许被旁路，则设置排队规则启用标记，调用函数 sch_direct_xmit()直接发送数据包，如果此函数返回非零值（发送队列数据包数量），则调用__qdisc_run(q)函数发送队列列中数据包。

如果排队规则发送数据包队列非空，或排队规则不允许旁路，则入队函数，将数据包添加到排队规则发送队列，然后再调用__qdisc_run(q)函数发送队列中数据包。

●直接发送数据包

sch_direct_xmit()函数用于直接发送数据包，函数定义如下（/net/sched/sch_generic.c）：

```
int sch_direct_xmit(struct sk_buff *skb, struct Qdisc *q,
                    struct net_device *dev, struct netdev_queue *txq,spinlock_t *root_lock, bool validate)
{
    int ret = NETDEV_TX_BUSY;
    spin_unlock(root_lock);
    if (validate)
        skb = validate_xmit_skb_list(skb, dev);     /*有效性判断*/
```

```
        if (skb) {
            HARD_TX_LOCK(dev, txq, smp_processor_id());
            if (!netif_xmit_frozen_or_stopped(txq))
                                        /*网络设备发送缓存队列是否可用，/include/linux/netdevice.h*/
                skb = dev_hard_start_xmit(skb, dev, txq, &ret);     /*直接发送*/

            HARD_TX_UNLOCK(dev, txq);
        }
        spin_lock(root_lock);

        if (dev_xmit_complete(ret)) {     /*发送是否成功，如果网络设备忙将返回不成功*/
            ret = qdisc_qlen(q);       /*q->q.qlen*/
        } else if (ret == NETDEV_TX_LOCKED) {
            ret = handle_dev_cpu_collision(skb, txq, q);
        } else {   /*网络设备忙，发送不成功，将未发送数据包添加到 gso_skb 队列，增加 q->q.qlen 值*/
            /* Driver returned NETDEV_TX_BUSY - requeue skb */
            if (unlikely(ret != NETDEV_TX_BUSY))
                net_warn_ratelimited("BUG %s code %d qlen %d\n",dev->name, ret, q->q.qlen);

            ret = dev_requeue_skb(skb, q);       /*/net/sched/sch_generic.c*/
                /*将数据包添加到 Qdisc->gso_skb 暂存队列，调用__netif_schedule(q)函数等，返回 0*/
        }

        if (ret && netif_xmit_frozen_or_stopped(txq))
            ret = 0;

        return ret;
    }
```

sch_direct_xmit()函数首先调用 dev_hard_start_xmit()函数将数据包直接发送给网络设备，如果发送成功，函数返回 Qdisc->q.qlen 值。如果发送不成功（如网络设备忙），随后调用 dev_requeue_skb(skb, q)函数将数据包添加到 Qdisc->gso_skb 暂存队列，并调用__netif_schedule(q)函数触发发送数据包软中断。

sch_direct_xmit()函数返回队列中数据包数量，如果不为 0，则函数返回后，在__dev_xmit_skb()函数还将调用__qdisc_run(q)函数发送队列中数据包。

●发送队列数据包

如果排队规则发送数据包队列不为空，或不允许旁路，则__dev_xmit_skb()函数将数据包插入到排队规则发送队列，然后调用__qdisc_run()函数发送排队规则发送队列中的数据包。

__qdisc_run()函数定义如下（/net/sched/sch_generic.c）：

```
void __qdisc_run(struct Qdisc *q)
{
    int quota = weight_p;     /*发送配额，初始值为 64*/
    int packets;

    while (qdisc_restart(q, &packets)) {     /*发送队列数据包，/net/sched/sch_generic.c*/
        quota -= packets;       /*配额减小*/
        if (quota <= 0 || need_resched()) {   /*配额用完或需要进程重调度*/
```

```
                __netif_schedule(q);      /*触发软中断，跳出循环，/net/core/dev.c*/
                break;
            }
        }
        qdisc_run_end(q);      /*设置关闭标志位*/
    }
```

__qdisc_run()函数循环调用 qdisc_restart(q, &packets)函数从排队规则发送队列中取出数据包并发送，参数 packets 表示发送的数据包数量。

如果发送数据包数量超过配额或需要执行重调度，则调用__netif_schedule(q)函数将 Qdisc 实例添加到 softnet_data->output_queue 链表，触发发送数据包 **NET_TX_SOFTIRQ** 软中断，函数返回。在软中断处理函数中将继续调用__qdisc_run()函数发送队列中数据包，详见下文。

qdisc_restart(q, &packets)函数定义如下，用于从队列中取出数据包并发送：

```
static inline int qdisc_restart(struct Qdisc *q, int *packets)
{
    struct netdev_queue *txq;
    struct net_device *dev;
    spinlock_t *root_lock;
    struct sk_buff *skb;
    bool validate;

    /* Dequeue packet */
    skb = dequeue_skb(q, &validate, packets);      /*/net/sched/sch_generic.c*/
                /*取出数据包，优先取 Qdisc->gso_skb 队列，然后才调用调度器的出队函数*/
    if (unlikely(!skb))
        return 0;

    root_lock = qdisc_lock(q);
    dev = qdisc_dev(q);      /*网络设备*/
    txq = skb_get_tx_queue(dev, skb);     /*网络设备发送数据包缓存队列*/

    return sch_direct_xmit(skb, q, dev, txq, root_lock, validate);
                                        /*直接发送数据包，见上文，/net/sched/sch_generic.c*/
}
```

__qdisc_run()函数发送了配额规定的数据包或需要重调度时，将中止数据包发送，触发软中断。下面将介绍发送队列数据包 **NET_TX_SOFTIRQ** 软中断的实现。

5 发送数据包软中断

内核为网络设备发送、接收数据包定义了软中断，软中断编号如下（/include/linux/interrupt.h）：

```
enum
{
    HI_SOFTIRQ=0,       /*高优先级 tasklet*/
    TIMER_SOFTIRQ,     /*处理低分辨率定时器软中断*/
    NET_TX_SOFTIRQ,     /*网络发送数据软中断*/
```

```
    NET_RX_SOFTIRQ,     /*网络接收数据软中断*/
    ...
}
```

在网络设备初始化函数 net_dev_init()中注册了接收、发送数据包软中断，代码简列如下：

```
static int __init net_dev_init(void)
{
    ...
    for_each_possible_cpu(i) {     /*初始化 softnet_data 实例*/
        struct softnet_data *sd = &per_cpu(softnet_data, i);

        skb_queue_head_init(&sd->input_pkt_queue);
        skb_queue_head_init(&sd->process_queue);
        INIT_LIST_HEAD(&sd->poll_list);
        sd->output_queue_tailp = &sd->output_queue;     /*初始化*/
    #ifdef CONFIG_RPS
        sd->csd.func = rps_trigger_softirq;
        sd->csd.info = sd;
        sd->cpu = i;
    #endif

        sd->backlog.poll = process_backlog;       /*接收数据包函数，详见下一小节*/
        sd->backlog.weight = weight_p;
    }
    ...
    open_softirq(NET_TX_SOFTIRQ, net_tx_action);     /*注册发送数据包软中断*/
    open_softirq(NET_RX_SOFTIRQ, net_rx_action);     /*注册接收数据包软中断*/
    ...
out:
    return rc;
}
```

■触发软中断

在前面介绍的发送队列数据包的__qdisc_run()函数中，在发送数据包达到配额或需要执行进程重调度时，将中止发送，调用__netif_schedule()函数将 Qdisc 实例添加到 **softnet_data->output_queue** 链表末尾，并触发接收数据包软中断。

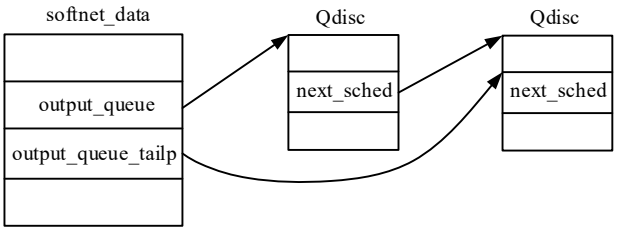
__netif_schedule()定义如下（/net/core/dev.c）：

```
void __netif_schedule(struct Qdisc *q)
{
    if (!test_and_set_bit(__QDISC_STATE_SCHED, &q->state))   /*是否加入 softnet_data 链表*/
                                    /*设置__QDISC_STATE_SCHED 标记位，原值需为 0*/
        __netif_reschedule(q);      /*触发软中断*/
}
```

__netif_reschedule()函数定义如下：

```
static inline void __netif_reschedule(struct Qdisc *q)
{
    struct softnet_data *sd;
    unsigned long flags;

    local_irq_save(flags);
    sd = this_cpu_ptr(&softnet_data);       /*CPU 核对应的 softnet_data 实例*/
    q->next_sched = NULL;
    /*以下两条语句的功能是将 Qdisc 实例添加到 softnet_data->output_queue 链表末尾*/
    *sd->output_queue_tailp = q;    /*排队规则添中到链表末尾*/
    sd->output_queue_tailp = &q->next_sched;     /*指向末尾排队规则*/
    raise_softirq_irqoff(NET_TX_SOFTIRQ);     /*触发软中断*/
    local_irq_restore(flags);
}
```

接收数据包软中断需要处理绑定到当前 CPU 的所有 Qdisc 实例（系统内可能有多个网络设备，有多个排队规则）。softnet_data 实例中包含 Qdisc 实例链表，如下图所示：



__netif_reschedule()函数将 Qdisc 实例添加到 softnet_data 实例中 Qdisc 实例链表的末尾，然后触发发送数据包软中断。

■软中断处理函数

发送数据包 NET_TX_SOFTIRQ 软中断处理函数内将遍历 softnet_data 实例中 Qdisc 实例链表，对每个 Qdisc 实例调用 qdisc_run(q)函数，执行发送队列数据包操作。

NET_TX_SOFTIRQ 软中断处理函数为 **net_tx_action()**，定义如下（/net/core/dev.c）：

```
static void net_tx_action(struct softirq_action *h)
{
    struct softnet_data *sd = this_cpu_ptr(&softnet_data);

    if (sd->completion_queue) {     /*释放完成队列中的 sk_buff 实例*/
        struct sk_buff *clist;

        local_irq_disable();
        clist = sd->completion_queue;
        sd->completion_queue = NULL;
        local_irq_enable();

        while (clist) {
            struct sk_buff *skb = clist;
            clist = clist->next;
```

```
                WARN_ON(atomic_read(&skb->users));
                if (likely(get_kfree_skb_cb(skb)->reason == SKB_REASON_CONSUMED))
                    trace_consume_skb(skb);
                else
                    trace_kfree_skb(skb, net_tx_action);
                __kfree_skb(skb);
            }
        }

        /*发送排队规则队列中的数据包*/
        if (sd->output_queue) {     /*softnet_data->output_queue 链表不为空*/
            struct Qdisc *head;

            local_irq_disable();
            head = sd->output_queue;     /*取出链表*/
            sd->output_queue = NULL;     /*链表清空*/
            sd->output_queue_tailp = &sd->output_queue;    /*恢复初始值*/
            local_irq_enable();

            while (head) {      /*遍历 softnet_data->output_queue 链表 Qdisc 实例*/
                struct Qdisc *q = head;
                spinlock_t *root_lock;

                head = head->next_sched;     /*下一个 Qdisc 实例*/

                root_lock = qdisc_lock(q);
                if (spin_trylock(root_lock)) {      /*获取锁*/
                    smp_mb__before_atomic();
                    clear_bit(__QDISC_STATE_SCHED,&q->state);
                    qdisc_run(q);
                      /*发送排队规则队列中数据包，调用__qdisc_run(q)函数，/include/net/pkt_sched.h*/
                    spin_unlock(root_lock);
                } else {      /*获取锁失败*/
                    if (!test_bit(__QDISC_STATE_DEACTIVATED,&q->state)) {     /*标记位为 0*/
                        __netif_reschedule(q);     /*重新触发软中断*/
                    } else {      /*__QDISC_STATE_DEACTIVATED 标记位为 1，关闭排队规则*/
                        smp_mb__before_atomic();
                        clear_bit(__QDISC_STATE_SCHED,&q->state);
                    }
                }
            }   /*遍历 softnet_data->output_queue 链表结束*/
        }   /*if (sd->output_queue)  结束*/
    }
```

net_tx_action()函数首先需要释放 softnet_data 实例中完成队列中的 sk_buff 实例，这是接收操作完成的

sk_buff 实例，详见下一小节。net_tx_action()函数随后取出 Qdisc 实例 **softnet_data->output_queue** 链表，原链表清空，恢复初始值，遍历链表中 Qdisc 实例，对每个实例调用 qdisc_run(q)函数执行发送队列数据包操作。

qdisc_run(q)函数定义如下：

```
static inline void qdisc_run(struct Qdisc *q)
{
    if (qdisc_run_begin(q))
        __qdisc_run(q);      /*函数定义见上文*/
}
```

13.2.4 接收数据包

网络设备从物理链路中接收到数据包后将产生中断，网络设备的中断处理程序是接收数据包的起点。在中断处理程序中将构建 sk_buff 实例，并向上层提交。

网络设备接收的数据包将缓存在 softnet_data 实例的接收数据包队列中，由 NET_RX_SOFTIRQ 软中断将此队列中数据包传递给网络层。

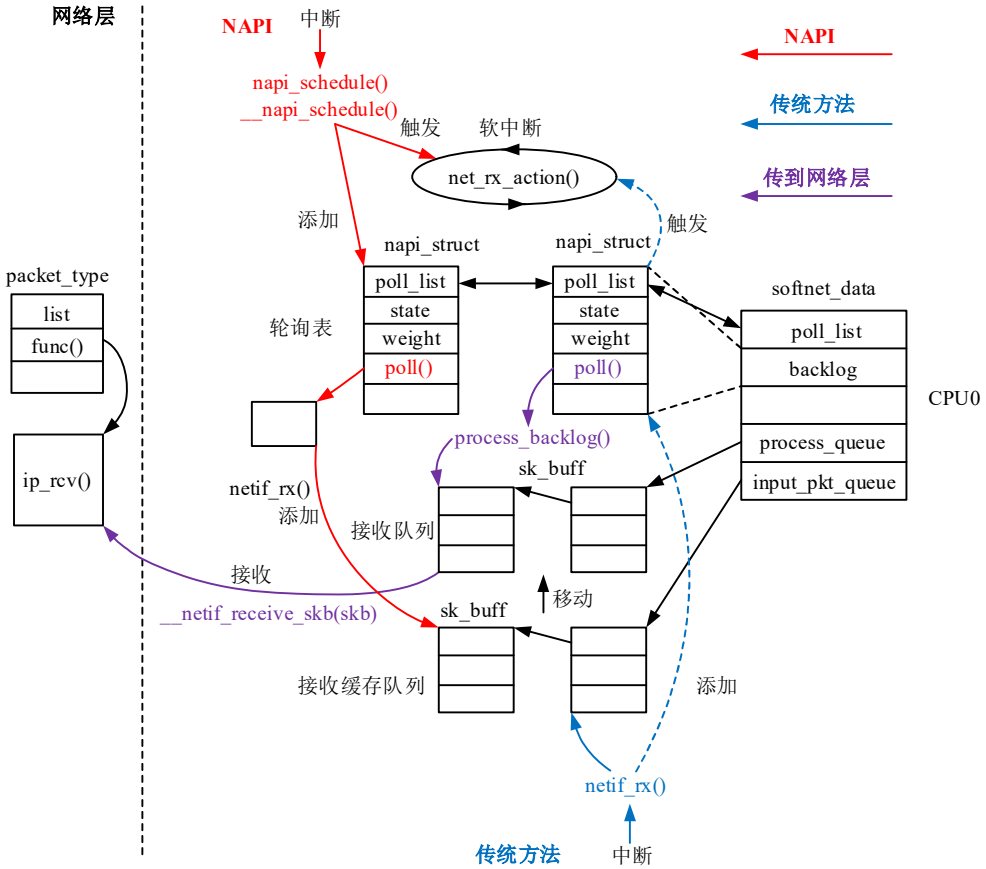
1 概述

网络设备从物理链路中收到数据帧后，将会产生中断。中断处理程序负责从网络设备中取出数据帧，构建 sk_buff 实例，并向上传递。内核为网络设备接收数据帧提供了两种方法：一是传统方法，二是 NAPI 方法。

在传统的接收方法中，网络设备在每个数据帧到达时都将产生中断，进入中断处理程序。随着网络速度的提高，传统方法在每个数据包到达时都要产生中断，增加了额外的开销，如果前一个数据包尚未处理完，下一个数据包又到达了，将会导致问题。为此内核实现了高速方法，通常称为 NAPI（New API）方法。高速方法下，在第一个数据包到达时网络设备将产生中断，为了防止产生更多的中断，将关闭网络设备中断，并将网络设备置于一个轮询表，轮询表中包含多个网络设备。只要网络设备一直有数据包到达就一直位于轮询表中，否则从轮询表中移除，开启网络设备中断。

内核定义了接收数据包 NET_RX_SOFTIRQ 软中断，软中断处理程序将不断轮询表中的网络设备，调用其定义的接收数据包函数接收数据包。

现在几乎所有的网络设备都支持 NAPI 方法，因此传统方法也已集成到了 NAPI 框架中，如下图所示。



内核为每个 CPU 核定义了 softnet_data 结构体实例，结构体中与接收数据包相关的成员有一个接收数据包缓存队列，一个接收（处理）队列，一个轮询表（链表），以及 napi_struct 结构体成员 backlog。轮询表是 napi_struct 结构体实例双链表，其中的 poll()函数是网络设备驱动程序中定义的接收数据包函数。由此可知 napi_struct 实例需由网络设备驱动程序定义，并在接收到第一个数据帧时将其注册到轮询表中。

NET_RX_SOFTIRQ 软中断负责轮询表中 napi_struct 实例，调用各实例中的 poll()实例接收数据包。各 napi_struct 实例中的 poll()函数通常是将数据包插入到 softnet_data 实例接收缓存队列 input_pkt_queue 中。

softnet_data 结构体中内嵌一个 napi_struct 结构体成员 backlog，此实例由内核设置并注册到轮询表，其 poll()函数为 process_backlog()，此函数的功能是将接收缓存队列 input_pkt_queue 中的数据包移动到接收队列 process_queue 中，然后再将接收队列中数据包传递到网络层。也就是说，网络设备 napi_struct 实例的 poll()函数将接收到的数据包添加到接收缓存队列即可，在接收软中断轮询到 softnet_data.backlog 实例时，由其 poll()函数 **process_backlog()**将接收缓存队列中数据包最终传递到网络层。

接口函数 netif_rx(skb)负责将数据包 skb 添加到 softnet_data 实例接收缓存队列。如果添加前队列为空，则需要先将 softnet_data.backlog 实例添加到轮询表，触发接收数据包软中断，然后再添加数据包到缓存队列。。

在传统方法中，中断处理程序中构建 sk_buff 实例后，可以直接调用 netif_rx(skb)函数将数据包添加到接收缓存队列即可从中断返回。

在 NAPI 方法中，在中断处理程序中将调用 napi_schedule()/__napi_schedule()接口函数将网络设备驱动程序定义的 napi_struct 实例添加到轮询表，并触发接收软中断。

NET_RX_SOFTIRQ 软中断处理函数将遍历轮询表中的 napi_struct 实例调用其中的 poll()函数接收数据包。softnet_data.backlog 实例中的 poll()函数会将接收缓存队列中的数据包先移动到接收队列，然后再传递到网络层。

2 数据结构

接收数据包流程相关的数据结构主要有 softnet_data 和 napi_struct。softnet_data 结构体此处主要用于管理轮询表和接收数据包。

napi_strcut 结构体实例由网络设备驱动程序定义并注册到 softnet_data 实例的轮询表中。softnet_data 结构体内嵌的 backlog 成员（napi_struct 实例）由内核设置，并添加到轮询表。

■softnet_data

softnet_data 结构体在前面介绍过了，下面简要列出与接收数据包相关的成员：

```
struct softnet_data {                    /*/include/linux/netdevice.h*/
    struct list_head      poll_list;     /*轮询表，管理 napi_strcut 实例*/
    struct sk_buff_head   process_queue;
                          /*正在接收（处理）的数据包队列，来自 input_pkt_queue 队列*/
    /*状态*/
    unsigned int          processed;     /*已接收的数据包数量*/
    unsigned int          time_squeeze;
    unsigned int          cpu_collision;
    unsigned int          received_rps;
    ...
    struct sk_buff        *completion_queue;   /*接收完成数据包链表*/
    ...
    unsigned int          dropped;
    struct sk_buff_head   input_pkt_queue;  /*接收数据包缓存队列，处理时移入 process_queue 链表*/
    struct napi_struct    backlog;       /*处理接收缓存队列的 napi_struct 实例*/
}
```

内核在/net/core/dev.c 文件内为每个 CPU 核定义了 softnet_data 实例（percpu 变量）：

```
DEFINE_PER_CPU_ALIGNED(struct softnet_data, softnet_data);
```

在网络设备初始化函数 net_dev_init()中初始化了 softnet_data 实例，代码简列如下：

```
static int __init net_dev_init(void)
{
    ...
    for_each_possible_cpu(i) {    /*遍历 softnet_data 实例*/
        struct softnet_data *sd = &per_cpu(softnet_data, i);

        skb_queue_head_init(&sd->input_pkt_queue);  /*初始化接收缓存队列链表头*/
        skb_queue_head_init(&sd->process_queue);    /*初始化接收队列链表头*/
        INIT_LIST_HEAD(&sd->poll_list);
        sd->output_queue_tailp = &sd->output_queue;
    #ifdef CONFIG_RPS
        sd->csd.func = rps_trigger_softirq;
        sd->csd.info = sd;
        sd->cpu = i;
    #endif
```

```
        sd->backlog.poll = process_backlog;        /*赋值 softnet_data.backlog 实例的 poll()函数*/
        sd->backlog.weight = weight_p;     /*配额，默认值为 64*/
    }
    ...
    open_softirq(NET_TX_SOFTIRQ, net_tx_action);     /*注册发送数据包软中断*/
    open_softirq(NET_RX_SOFTIRQ, net_rx_action);     /*注册接收数据包软中断*/
    ...
}
```

■napi_struct

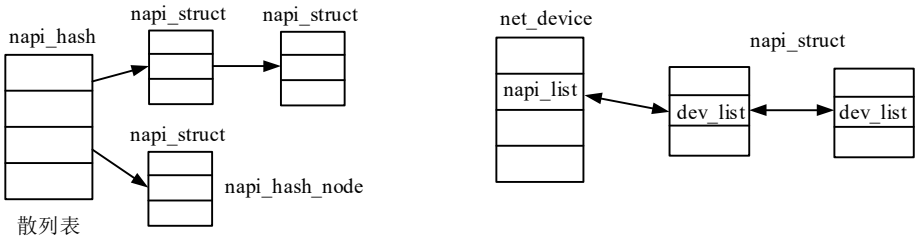
轮询表中成员为 napi_struct 结构体实例，结构体定义在/include/linux/netdevice.h 头文件：

```
struct napi_struct {
    struct list_head    poll_list;   /*将实例添加到轮询表*/
    unsigned long       state;     /*状态*/
    int                 weight;     /*压力值，接收数据包配额*/
    unsigned int        gro_count;
    int                 (*poll)(struct napi_struct *, int);     /*轮询函数，接收数据包*/
#ifdef CONFIG_NETPOLL
    spinlock_t          poll_lock;
    int                 poll_owner;
#endif
    struct net_device   *dev;        /*网络设备*/
    struct sk_buff      *gro_list;
    struct sk_buff      *skb;
    struct hrtimer      timer;      /*高分辨率定时器*/
    struct list_head    dev_list;
                        /*将实例添加到 net_device 实例链表中，网络设备可有多个 napi_struct 实例*/
    struct hlist_node   napi_hash_node;     /*将实例添加到散列表*/
    unsigned int        napi_id;      /*id 值*/
};
```

napi_struct 结构体状态 state 成员取值定义如下：

```
enum {
    NAPI_STATE_SCHED,       /*设备可接受轮询*/
    NAPI_STATE_DISABLE,     /*设备挂起*/
    NAPI_STATE_NPSVC,       /* Netpoll - don't dequeue from poll_list */
    NAPI_STATE_HASHED,      /* In NAPI hash */
};
```

内核定义了全局散列表 napi_hash 用于管理 napi_struct 实例，网络设备 net_device 实例中 napi_list 成员链接了本设备的 napi_struct 实例，如下图所示：



以下接口函数用于将 napi_struct 实例添加到全局散列表和 net_device 实例中双链表：

●void **napi_hash_add**(struct napi_struct *napi)：将实例添加到全局散列表。

●void **netif_napi_add**(struct net_device *dev, struct napi_struct *napi,int (***poll**)(struct napi_struct *, int), int weight)：初始化 napi_struct 实例，并将其添加到 net_device 实例 napi_list 链表中。

3 接口函数

netif_rx(struct sk_buff *skb)接口函数用于传统方法中网络设备中断处理程序接收数据包，也可以在 NAPI 方法中 napi_struct 实例的 poll()函数中调用。netif_rx()函数将接收数据包 skb 添加到 softnet_data 实例接收数据包缓存队列。如果接收缓存队列先前为空，需要先将 softnet_data.**backlog** 成员添加到轮询表，触发接收软中断，然后再添加数据包到缓存队列。。

napi_schedule()/__napi_schedule()接口函数用于采用 NAPI 的网络设备，在中断处理程序中需要将网络设备定义的 napi_struct 实例添加到轮询表，并触发接收软中断。

■网络设备接收数据包

在传统接收方法中，网络设备中断处理程序构建 sk_buff 实例后，通常调用 **netif_rx()**函数将数据包添加到接收缓存队列，然后从中断返回。NAPI 方法中，napi_struct 实例中的 poll()函数内也可能调用 netif_rx()函数。

netif_rx()函数定义如下（/net/core/dev.c）：

```
int netif_rx(struct sk_buff *skb)
{
    trace_netif_rx_entry(skb);

    return netif_rx_internal(skb);      /*/net/core/dev.c*/
}
```

netif_rx_internal()函数定义如下（/net/core/dev.c）：

```
static int netif_rx_internal(struct sk_buff *skb)
{
    int ret;

    net_timestamp_check(netdev_tstamp_prequeue, skb);

    trace_netif_rx(skb);
#ifdef CONFIG_RPS
    if (static_key_false(&rps_needed)) {
        struct rps_dev_flow voidflow, *rflow = &voidflow;
        int cpu;
```

```
            preempt_disable();
            rcu_read_lock();

            cpu = get_rps_cpu(skb->dev, skb, &rflow);
            if (cpu < 0)
                cpu = smp_processor_id();

            ret = enqueue_to_backlog(skb, cpu, &rflow->last_qtail);

            rcu_read_unlock();
            preempt_enable();
        } else
#endif
        {
            unsigned int qtail;
            ret = enqueue_to_backlog(skb, get_cpu(), &qtail);     /*将数据包添加到接收缓存队列*/
            put_cpu();
        }
        return ret;
}
```

netif_rx_internal()函数调用 **enqueue_to_backlog**()函数，将数据包添加到接收缓存队列，若接收缓存队列先前为空，还需先将 softnet_data.backlog 成员（napi_struct 实例）添加到轮询表，触发接收软中断，然后再添加数据包到缓存队列。如果接收队列数据包数量超标了，则直接将数据包丢弃。

enqueue_to_backlog()函数定义如下（/net/core/dev.c）：

```
static int enqueue_to_backlog(struct sk_buff *skb, int cpu,unsigned int *qtail)
{
    struct softnet_data *sd;
    unsigned long flags;
    unsigned int qlen;

    sd = &per_cpu(softnet_data, cpu);     /*softnet_data 实例*/
    local_irq_save(flags);

    rps_lock(sd);
    if (!netif_running(skb->dev))     /*网络设备是否已启用*/
        goto drop;
    qlen = skb_queue_len(&sd->input_pkt_queue);   /*接收数据包缓存队列长度*/
    if (qlen <= netdev_max_backlog && !skb_flow_limit(skb, qlen)) {   /*接收数据包队列未超标*/
        if (qlen) {       /*接收数据包队列不为空*/
enqueue:
            __skb_queue_tail(&sd->input_pkt_queue, skb);     /*将 sk_buff 添加到接收缓存队列*/
            input_queue_tail_incr_save(sd, qtail);
            rps_unlock(sd);
            local_irq_restore(flags);
            return NET_RX_SUCCESS;
```

```
            }

            /*接收数据包缓存队列为空时*/
            if (!__test_and_set_bit(NAPI_STATE_SCHED, &sd->backlog.state)) {
                if (!rps_ipi_queued(sd))
                    ____napi_schedule(sd, &sd->backlog);
                                        /*将 sd->backlog 实例添加到轮询表末尾，触发接收软中断*/
            }
            goto enqueue;    /*跳回至 enqueue 处，添加数据包到缓存队列*/
        }

    drop:    /*接收队列超标了，将数据包丢弃*/
        sd->dropped++;
        rps_unlock(sd);

        local_irq_restore(flags);

        atomic_long_inc(&skb->dev->rx_dropped);
        kfree_skb(skb);
        return NET_RX_DROP;
    }
```

■添加到轮询表

向轮询表添加 napi_struct 实例的接口函数为 napi_schedule()/__napi_schedule()，napi_schedule()函数定义如下（/include/linux/netdevice.h）：

```
    static inline void napi_schedule(struct napi_struct *n)
    {
        if (napi_schedule_prep(n))    /*检测 napi_struct 是否可接受调度，/include/linux/netdevice.h*/
            __napi_schedule(n);
    }

    __napi_schedule(n)函数定义如下（/net/core/dev.c）：
    void __napi_schedule(struct napi_struct *n)
    {
        unsigned long flags;

        local_irq_save(flags);
        ____napi_schedule(this_cpu_ptr(&softnet_data), n);    /*/net/core/dev.c*/
        local_irq_restore(flags);
    }

    ____napi_schedule()函数定义如下（/net/core/dev.c）：
    static inline void ____napi_schedule(struct softnet_data *sd,struct napi_struct *napi)
    {
        list_add_tail(&napi->poll_list, &sd->poll_list);    /*将实例添加到轮询表末尾*/
```

```
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);      /*触发接收软中断*/
}
```

4 接收数据包软中断

NET_RX_SOFTIRQ 软中断处理函数 net_rx_action()将轮询 softnet_data 实例中 napi_struct 实例链表（轮询表），对每个实例调用其 **poll()**函数。

在网络设备初始化函数 net_dev_init()中注册了发送、接收数据包软中断，其中 NET_RX_SOFTIRQ 软中断处理函数为 net_rx_action()。

net_rx_action()函数代码如下（/net/core/dev.c）：

```
static void net_rx_action(struct softirq_action *h)
{
    struct softnet_data *sd = this_cpu_ptr(&softnet_data);   /*softnet_data 实例*/
    unsigned long time_limit = jiffies + 2;
    int budget = netdev_budget;     /*配额，接收数据包数量，默认值为 300*/
    LIST_HEAD(list);     /*缓存轮询表*/
    LIST_HEAD(repoll);   /*缓存需要重新插入 sd->poll_list 链表的实例*/

    local_irq_disable();
    list_splice_init(&sd->poll_list, &list);     /*将 sd->poll_list 链表成员移入 list 链表*/
    local_irq_enable();

    for (;;) {     /*遍历 list 链表中 napi_struct 实例*/
        struct napi_struct *n;

        if (list_empty(&list)) {      /*轮询表为空，跳出循环*/
            if (!sd_has_rps_ipi_waiting(sd) && list_empty(&repoll))
                return;
            break;
        }

        n = list_first_entry(&list, struct napi_struct, poll_list);     /*从轮询表取出 napi_struct 实例*/
        budget -= napi_poll(n, &repoll);     /*调用 poll()函数接收数据包，返回接收数据包数量*/

        if (unlikely(budget <= 0 ||time_after_eq(jiffies, time_limit))) {     /*超过配额或超时，退出*/
            sd->time_squeeze++;
            break;
        }
    }   /*遍历 napi_struct 实例链表结束*/

    local_irq_disable();

    list_splice_tail_init(&sd->poll_list, &list);   /*将 sd->poll_list 链表成员移到 list 链表末尾*/
    list_splice_tail(&repoll, &list);       /*将 repoll 链表中成员移到 list 链表末尾*/
    list_splice(&list, &sd->poll_list);     /*将 list 链表中成员移到 sd->poll_list 链表头部*/
```

```
        if (!list_empty(&sd->poll_list))   /*如果 sd->poll_list 链表不为空，继续触发接收软中断*/
            __raise_softirq_irqoff(NET_RX_SOFTIRQ);

        net_rps_action_and_irq_enable(sd);
    }
```

net_rx_action()函数内定义了两个临时双链表，list 和 repoll，在轮询开始前将 sd->poll_list 链表中全部成员移动到 list 链表。轮询操作从 list 链表中取出 napi_struct 实例，对其调用 napi_poll()函数，执行完此函数后如果 napi_struct 实例还需要添加到 sd->poll_list 链表，则将其添加到参数指定的 repoll 临时链表。在本次轮询结束后会将 repoll 临时链表中实例放回到 sd->poll_list 链表头部。

napi_poll()函数用于执行 napi_struct 实例中的 poll()函数，函数定义如下（/net/core/dev.c）：

```
    static int napi_poll(struct napi_struct *n, struct list_head *repoll)
    {
        void *have;
        int work, weight;

        list_del_init(&n->poll_list);     /*将 napi_struct 从链表中移除*/
        have = netpoll_poll_lock(n);

        weight = n->weight;     /*接收数据包数量配额*/

        work = 0;
        if (test_bit(NAPI_STATE_SCHED, &n->state)) {    /*napi_struct 实例可接受调度*/
            work = n->poll(n, weight);       /*调用 poll()函数，返回接收数据包数量*/
            trace_napi_poll(n);
        }

        WARN_ON_ONCE(work > weight);

        if (likely(work < weight))       /*没有超过配额，函数返回*/
            goto out_unlock;

        /*以下是接收数据包超过配额的情形*/
        if (unlikely(napi_disable_pending(n))) {     /*是否设置了 NAPI_STATE_DISABLE 状态*/
            napi_complete(n);     /*处理 n->gro_list 中成员，/include/linux/netdevice.h*/
            goto out_unlock;
        }

        /*如果没有设置 NAPI_STATE_DISABLE 状态*/
        if (n->gro_list) {
            napi_gro_flush(n, HZ >= 1000);
        }
        ...
        list_add_tail(&n->poll_list, repoll);     /*将 napi_struct 实例添加到 repoll 指向双链表末尾*/

    out_unlock:
```

```
    netpoll_poll_unlock(have);
    return work;
}
```

如果在 napi_poll()函数中接收的数据包超过了（或等于）配额，说明还有数据包没有接收完，napi_struct 实例将添加到 repoll 指向双链表末尾，在此轮询结束后将放回 sd->poll_list 链表，继续接受轮询。

5 向网络层传递数据包

在接口函数 netif_rx()中，若接收缓存队列为空，则将 softnet_data.backlog 实例添加到轮询表，其 poll()函数为 **process_backlog()**，此函数先将接收缓队列中数据包移动到接收队列，然后将接收队列中数据包传递到网络层。

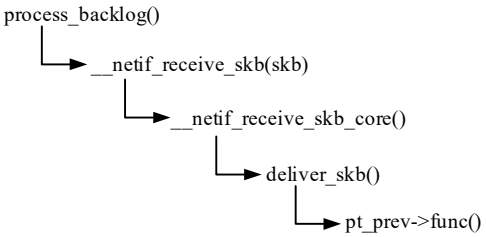
process_backlog()函数定义如下（/net/core/dev.c）：

```
static int process_backlog(struct napi_struct *napi, int quota)
/*quota：配额，接收了多于配额的数据包后，函数返回*/
{
    int work = 0;     /*接收数据包数量*/
    struct softnet_data *sd = container_of(napi, struct softnet_data, backlog);   /*softnet_data 实例*/

    if (sd_has_rps_ipi_waiting(sd)) {
        local_irq_disable();
        net_rps_action_and_irq_enable(sd);
    }

    napi->weight = weight_p;     /*配额*/
    local_irq_disable();       /*关中断*/
    while (1) {       /*无限循环*/
        struct sk_buff *skb;

        while ((skb = __skb_dequeue(&sd->process_queue))) {   /*从接收队列取出 sk_buff 实例*/
            rcu_read_lock();
            local_irq_enable();
            __netif_receive_skb(skb);     /*接收数据包，传递到网络层*/
            rcu_read_unlock();
            local_irq_disable();
            input_queue_head_incr(sd);
            if (++work >= quota) {       /*达到配额，函数返回接收数据包数量*/
                local_irq_enable();
                return work;
            }
        }
        /*接收队列为空，从接收缓存队列中移入数据包*/
        rps_lock(sd);
        if (skb_queue_empty(&sd->input_pkt_queue)) {     /*接收缓存队列为空*/
            napi->state = 0;
            rps_unlock(sd);
            break;
```

```
            }
            skb_queue_splice_tail_init(&sd->input_pkt_queue,&sd->process_queue);
                                                    /*将接收缓存队列中数据包移入到接收队列*/
            rps_unlock(sd);
        }    /*无限循环结束*/
        local_irq_enable();

        return work;        /*返回接收数据包数量*/
    }
```

process_backlog()函数内是一个无限循环。网络设备接收的数据包将添加到 **sd->input_pkt_queue** 队列，process_backlog()函数将此队列中的数据包移动到 sd->process_queue 接收队列，准备接收。process_backlog()函数从接收队列取出数据包，调用**__netif_receive_skb(skb)**函数将数据包传递到网络层。如果接收数据包数量超过了配额或接收队列清空了，process_backlog()函数将返回，返回值是接收数据包的数量。

__netif_receive_skb(skb)函数用于将数据包传递到网络层，函数内将调用__netif_receive_skb_core()函数遍历内核中注册的 packet_type 实例，对每个实例调用 deliver_skb()函数。deliver_skb()函数调用 packet_type 实例中的 func()函数接收数据包。函数调用关系如下图所示：



__netif_receive_skb(skb)函数定义如下（/net/core/dev.c）：

```
static int __netif_receive_skb(struct sk_buff *skb)
{
    int ret;

    if (sk_memalloc_socks() && skb_pfmemalloc(skb)) {
        unsigned long pflags = current->flags;
        current->flags |= PF_MEMALLOC;
        ret = __netif_receive_skb_core(skb, true);
        tsk_restore_flags(current, pflags, PF_MEMALLOC);
    } else
        ret = __netif_receive_skb_core(skb, false);

    return ret;
}
```

__netif_receive_skb_core()函数定义如下，注意函数内需要遍历所有注册的匹配 packet_type 实例：

```
static int __netif_receive_skb_core(struct sk_buff *skb, bool pfmemalloc)
{
    struct packet_type *ptype, *pt_prev;
    rx_handler_func_t *rx_handler;
    struct net_device *orig_dev;
```

```
    bool deliver_exact = false;
    int ret = NET_RX_DROP;     /*返回结果*/
    __be16 type;

    net_timestamp_check(!netdev_tstamp_prequeue, skb);

    trace_netif_receive_skb(skb);

    orig_dev = skb->dev;     /*网络设备*/

    skb_reset_network_header(skb);
    if (!skb_transport_header_was_set(skb))
        skb_reset_transport_header(skb);
    skb_reset_mac_len(skb);

    pt_prev = NULL;

another_round:
    skb->skb_iif = skb->dev->ifindex;     /*网络设备索引值*/

    __this_cpu_inc(softnet_data.processed);      /*增加处理数据包数量*/
    ...
    if (pfmemalloc)
        goto skip_taps;

    list_for_each_entry_rcu(ptype, &ptype_all, list) {     /*遍历适用于所有协议的 packet_type 实例*/
        if (pt_prev)
            ret = deliver_skb(skb, pt_prev, orig_dev);
        pt_prev = ptype;
    }

    list_for_each_entry_rcu(ptype, &skb->dev->ptype_all, list) {
                                /*遍历绑定到网络设备适用所有协议的 packet_type 实例*/
        if (pt_prev)
            ret = deliver_skb(skb, pt_prev, orig_dev);
        pt_prev = ptype;
    }

skip_taps:
#ifdef CONFIG_NET_INGRESS
    if (static_key_false(&ingress_needed)) {
        skb = handle_ing(skb, &pt_prev, &ret, orig_dev);
        if (!skb)
            goto out;

        if (nf_ingress(skb, &pt_prev, &ret, orig_dev) < 0)
```

```
                goto out;
        }
#endif
        ...
        if (pfmemalloc && !skb_pfmemalloc_protocol(skb))
                goto drop;

        if (skb_vlan_tag_present(skb)) {
                if (pt_prev) {
                        ret = deliver_skb(skb, pt_prev, orig_dev);
                        pt_prev = NULL;
                }
                if (vlan_do_receive(&skb))
                        goto another_round;
                else if (unlikely(!skb))
                        goto out;
        }

        rx_handler = rcu_dereference(skb->dev->rx_handler);     /*网络设备定义的处理函数*/
        if (rx_handler) {
                if (pt_prev) {
                        ret = deliver_skb(skb, pt_prev, orig_dev);
                        pt_prev = NULL;
                }
                switch (rx_handler(&skb)) {     /*调用网络设备定义的处理函数*/
                case RX_HANDLER_CONSUMED:
                        ret = NET_RX_SUCCESS;         /*由网络设备成功处理*/
                        goto out;
                case RX_HANDLER_ANOTHER:
                        goto another_round;
                case RX_HANDLER_EXACT:     /*正常往下执行*/
                        deliver_exact = true;
                case RX_HANDLER_PASS:
                        break;
                default:
                        BUG();
                }
        }     /*if(rx_handler)结束*/

        if (unlikely(skb_vlan_tag_present(skb))) {
                if (skb_vlan_tag_get_id(skb))
                        skb->pkt_type = PACKET_OTHERHOST;
                skb->vlan_tci = 0;
        }

        type = skb->protocol;    /*网络层协议类型*/
```

```
    /*在 ptype_base[]散列表中查找协议类型匹配的 packet_type 实例，并调用 deliver_skb()函数*/
    if (likely(!deliver_exact)) {
        deliver_ptype_list_skb(skb, &pt_prev, orig_dev, type,
                               &ptype_base[ntohs(type) &PTYPE_HASH_MASK]);
    }

    /*在 net_device->ptype_specific 链表中查找协议类型匹配的 packet_type 实例，
    *并调用 deliver_skb()函数*/
    deliver_ptype_list_skb(skb, &pt_prev, orig_dev, type,&orig_dev->ptype_specific);

    if (unlikely(skb->dev != orig_dev)) {
        deliver_ptype_list_skb(skb, &pt_prev, orig_dev, type,&skb->dev->ptype_specific);
    }

    if (pt_prev) {   /*前面遍历的最后一个 packet_type 实例*/
        if (unlikely(skb_orphan_frags(skb, GFP_ATOMIC)))
            goto drop;
        else
            ret = pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
    } else {
drop:
        atomic_long_inc(&skb->dev->rx_dropped);
        kfree_skb(skb);
        ret = NET_RX_DROP;
    }

out:
    return ret;
}
```

由上面的分析可知，函数遍历内核中所有注册的 packet_type 实例顺序如下：

- 适用于所有网络层协议且未绑定到网络设备的实例。
- 适用于所有网络层协议且绑定到网络设备的实例。
- 网络设备定义的 **rx_handler()**函数。
- 适用于特定网络层协议且未绑定到网络设备的实例。
- 适用于特定网络层协议且绑定到网络设备的实例。

函数遍历 packet_type 实例，对每个实例调用 deliver_skb()函数，定义如下（/net/core/dev.c）：

```
static inline int deliver_skb(struct sk_buff *skb,struct packet_type *pt_prev,struct net_device *orig_dev)
{
    if (unlikely(skb_orphan_frags(skb, GFP_ATOMIC)))
        return -ENOMEM;
    atomic_inc(&skb->users);
    return pt_prev->func(skb, skb->dev, pt_prev, orig_dev);     /*调用 packet_type->func()函数*/
}
```

deliver_skb()函数将调用 packet_type->func()函数接收数据包，这是由网络层协议定义的接收函数。例

如，IPv4 协议定义的接收数据包函数为 ip_rcv()，这是数据包进入 IPv4 网络层的入口函数。

13.2.5 以太网与设备驱动

数据链路层在网络通信中实现相邻节点之间的数据传输，即实现数据的接力传输。数据链路层数据包被称为帧。在链路层中存在两种截然不同的链路层信道。第一种是广播信道，这种信道用于连接有线局域网、卫星网和混合光纤同轴电缆接入网中的多台主机。因为许多主机与相同的广播信道连接，需要所谓的媒体访问协议（MAC）来协调帧传输。第二种链路层信道是点对点通信链路，协调点对点链路的访问较为简单。

在因特网传输路径中，路由器与路由器之间多数是通过有线点对点通信链路相连。主机（端系统）通常与一局域网（LAN）相连，通过局域网中心节点与因特网路由器相连，从而连接上因特网。局域网就是在本地将多个主机相互连接起来，再通过统一的上行端口与因特网相连。局域网内主机之间可能是通过广播信道通信，也可能是通过点对点通信链路通信。

局域网是实现主机之间在链路层通信的技术。在局域网中每台主机（网络适配器）由一个唯一的物理地址表示（MAC 地址）。局域网可分为有线局域网、无线局域网等。以太网是目前最为流行的有线局域网技术，无线局域网典型的有 WiFi，它可以认为是无线以太网技术。

每种局域网技术有一个相关的协议（链路层协议），描述由网络适配器传输帧的格式和方法，此链路层协议主要由硬件实现（网络适配器）。

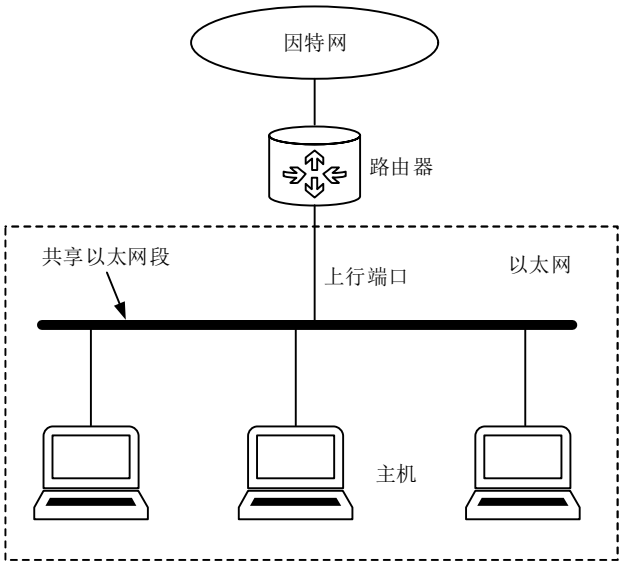
本小节简要介绍以太网结构和规范，并举例简要说明以太网网络适配器（网卡）驱动程序的实现。

1 以太网简介

以太网是目前最流行的有线局域网技术。以太网网络适配器（网卡、网络设备）通过一个 48bit（6 字节）的 MAC 地址标识。链路上传输的帧报头中包含源主机 MAC 地址和目的主机 MAC 地址，目的主机通过目的 MAC 地址确定帧是否是传递给本主机。

■以太网结构

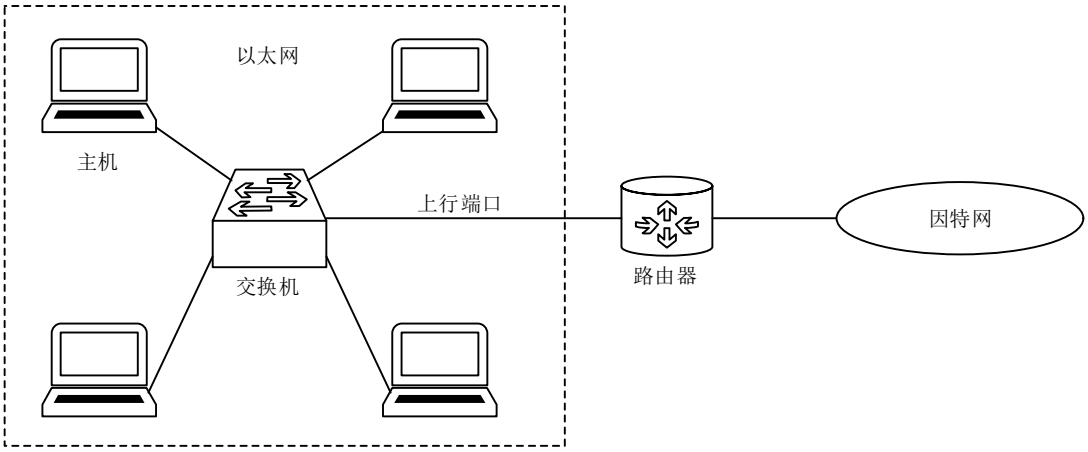
以太网这个术语通常指一套标准，由 DEC、Intel 公司和 Xerox 公司在 1980 年首次发布，并在 1982 年加以修订。第一个常见格式的以太网，目前称为“10Mb/s 以太网”或“共享以太网”，它被 IEEE 采纳为 802.3 标准。这种网络的结构通常如下图所示：



基本的共享以太网包含一个或多个主机，它们都被连接到一个共享的电缆段上。当介质（电缆）被确定为空闲状态时，链路层的 PDU（帧）可从一个主机发送到一个或多个其它主机。如果多个主机同时发送

数据，可能因信号传播延迟而发生碰撞。碰撞可以被检测到，它会导致发送主机等待一个随机时间，然后重新发送 PDU，这种常见的方法称为带冲突检测的载波侦听多路访问（CSMA/CD）。它协调哪些计算机可访问共享的介质，同时不需要其它特殊协议或同步。这种相对简单的方法有助于降低成本和促进以太网技术普及。采用 CSMA/CD，在任何给定的时间内，网络中只能有一个帧传输。如 CSMA/CD 这样的访问方法更正式的名称为介质访问控制（MAC）协议。

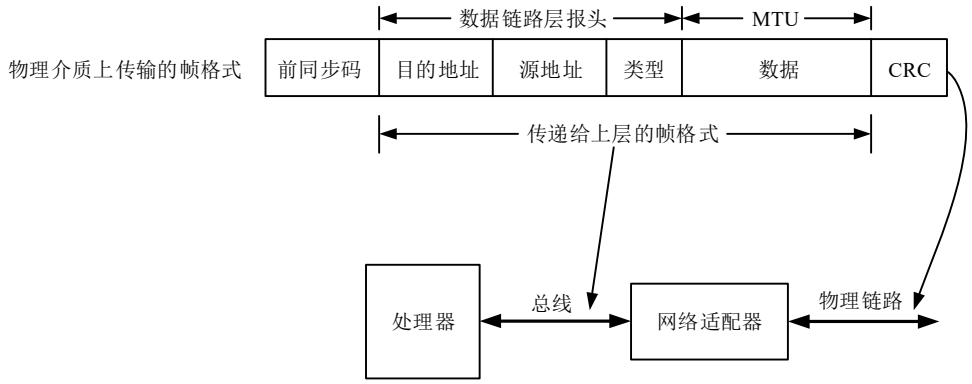
随着 100Mb/s 以太网（也称为“快速以太网”）的发展，基于竞争的 MAC 协议已经不再流行。相反，局域网中每个主机之间的线路通常不共享，而是提供了一个专用的星形拓扑结构。这可以通过一个以太网交换机来实现，如下图所示：



一个交换式以太网包含一个或多个主机，每个主机使用一条专用的线路连接到一个交换机端口。在大多数情况下，交换式以太网以全双工方式运行，并且不需要使用 CSMA/CD 算法。交换机上有一个上行端口与路由器相连，实现与因特网的连接。

交换机中管理一个交换机表，表项中通过 MAC 地址标识局域网内主机，表项中还指示指定主机与交换机哪个端口相连。交换机在收到链路层帧后通过目的 MAC 地址确定目的主机的端口，由此端口转发帧。交换机可通过自学习填充交换机表。对于外发的数据包则通过上行端口发送到因特网。

所有以太网帧都基于一个共同的格式。在原有的基础上，帧格式已被改进以支持额外功能。以太网帧基础格式如下图所示：



需要注意的是链路层帧是通过网络适配器按照链路层协议加工后，在物理链路上传输的数据格式。网络适配器在将帧传递给处理器前，会去掉特定于链路层协议的字段，而在发送数据包时会加上这些字段。上图中的前同步码、CRC 字段都是特定于链路层协议的字段，传递给处理器时，这些字段会去掉。帧格式通常支持可变的帧长度，范围从几字节到几千字节。这个范围的上限称为最大传输单元（MTU）。

以太网网络适配器与处理器之间传输的帧中主要包含以下字段：

●**目的地址（6 字节）**：这个字段包含目的网络适配器（接口）的 MAC 地址，如 62-FE-F7-11-89-A3。当适配器收到一个以太网帧后，如果帧的目的地址为 62-FE-F7-11-89-A3 或广播地址（FF-FF-FF-FF-FF-FF），适配器会将该帧传递给主机，否则丢弃。

●**源目的地址（6 字节）**：源网络适配器 MAC 地址。

●**类型（2 字节）**：类型字段允许以太网复用多种网络层协议，此字段表示数据字段适用的网络层协议。也就是说，以太网（数据链路层协议）可与多种网络层协议组合使用。

●**数据**：承载网络层数据包。以太网最大传输单元（MTU）典型值为 1500 字节，这意味着如果 IP 数据包超过了 1500 字节就需要将数据包分段。数据字段最小长度为 46 字节，如果 IP 数据包小于 46 字节则需要对其进行填充。

■以太网协议实现

在 Linux 内核中，以太网报头（数据链路层报头）由 ethhdr 结构体表示，结构体定义如下：

```
struct ethhdr {       /*/include/uapi/linux/if_ether.h*/
    unsigned char h_dest[ETH_ALEN];    /*目的主机 MAC 地址*/
    unsigned char h_source[ETH_ALEN];  /*源主机 MAC 地址*/
    __be16        h_proto;        /*网络层协议类型*/
} __attribute__((packed));
```

报头中包含 6 字节的目的 MAC 地址，6 字节的源 MAC 地址，以及 2 字节的网络层协议类型。类型值 h_proto 取值定义在头文件/include/uapi/linux/if_ether.h，例如：

```
#define   ETH_P_LOOP     0x0060        /*环回设备数据包*/
#define   ETH_P_PUP      0x0200        /* Xerox PUP packet*/
#define   ETH_P_PUPAT    0x0201        /* Xerox PUP Addr Trans packet  */
#define   ETH_P_IP       0x0800        /*IPv4 数据包*/
#define   ETH_P_X25      0x0805        /* CCITT X.25*/
#define   ETH_P_ARP      0x0806        /*ARP 数据包*/
...
#define   ETH_P_IPV6     0x86DD        /* IPv6 数据包*/
...
```

以太网协议公共代码在/net/ethernet/eth.c 文件内实现。在分配 net_device 实例的 alloc_netdev()函数中第三个参数是一个函数指针，用于设置 net_device 实例。通常调用数据链路层协议定义的函数。在以太网协议中此函数为 ether_setup()，代码如下：

```
void ether_setup(struct net_device *dev)
{
    dev->header_ops     = &eth_header_ops;      /*设置数据链路层报头操作结构实例*/
    dev->type           = ARPHRD_ETHER;     /*网络设备类型，以太网设备*/
    dev->hard_header_len    = ETH_HLEN;      /*以太网报头长度（14 字节）*/
    dev->mtu        = ETH_DATA_LEN;         /*MTU 长度，1500*/
    dev->addr_len       = ETH_ALEN;        /*物理地址长度，6 字节*/
    dev->tx_queue_len       = 1000;         /*发送队列长度（字节）*/
    dev->flags          = IFF_BROADCAST|IFF_MULTICAST;     /*设备标志*/
    dev->priv_flags     |= IFF_TX_SKB_SHARING;          /*私有标志*/

    eth_broadcast_addr(dev->broadcast);      /*设置广播地址，FF.FF.FF.FF.FF.FF*/
}
```

以太网报头操作结构 eth_header_ops 实例定义如下：

```
const struct header_ops eth_header_ops ____cacheline_aligned = {
    .create          = eth_header,          /*生成数据链路层报头，写入数据包报头中*/
    .parse           = eth_header_parse,    /*从报头中抽取源主机物理地址*/
    .cache           = eth_header_cache,    /*将 L2 层报头写入邻居实例*/
    .cache_update = eth_header_cache_update,    /*更新邻居实例中 L2 层报头缓存*/
};
```

以上函数实现都比较简单，请读者行阅读源代码。

在/net/ethernet/eth.c 文件内还定义其它接口函数，例如：

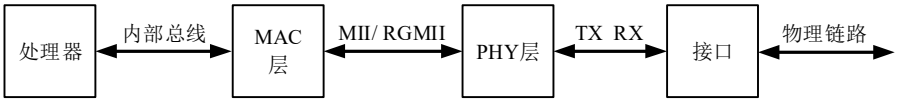
- __be16 **eth_type_trans**(struct sk_buff *skb, struct net_device *dev)：返回报头中协议类型字段。
- int **eth_change_mtu**(struct net_device *dev, int new_mtu)：设置新 MTU 值。
- int **eth_mac_addr**(struct net_device *dev, void *p)：设置网络设备新 MAC 地址。

2 驱动示例

网络适配器及其驱动程序将实现数据链路层协议，本节以以太网网络适配器为例，说明网络适配器物理结构和驱动程序实现。

■网络适配器

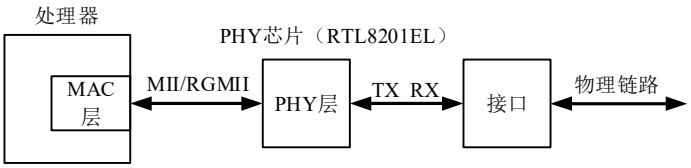
从硬件的角度看，网络适配器通常包括处理器、MAC 层、PHY 层、接口四个层次，如下图所示：



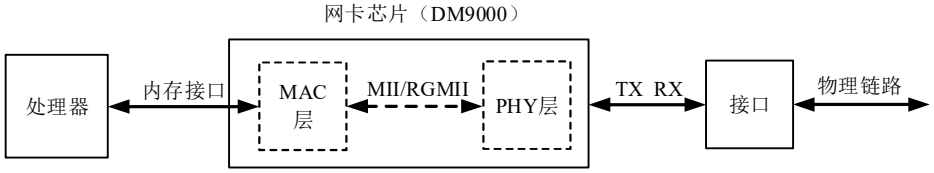
MAC 层即媒体接入控制器，属于网络模型的数据链路层，对后面 PHY 层（物理层）的接收和发送数据行为进行控制，相当于一个总线的主机控制器。PHY 层即物理层，是真正实现发送和接收数据的地方，但它不会对数据进行任何区分和处理。当发现网络上有数据时，就把数据取出上传给 MAC 层，当 MAC 层上有数据时就将它放到网络上。PHY 层通常由一个专用芯片实现。MAC 层需要对数据进行区分，如果是发送给本主机的就将其提交到上层协议，否则将其丢弃。MAC 层在发送数据时会在数据包上附加一些额外的信息，并发送给 PHY 层，接收时将额外的信息去掉。

MAC 层与 PHY 层的接口是标准的，如百兆模式（MII）、千兆模式（RGMII）。不管什么样的物理网络，MAC 层都可以对 PHY 层进行控制，且一个 MAC 可控制多个 PHY 层。MAC 控制器也有一组标准的寄存器，处理器只需要对寄存器进行操作即可。

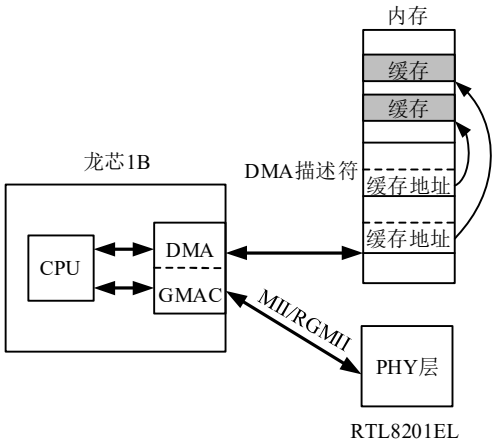
有的处理器集成了 MAC 控制器，只需要添加一个 PHY 芯片即可，如下图所示：



有的处理器没有 MAC 控制器，一般采用带 MAC 和 PHY 的集成网卡芯片实现网络收发，如下图所示，，这两种组合都是比较常见的情况。



龙芯 1B 处理器属于自带 MAC 控制器的情形，只需外接 PHY 芯片即可实现网络收发。如下图所示，龙芯 1B 内嵌两个 GMAC（千兆网媒体控制器），图中只画出一个，龙芯 1B 开发板外接 RTL8201EL 芯片，用于实现 PHY 层。



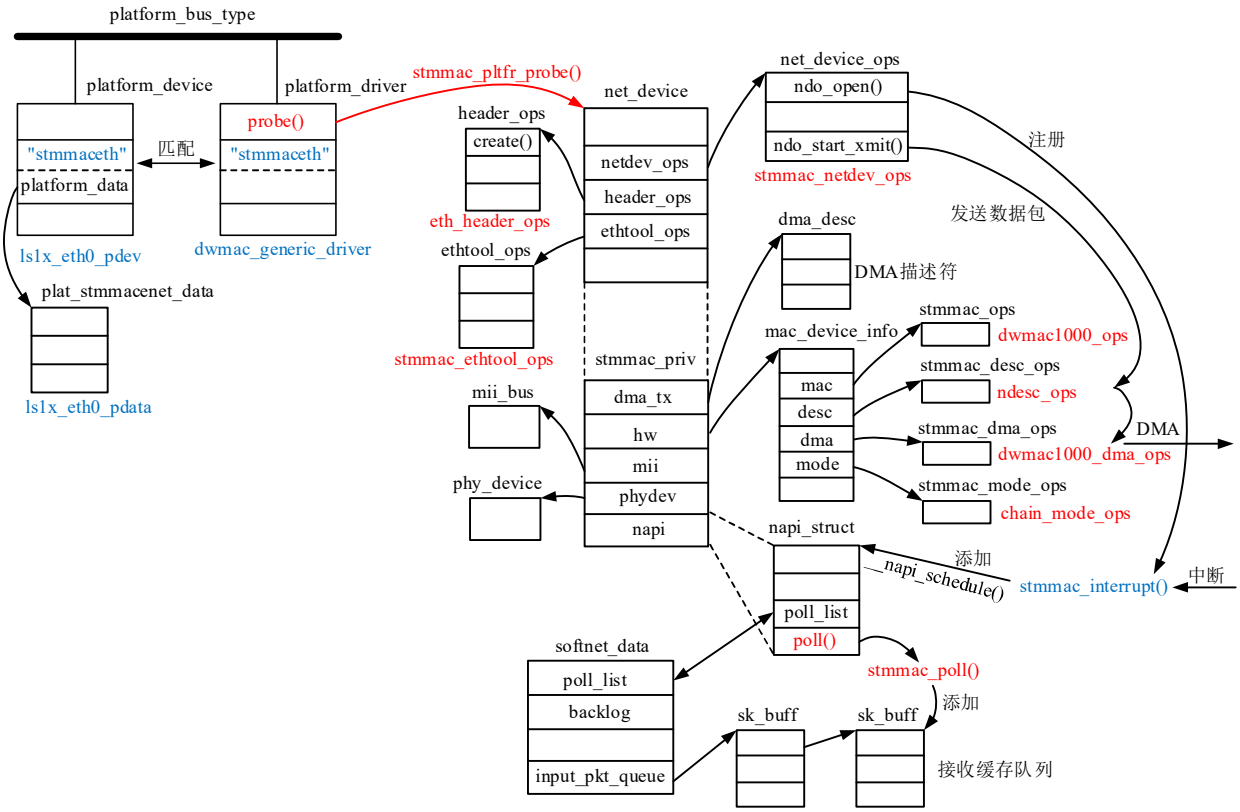
GMAC 控制器中还包含专用的 DMA 控制器，DMA 控制器关联 DMA 描述符，描述符分为接收描述符和发送描述符。描述符中包含缓存基地址，数据发送、接收状态等信息。GMAC 中包含 FIFO 数据缓存区，用户启动 GMAC 后，DMA 会自动将内存缓存区数据发送到 GMAC 缓存区，由 GMAC 自动将数据发送到 PHY。GMAC 接收到数据后，DMA 从 GMAC 缓存中读取数据至内存缓存。

DMA 和 GMAC 都具有控制寄存器，处理器通过寄存器控制其行为。GMAC 实现对 PHY 的控制，用户可通过 GMAC 实现对 PHY 的控制。

■驱动框架

内核以太网网络适配器驱动程序代码位于/drivers/net/ethernet/目录下。龙芯 1B 开发板网络适配器驱动程序复用 STM 处理器网络适配器驱动程序，驱动程序代码位于/drivers/net/ethernet/stmicro/stmmac/目录下。

STM 处理器网络适配器可通过处理器内部总线与 CPU 相连（平台 platform 总线），或通过 PCI 总线与 CPU 相连。龙芯 1B 需要选择配置使用平台总线，即选择 STMMAC_PLATFORM 配置选项。平台总线网络适配器驱动程序框架如下图所示：



GMAC 控制器及其驱动挂载到平台总线上，平台设备通过 plat_stmmacenet_data 结构体向驱动程序传递 GMAC 控制器的设备信息。控制器驱动 platform_driver 实例的 probe()函数为 **stmmac_pltfr_probe()**，此函数内将创建、设置、注册 net_device 实例，创建并设置私有数据结构 stmmac_priv 结构体实例。

网络设备关联的网络设备操作结构 net_device_ops 实例为 stmmac_netdev_ops，数据链路层报头操作结构为以太网 eth_header_ops 实例。

stmmac_pltfr_probe()函数内还将设置 stmmac_priv 实例内嵌的 napi_struct 实例，以采用 NAPI 方法接收数据帧，napi_struct 实例中 poll()函数为 stmmac_poll()，负责接收数据帧。

net_device_ops 结构体 stmmac_netdev_ops 实例在打开函数 ndo_open()中将初始化硬件，注册中断处理函数，中断处理函数中将 napi_struct 实例添加到轮询表，并触发接收软中断。stmmac_netdev_ops 实例中 ndo_start_xmit()函数指针为 stmmac_xmit()，此函数用于发送数据帧。

在涉及到具体的操作中，主要是由 mac_device_info 结构体中关联的三个结构体中的函数完成。其中，stmmac_ops 结构体表示对 MAC 控制器的操作，stmmac_desc_ops 结构体表示 DMA 描述符的操作，结构体 stmmac_dma_ops 表示 DMA 操作，stmmac_mode_ops 表示 DMA 描述符列表操作。

发送数据包 stmmac_xmit()函数中将调用 stmmac_desc_ops 结构体中的函数设置 DMA 描述符（包括将数据包写入缓存），然后调用 stmmac_dma_ops 结构体中函数启动 DMA，将数据传递给 GMAC，然后由硬件将数据发送到物理链路上。

下面先简要介绍一下 STM 网络适配器驱动程序中使用的数据结构。

●设备私有数据结构

plat_stmmacenet_data 结构体用于板级（平台）代码向驱动程序传递设备信息，结构体定义如下：

```
struct plat_stmmacenet_data {              /*/include/linux/stmmac.h*/
    char *phy_bus_name;
    int bus_id;
    int phy_addr;     /*物理地址*/
    int interface;     /*接口类型，如 PHY_INTERFACE_MODE_MII，/include/linux/phy.h*/
    struct stmmac_mdio_bus_data *mdio_bus_data;     /*/include/linux/stmmac.h*/
```

```
    struct device_node *phy_node;        /*设备树，设备节点*/
    struct stmmac_dma_cfg *dma_cfg;     /*DMA 配置信息，/include/linux/stmmac.h*/
    int clk_csr;
    int has_gmac;     /*千兆以太网控制器*/
    int enh_desc;     /**/
    int tx_coe;
    int rx_coe;
    int bugged_jumbo;
    int pmt;
    int force_sf_dma_mode;
    int force_thresh_dma_mode;
    int riwt_off;
    int max_speed;
    int maxmtu;
    int multicast_filter_bins;
    int unicast_filter_entries;
    int tx_fifo_size;
    int rx_fifo_size;
    void (*fix_mac_speed)(void *priv, unsigned int speed);
    void (*bus_setup)(void __iomem *ioaddr);
    void *(*setup)(struct platform_device *pdev);
    void (*free)(struct platform_device *pdev, void *priv);
    int (*init)(struct platform_device *pdev, void *priv);       /*初始化函数*/
    void (*exit)(struct platform_device *pdev, void *priv);
    void *custom_cfg;
    void *custom_data;
    void *bsp_priv;
};
```

●驱动私有数据结构

网络适配器驱动程序私有数据结构为 stmmac_priv，它做为网络设备 net_device 实例的私有数据，在分配 net_device 实例时，一同分配结构体实例，附在 net_device 实例后面。

stmmac_priv 结构体定义如下（/drivers/net/ethernet/stmicro/stmmac/stmmac.h）：

```
struct stmmac_priv {
    struct dma_extended_desc *dma_etx ____cacheline_aligned_in_smp;
                                        /*/drivers/net/ethernet/stmicro/stmmac/descs.h*/
    struct dma_desc *dma_tx;      /*发送 DMA 描述符，/drivers/net/ethernet/stmicro/stmmac/descs.h*/
    struct sk_buff **tx_skbuff;
    unsigned int cur_tx;
    unsigned int dirty_tx;
    unsigned int dma_tx_size;
    u32 tx_count_frames;
    u32 tx_coal_frames;
    u32 tx_coal_timer;
    struct stmmac_tx_info *tx_skbuff_dma; /*发送信息，/drivers/net/ethernet/stmicro/stmmac/stmmac.h*/
    dma_addr_t dma_tx_phy;
```

```
int tx_coalesce;
int hwts_tx_en;
spinlock_t tx_lock;
bool tx_path_in_lpi_mode;
struct timer_list txtimer;

struct dma_desc *dma_rx ____cacheline_aligned_in_smp;   /*接收 DMA 描述符*/
struct dma_extended_desc *dma_erx;
struct sk_buff **rx_skbuff;
unsigned int cur_rx;
unsigned int dirty_rx;
unsigned int dma_rx_size;
unsigned int dma_buf_sz;
u32 rx_riwt;
int hwts_rx_en;
dma_addr_t *rx_skbuff_dma;
dma_addr_t dma_rx_phy;

struct napi_struct napi ____cacheline_aligned_in_smp;      /*napi_struct 实例*/

void __iomem *ioaddr;     /*控制寄存器基地址*/
struct net_device *dev;     /*指向网络设备 net_device 实例*/
struct device *device;      /*指向 platform_device 实例中 device 实例*/
struct mac_device_info *hw;      /*/drivers/net/ethernet/stmicro/stmmac/common.h*/
spinlock_t lock;

struct phy_device *phydev ____cacheline_aligned_in_smp;   /*表示物理设备，/include/linux/phy.h*/
int oldlink;
int speed;
int oldduplex;
unsigned int flow_ctrl;
unsigned int pause;
struct mii_bus *mii;      /*表示与 PHY 连接的总线，/include/linux/phy.h*/
int mii_irq[PHY_MAX_ADDR];

struct stmmac_extra_stats xstats ____cacheline_aligned_in_smp;
struct plat_stmmacenet_data *plat;     /*指向设备私有数据结构，与/include/linux/stmmac.h*/
struct dma_features dma_cap;
struct stmmac_counters mmc;
int hw_cap_support;
int synopsys_id;
u32 msg_enable;
int wolopts;
int wol_irq;
struct clk *stmmac_clk;
struct clk *pclk;
```

```
    struct reset_control *stmmac_rst;
    int clk_csr;
    struct timer_list eee_ctrl_timer;
    int lpi_irq;
    int eee_enabled;
    int eee_active;
    int tx_lpi_timer;
    int pcs;
    unsigned int mode;    /*描述符列表模式，链表或环形表*/
    int extend_desc;
    struct ptp_clock *ptp_clock;
    struct ptp_clock_info ptp_clock_ops;
    unsigned int default_addend;
    struct clk *clk_ptp_ref;
    unsigned int clk_ptp_rate;
    u32 adv_ts;
    int use_riwt;
    int irq_wake;
    spinlock_t ptp_lock;
    ...
};
```

stmmac_priv 结构体主要成员简介如下：

◎**napi**：napi_struct 结构体实例，添加到轮询表中。

◎**hw**：mac_device_info 结构体指针，结构体定义如下：

```
struct mac_device_info {
    const struct stmmac_ops *mac;    /*指向 MAC 操作结构实例*/
    const struct stmmac_desc_ops *desc;   /*指向描述符操作结构实例*/
    const struct stmmac_dma_ops *dma;   /*指向 DMA 操作结构实例*/
    const struct stmmac_mode_ops *mode;   /*DMA 描述符列表操作结构实例*/
    const struct stmmac_hwtimestamp *ptp;
    struct mii_regs mii;   /*MAC 地址*/
    struct mac_link link;
    ...
};
```

mac_device_info 结构体中指向的各操作结构定义在/drivers/net/ethernet/stmicro/stmmac/common.h 文件内，主是对 MAC 控制器，DMA 控制器，DMA 描述符的操作。

◎**phydev**：指向 phy_device 结构体，表示 PHY 芯片，结构体定义在/include/linux/phy.h 头文件。结构体内主要是一些函数指针，用于对 PHY 的配置控制等。

◎**mii**：指向 mii_bus 结构体，表示与 PHY 连接的总线，结构体定义在/include/linux/phy.h 头文件。

■平台代码

在板级（平台）文件内需要定义并注册表示 GMAC 控制器的 platform_device 实例，并通过结构体 plat_stmmacenet_data 实例向驱动程序传递板级设备信息。

龙芯 1B 平台代码在/arch/mips/loongson32/common/platform.c 文件内定义了表示 GMAC0 和 GMAC1 控制器的 platform_device 实例。下面列出表示 GMAC0 的 platform_device 实例：

```
static struct stmmac_mdio_bus_data ls1x_mdio_bus_data = {
        .phy_mask       = 0,
};

static struct stmmac_dma_cfg    ls1x_eth_dma_cfg = {
        .pbl        = 1,
};

static struct plat_stmmacenet_data    ls1x_eth0_pdata = {     /*plat_stmmacenet_data 结构体实例*/
        .bus_id         = 0,
        .phy_addr       = -1,
        .interface = PHY_INTERFACE_MODE_MII,     /*接口类型*/
        .mdio_bus_data      = &ls1x_mdio_bus_data,
        .dma_cfg = &ls1x_eth_dma_cfg,
        .has_gmac      = 1,         /*具有千兆网络适配器*/
        .tx_coe         = 1,
        .init        = ls1x_eth_mux_init,       /*设置处理器 GMAC 寄存器*/
};

static struct resource ls1x_eth0_resources[] = {       /*设备资源*/
        [0] = {
                .start= LS1X_GMAC0_BASE,                /*GMAC 控制寄存器起始地址*/
                .end = LS1X_GMAC0_BASE + SZ_64K - 1,         /*控制寄存器长度*/
                .flags= IORESOURCE_MEM,
        },
        [1] = {
                .name      = "macirq",                /*中断名称*/
                .start= LS1X_GMAC0_IRQ,     /*中断编号*/
                .flags      = IORESOURCE_IRQ,
        },
};

struct platform_device ls1x_eth0_pdev = {     /*platform_device 实例*/
        .name           = "stmmaceth",        /*名称，匹配驱动*/
        .id         = 0,
        .num_resources      = ARRAY_SIZE(ls1x_eth0_resources),
        .resource = ls1x_eth0_resources,       /*设备资源*/
        .dev        = {
                .platform_data = &ls1x_eth0_pdata,     /*设备私有数据结构*/
        },
};
```

在平台相关的初始化函数 ls1b_platform_init(void)中将注册 ls1x_eth0_pdev 实例。

■驱动代码简介

STM 微处理器网络适配器驱动程序代码在/drivers/net/ethernet/stmicro/stmmac/目录下。

●**探测函数**

龙芯 1B 网络适配器驱动需要选择 STMMAC_PLATFORM 和 DWMAC_GENERIC 配置选项，使用通用的平台总线驱动。驱动代码在 **dwmac-generic.c** 文件内定义了表示网络适配器驱动的 platform_driver 实例，如下图示：

```
static struct platform_driver dwmac_generic_driver = {
    .probe  = stmmac_pltfr_probe,      /*探测函数*/
    .remove = stmmac_pltfr_remove,
    .driver = {
        .name     = STMMAC_RESOURCE_NAME,    /*名称为"stmmaceth"，用于匹配设备*/
        .pm       = &stmmac_pltfr_pm_ops,
        .of_match_table = of_match_ptr(dwmac_generic_match),
    },
};
module_platform_driver(dwmac_generic_driver);    /*注册驱动*/
```

探测函数 **stmmac_pltfr_probe()**定义在 stmmac_platform.c 文件内，函数代码简列如下：

```
int stmmac_pltfr_probe(struct platform_device *pdev)
{
    struct stmmac_resources stmmac_res;
    int ret = 0;
    struct resource *res;
    struct device *dev = &pdev->dev;
    struct plat_stmmacenet_data *plat_dat = NULL;    /*plat_stmmacenet_data 指针*/

    memset(&stmmac_res, 0, sizeof(stmmac_res));

    stmmac_res.irq = platform_get_irq_byname(pdev, "macirq");      /*获取中断编号*/
    if (stmmac_res.irq < 0) {
        ...
    }
    ...      /*处理不同名称的中断号*/

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);    /*GMAC 控制寄存器基地址*/
    stmmac_res.addr = devm_ioremap_resource(dev, res);      /*控制寄存器基地址*/
    ...
    plat_dat = dev_get_platdata(&pdev->dev);   /*plat_stmmacenet_data 实例*/

    if (!plat_dat)
        plat_dat = devm_kzalloc(&pdev->dev,sizeof(struct plat_stmmacenet_data),GFP_KERNEL);
                                    /*platform_device 未指定 plat_stmmacenet_data 实例则分配*/

    plat_dat->multicast_filter_bins = HASH_TABLE_SIZE;

    plat_dat->unicast_filter_entries = 1;

    if (pdev->dev.of_node) {    /*使用了设备节点，从节点获取信息填充 plat_stmmacenet_data 实例*/
```

```
        ret = stmmac_probe_config_dt(pdev, plat_dat, &stmmac_res.mac);
        ...
    }

    if (plat_dat->setup) {
        plat_dat->bsp_priv = plat_dat->setup(pdev);     /*调用 setup()函数，如果定义了此函数*/
        ...
    }

    if (plat_dat->init) {       /*如果定义了初始化函数，则调用它*/
        ret = plat_dat->init(pdev, plat_dat->bsp_priv);
        ...
    }

    return stmmac_dvr_probe(&pdev->dev, plat_dat, &stmmac_res);
}
```

stmmac_pltfr_probe()函数通过 stmmac_resources 结构体向 stmmac_dvr_probe()函数传递 MAC 信息，结构体定义如下（/drivers/net/ethernet/stmicro/stmmac/stmmac.h）：

```
struct stmmac_resources {
    void __iomem *addr;     /*控制寄存器基地址*/
    const char *mac;        /*MAC 地址*/
    int wol_irq;
    int lpi_irq;
    int irq;        /*中断编号*/
};
```

stmmac_pltfr_probe()函数在获取控制器信息后，调用 stmmac_dvr_probe()函数创建、设置、注册网络设备 net_device 实例及私有数据结构 stmmac_priv 实例等。

stmmac_dvr_probe()函数定义如下（stmmac_main.c）：

```
int stmmac_dvr_probe(struct device *device,
                     struct plat_stmmacenet_data *plat_dat,struct stmmac_resources *res)
{
    int ret = 0;
    struct net_device *ndev = NULL;
    struct stmmac_priv *priv;

    ndev = alloc_etherdev(sizeof(struct stmmac_priv));
                                        /*分配以太网设备，/include/linux/etherdevice.h*/
    ...
    SET_NETDEV_DEV(ndev, device);
    priv = netdev_priv(ndev);     /*指向 stmmac_priv 实例*/
    priv->device = device;
    priv->dev = ndev;       /*指向 net_device 实例*/

    stmmac_set_ethtool_ops(ndev);    /*netdev->ethtool_ops = &stmmac_ethtool_ops*/
                                     /*stmmac_ethtool.c*/
```

```
    priv->pause = pause;
    priv->plat = plat_dat;
    priv->ioaddr = res->addr;       /*GMAC 寄存器基地址*/
    priv->dev->base_addr = (unsigned long)res->addr;

    priv->dev->irq = res->irq;     /*中断号*/
    priv->wol_irq = res->wol_irq;
    priv->lpi_irq = res->lpi_irq;

    if (res->mac)     /*如果指定了 MAC 地址*/
        memcpy(priv->dev->dev_addr, res->mac, ETH_ALEN);

    dev_set_drvdata(device, priv->dev);     /*驱动私有数据，指向 net_device 实例*/

    stmmac_verify_args();

    if ((phyaddr >= 0) && (phyaddr <= 31))     /*物理地址（模块参数）*/
        priv->plat->phy_addr = phyaddr;

    priv->stmmac_clk = devm_clk_get(priv->device, STMMAC_RESOURCE_NAME);
    ...
    clk_prepare_enable(priv->stmmac_clk);

    priv->pclk = devm_clk_get(priv->device, "pclk");
    ...
    clk_prepare_enable(priv->pclk);

    priv->stmmac_rst = devm_reset_control_get(priv->device,STMMAC_RESOURCE_NAME);
    ...
    if (priv->stmmac_rst)
        reset_control_deassert(priv->stmmac_rst);

    ret = stmmac_hw_init(priv);   /*硬件初始化，stmmac_main.c*/
    ...

    ndev->netdev_ops = &stmmac_netdev_ops;     /*网络设备操作结构实例，stmmac_main.c*/

    ndev->hw_features = NETIF_F_SG | NETIF_F_IP_CSUM | NETIF_F_IPV6_CSUM |
                        NETIF_F_RXCSUM;              /*设置网络设备功能集*/
    ndev->features |= ndev->hw_features | NETIF_F_HIGHDMA;
    ndev->watchdog_timeo = msecs_to_jiffies(watchdog);
#ifdef STMMAC_VLAN_TAG_USED
    /* Both mac100 and gmac support receive VLAN tag detection */
    ndev->features |= NETIF_F_HW_VLAN_CTAG_RX;
#endif
    priv->msg_enable = netif_msg_init(debug, default_msg_level);
```

```
    if (flow_ctrl)
        priv->flow_ctrl = FLOW_AUTO;       /* RX/TX pause on */

    if ((priv->synopsys_id >= DWMAC_CORE_3_50) && (!priv->plat->riwt_off)) {
        priv->use_riwt = 1;
        pr_info(" Enable RX Mitigation via HW Watchdog Timer\n");
    }

    netif_napi_add(ndev, &priv->napi, stmmac_poll, 64);
                                        /*设置 napi_struct 实例，poll()函数为 stmmac_poll()*/
    ...
    ret = register_netdev(ndev);     /*注册 net_device 实例*/
    ...
    if (!priv->plat->clk_csr)
        stmmac_clk_csr_set(priv);
    else
        priv->clk_csr = priv->plat->clk_csr;

    stmmac_check_pcs_mode(priv);

    if (priv->pcs != STMMAC_PCS_RGMII && priv->pcs != STMMAC_PCS_TBI &&
        priv->pcs != STMMAC_PCS_RTBI) {
        /* MDIO bus Registration */
        ret = stmmac_mdio_register(ndev);        /*创建并注册 mii_bus 实例，stmmac_mdio.c*/
        ...
    }
    return 0;
    ...
}
```

stmmac_dvr_probe()函数主要工作是分配网络设备 net_device 和私有数据结构 stmmac_priv，并设置这两个实例，最后注册 net_device 实例。

stmmac_set_ethtool_ops(ndev)函数设置 net_device->ethtool_ops 成员为 **stmmac_ethtool_ops** 实例，实例中函数用于设置/获取网络设备参数。stmmac_ethtool_ops 实例中函数代码请读者自行阅读。

网络设备操作结构指针成员 netdev_ops 指向 **stmmac_netdev_ops** 实例，实现网络设备操作（后面再介绍此实例）。**stmmac_hw_init(priv)**函数用于实现硬件设备的初始化，见下文。netif_napi_add()函数设置并添加 stmmac_priv 实例中的 napi_struct 实例成员。register_netdev(ndev)用于注册 net_device 实例，最后调用的函数 stmmac_mdio_register(ndev)用于创建和注册 mii_bus 实例。

下面看一下初始化硬件的 stmmac_hw_init()函数的实现，函数代码如下（stmmac_main.c）：

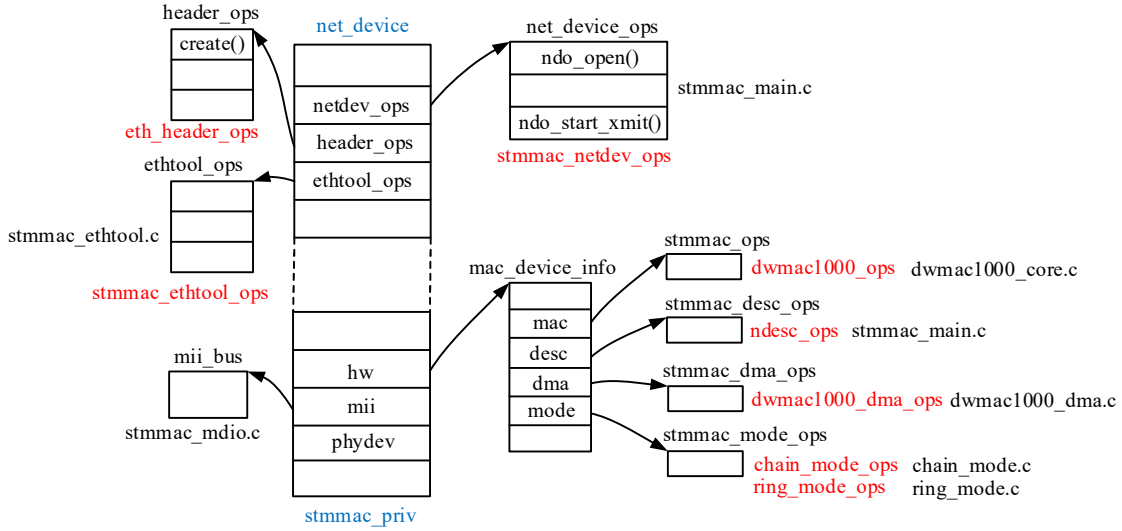
```
static int stmmac_hw_init(struct stmmac_priv *priv)
{
    struct mac_device_info *mac;

    if (priv->plat->has_gmac) {      /*具有 GMAC 控制器，初始化硬件，千兆网卡*/
        priv->dev->priv_flags |= IFF_UNICAST_FLT;
```

```
        mac = dwmac1000_setup(priv->ioaddr,priv->plat->multicast_filter_bins,
                                        priv->plat->unicast_filter_entries);
                        /*创建并设置 mac_device_info 实例，dwmac1000_core.c*/
} else {        /*百兆网卡*/
        mac = dwmac100_setup(priv->ioaddr);
}
if (!mac)
        return -ENOMEM;

priv->hw = mac;      /*指向 mac_device_info 实例*/

/* Get and dump the chip ID */
priv->synopsys_id = stmmac_get_synopsys_id(priv);

/*设置描述符列表操作结构 stmmac_mode_ops*/
if (chain_mode) {      /*模块参数*/
        priv->hw->mode = &chain_mode_ops;      /*链表 DMA 描述符，chain_mode.c*/
        pr_info(" Chain mode enabled\n");
        priv->mode = STMMAC_CHAIN_MODE;
} else {                                        /*环式 DMA 描述符*/
        priv->hw->mode = &ring_mode_ops;    /*ring_mode.c*/
        pr_info(" Ring mode enabled\n");
        priv->mode = STMMAC_RING_MODE;
}

/* Get the HW capability (new GMAC newer than 3.50a) */
priv->hw_cap_support = stmmac_get_hw_features(priv);
if (priv->hw_cap_support) {
        pr_info(" DMA HW capability register supported");
        priv->plat->enh_desc = priv->dma_cap.enh_desc;
        priv->plat->pmt = priv->dma_cap.pmt_remote_wake_up;

        /* TXCOE doesn't work in thresh DMA mode */
        if (priv->plat->force_thresh_dma_mode)
                priv->plat->tx_coe = 0;
        else
                priv->plat->tx_coe = priv->dma_cap.tx_coe;

        if (priv->dma_cap.rx_coe_type2)
                priv->plat->rx_coe = STMMAC_RX_COE_TYPE2;
        else if (priv->dma_cap.rx_coe_type1)
                priv->plat->rx_coe = STMMAC_RX_COE_TYPE1;

} else
        pr_info(" No HW DMA feature register supported");
```

```
    /* To use alternate (extended) or normal descriptor structures */
    stmmac_selec_desc_mode(priv);       /*priv->hw->desc = &ndesc_ops*/
            /*选择 stmmac_desc_ops 实例，stmmac_main.c*/

    if (priv->plat->rx_coe) {
        priv->hw->rx_csum = priv->plat->rx_coe;
        ...
    }
    ...

    if (priv->plat->pmt) {
        pr_info(" Wake-Up On Lan supported\n");
        device_set_wakeup_capable(priv->device, 1);
    }

    return 0;
}
```

stmmac_hw_init()函数中将根据 MAC 控制器模式调用 dwmac1000_setup()或 dwmac100_setup()函数执行硬件初始化，主要是设置 stmmac_priv->hw->mac 和 stmmac_priv->hw->dma 成员。根据 DMA 描述符的形式设置 stmmac_priv->hw->mode 成员，调用 stmmac_selec_desc_mode()函数设置 stmmac_priv->hw->mode 成员。

总之，在探测函数 stmmac_pltfr_probe()创建、设置的 net_device 和 stmmac_priv 实例如下图所示，并最后向内核注册 net_device 实例。各结构体实例定义所在的文件名称已注明，源代码请读者自行阅读。



●网络设备操作结构

在启用网络设备时，将调用网络设备操作结构中的 ndo_open()函数执行特定于网络设备的打开操作。在发送数据包的操作中，dev_queue_xmit()函数最终调用网络设备操作结构中的 ndo_start_xmit()函数将帧传递到网络设备。下面简要列出网络设备操作结构 **stmmac_netdev_ops** 实例（stmmac_main.c）：

```
static const struct net_device_ops stmmac_netdev_ops = {
    .ndo_open = stmmac_open,        /*打开网络设备操作函数，stmmac_main.c*/
    .ndo_start_xmit = stmmac_xmit,      /*发送数据帧函数*/
    .ndo_stop = stmmac_release,
```

```
    .ndo_change_mtu = stmmac_change_mtu,
    .ndo_fix_features = stmmac_fix_features,
    .ndo_set_features = stmmac_set_features,
    .ndo_set_rx_mode = stmmac_set_rx_mode,
    .ndo_tx_timeout = stmmac_tx_timeout,
    .ndo_do_ioctl = stmmac_ioctl,
#ifdef CONFIG_NET_POLL_CONTROLLER
    .ndo_poll_controller = stmmac_poll_controller,
#endif
    .ndo_set_mac_address = eth_mac_addr,
};
```

◎打开操作函数

打开操作函数为 stmmac_open()，代码简列如下：

```
static int stmmac_open(struct net_device *dev)
{
    struct stmmac_priv *priv = netdev_priv(dev);
    int ret;

    stmmac_check_ether_addr(priv);

    if (priv->pcs != STMMAC_PCS_RGMII && priv->pcs != STMMAC_PCS_TBI &&
        priv->pcs != STMMAC_PCS_RTBI) {
        ret = stmmac_init_phy(dev);     /*创建 phy_device 实例，stmmac_main.c*/
        ...
    }

    memset(&priv->xstats, 0, sizeof(struct stmmac_extra_stats));
    priv->xstats.threshold = tc;

    /* Create and initialize the TX/RX descriptors chains. */
    priv->dma_tx_size = STMMAC_ALIGN(dma_txsize);
    priv->dma_rx_size = STMMAC_ALIGN(dma_rxsize);
    priv->dma_buf_sz = STMMAC_ALIGN(buf_sz);

    ret = alloc_dma_desc_resources(priv);
            /*分配发送接收资源，缓存空间，DMA 描述符等，stmmac_main.c*/
    ...
    ret = init_dma_desc_rings(dev, GFP_KERNEL);   /*初始化 DMA 描述符，stmmac_main.c*/
    ...
    ret = stmmac_hw_setup(dev, true);     /*启动 MAC 控制器，stmmac_main.c*/
    ...
    stmmac_init_tx_coalesce(priv);

    if (priv->phydev)
        phy_start(priv->phydev);   /*PHY 初始化，/drivers/net/phy/phy.c*/
```

```
    ret = request_irq(dev->irq, stmmac_interrupt,IRQF_SHARED, dev->name, dev);
                                    /*注册中断，中断处理函数为 stmmac_interrupt()，stmmac_main.c*/
    ...

    napi_enable(&priv->napi);     /*使能 napi_struct 实例，/include/linux/netdevice.h*/
    netif_start_queue(dev);       /*启用发送队列，/include/linux/netdevice.h*/

    return 0;
    ...
}
```

注意在 stmmac_open()函数中注册了中断处理函数 stmmac_interrupt()。

◎发送数据包函数

发送链路层帧的 **stmmac_xmit()**函数定义如下：

```
static netdev_tx_t stmmac_xmit(struct sk_buff *skb, struct net_device *dev)
{
    struct stmmac_priv *priv = netdev_priv(dev);
    unsigned int txsize = priv->dma_tx_size;
    unsigned int entry;
    int i, csum_insertion = 0, is_jumbo = 0;
    int nfrags = skb_shinfo(skb)->nr_frags;     /*分段数量*/
    struct dma_desc *desc, *first;       /*DMA 描述符*/
    unsigned int nopaged_len = skb_headlen(skb);
    unsigned int enh_desc = priv->plat->enh_desc;

    spin_lock(&priv->tx_lock);
    ...

    entry = priv->cur_tx % txsize;

    csum_insertion = (skb->ip_summed == CHECKSUM_PARTIAL);

    if (priv->extend_desc)
        desc = (struct dma_desc *)(priv->dma_etx + entry);
    else
        desc = priv->dma_tx + entry;     /*指向 DMA 描述符*/

    first = desc;     /*第一个 DMA 描述符*/

    /*对 DMA 描述符编程*/
    if (enh_desc)
        is_jumbo = priv->hw->mode->is_jumbo_frm(skb->len, enh_desc);

    if (likely(!is_jumbo)) {
        desc->des2 = dma_map_single(priv->device, skb->data,nopaged_len, DMA_TO_DEVICE);
```

```
        if (dma_mapping_error(priv->device, desc->des2))
            goto dma_map_err;
        priv->tx_skbuff_dma[entry].buf = desc->des2;
        priv->hw->desc->prepare_tx_desc(desc, 1, nopaged_len,csum_insertion, priv->mode);
} else {
        desc = first;
        entry = priv->hw->mode->jumbo_frm(priv, skb, csum_insertion);
        ...
}

for (i = 0; i < nfrags; i++) {
        const skb_frag_t *frag = &skb_shinfo(skb)->frags[i];
        int len = skb_frag_size(frag);

        priv->tx_skbuff[entry] = NULL;
        entry = (++priv->cur_tx) % txsize;
        if (priv->extend_desc)
            desc = (struct dma_desc *)(priv->dma_etx + entry);
        else
            desc = priv->dma_tx + entry;

        desc->des2 = skb_frag_dma_map(priv->device, frag, 0, len,DMA_TO_DEVICE);
        ...
        priv->tx_skbuff_dma[entry].buf = desc->des2;
        priv->tx_skbuff_dma[entry].map_as_page = true;
        priv->hw->desc->prepare_tx_desc(desc, 0, len, csum_insertion,priv->mode);
        wmb();
        priv->hw->desc->set_tx_owner(desc);
        wmb();
}

priv->tx_skbuff[entry] = skb;

/*最后一个分段*/
priv->hw->desc->close_tx_desc(desc);

wmb();
priv->tx_count_frames += nfrags + 1;
if (priv->tx_coal_frames > priv->tx_count_frames) {
        priv->hw->desc->clear_tx_ic(desc);
        priv->xstats.tx_reset_ic_bit++;
        mod_timer(&priv->txtimer,STMMAC_COAL_TIMER(priv->tx_coal_timer));
} else
        priv->tx_count_frames = 0;

priv->hw->desc->set_tx_owner(first);
```

```
        wmb();

        priv->cur_tx++;

        ...
        dev->stats.tx_bytes += skb->len;

        if (unlikely((skb_shinfo(skb)->tx_flags & SKBTX_HW_TSTAMP) &&priv->hwts_tx_en)) {
            /* declare that device is doing timestamping */
            skb_shinfo(skb)->tx_flags |= SKBTX_IN_PROGRESS;
            priv->hw->desc->enable_tx_timestamp(first);
        }

        if (!priv->hwts_tx_en)
            skb_tx_timestamp(skb);

        netdev_sent_queue(dev, skb->len);
        priv->hw->dma->enable_dma_transmission(priv->ioaddr);      /*使能 DMA 传输*/

        spin_unlock(&priv->tx_lock);
        return NETDEV_TX_OK;
        ...
    }
```

发送链路层帧的 **stmmac_xmit()**函数简单地说就是先调用 stmmac_desc_ops 结构体实例中的函数对 DMA 描述符进行设置（编程），然后调用 stmmac_dma_ops 实例中的函数使能 DMA 传输。

●接收数据包函数

在网络设备操作 stmmac_open()函数中注册了中断处理函数 **stmmac_interrupt()**，函数代码简列如下：

```
    static irqreturn_t stmmac_interrupt(int irq, void *dev_id)     /*stmmac_main.c*/
    {
        struct net_device *dev = (struct net_device *)dev_id;
        struct stmmac_priv *priv = netdev_priv(dev);

        if (priv->irq_wake)
            pm_wakeup_event(priv->device, 0);
        ...
        /* To handle GMAC own interrupts */
        if (priv->plat->has_gmac) {
            int status = priv->hw->mac->host_irq_status(priv->hw,&priv->xstats);
            if (unlikely(status)) {
                /* For LPI we need to save the tx status */
                if (status & CORE_IRQ_TX_PATH_IN_LPI_MODE)
                    priv->tx_path_in_lpi_mode = true;
                if (status & CORE_IRQ_TX_PATH_EXIT_LPI_MODE)
                    priv->tx_path_in_lpi_mode = false;
            }
```

```
    }

    /*处理 DMA 中断，stmmac_main.c*/
    stmmac_dma_interrupt(priv);

    return IRQ_HANDLED;
}
```

stmmac_dma_interrupt(priv)函数用于处理 DMA 中断，函数定义如下（stmmac_main.c）：

```
static void stmmac_dma_interrupt(struct stmmac_priv *priv)
{
    int status;
    int rxfifosz = priv->plat->rx_fifo_size;

    status = priv->hw->dma->dma_interrupt(priv->ioaddr, &priv->xstats);     /*DMA 中断处理函数*/
    if (likely((status & handle_rx)) || (status & handle_tx)) {
        if (likely(napi_schedule_prep(&priv->napi))) {
            stmmac_disable_dma_irq(priv);     /*关闭 DMA 中断*/
            __napi_schedule(&priv->napi);      /*将 napi_struct 实例添加到轮询表，触发软中断*/
        }
    }
    if (unlikely(status & tx_hard_error_bump_tc)) {
        ...
    } else if (unlikely(status == tx_hard_error))
        stmmac_tx_err(priv);
}
```

stmmac_dma_interrupt()函数调用 stmmac_dma_ops 实例中的 dma_interrupt()函数处理中断，然后调用函数__napi_schedule(&priv->napi)将 napi_struct 实例添加到轮询表，并触发接收软中断。

在前面介绍的探测函数 stmmac_pltfr_probe()中调用了 netif_napi_add(ndev, &priv->napi, stmmac_poll, 64)函数添加了 napi_struct 实例，实例中 poll()函数为 stmmac_poll()，用于接收数据帧，函数定义如下：

```
static int stmmac_poll(struct napi_struct *napi, int budget)       /*stmmac_main.c*/
{
    struct stmmac_priv *priv = container_of(napi, struct stmmac_priv, napi);
    int work_done = 0;

    priv->xstats.napi_poll++;
    stmmac_tx_clean(priv);

    work_done = stmmac_rx(priv, budget);     /*管理接收过程，创建 sk_buff 实例等，stmmac_main.c*/
    if (work_done < budget) {
        napi_complete(napi);     /*将实例从轮询表移出等，/include/linux/netdevice.h*/
        stmmac_enable_dma_irq(priv);      /*开启 DMA 中断，stmmac_main.c*/
    }
    return work_done;
}
```

3 小结

本小节简要介绍了以太网协议及网络设备驱动程序的实现。至此，数据从用户空间到网络设备和从网络设备到用户空间的流程大致走了一遍。读者通过这几节的学习对 TCP/IPv4 网络协议簇在 Linux 内核中的实现应该有了一个整体的认识。由作者水平有限，还有许多的细节有待研究。

13.3 IPv6 协议实现简介

IP 是 TCP/IP 协议簇中的核心协议，即网络层协议。所有 UDP、TCP、ICMP 和 IGMP 数据都通过 IP 数据报传输。IP 提供一种尽力而为、无连接的数据报交付服务。IP 协议目前有两个版本，分别是 IPv4 和 IPv6。在 IPv4 中 IP 地址由 32 位表示，能标识的地址数量为 2^{32}，IPv4 地址将很快被用尽。IPv6 是 IPv4 的升级版本，主要解决 IP 地址不足的问题，它用 128 位来表示 IP 地址，提供了巨大的地址空间。IPv6 与 IPv4 类似，但又有所改进，不只是扩大了地址空间。

本节简要介绍 IPv6 协议内容，然后介绍 IPv6 在内核中的实现，其实现代码位于/net/ipv6/目录下。

13.3.1 IPv6 协议概述

在 IPv6 中，IP 地址是 128 位（16 字节）的，是 IPv4 地址的 4 倍，以解决地址空间不足问题。IP 是 TCP/IP 协议簇中网络层协议，IP 协议的版本升级对传输层、数据链路层协议的影响不是很大。对数据链路层几乎没有影响，对传输层主要是 ICMP 变化比较大。在 IPv6 中 ICMP 被赋予了更多的功能，例如，IPv4 中的 ARP、IGMP 等，在 IPv6 中都由 ICMP 实现。

1 地址

IPv6 地址的传统表示方法是采用称为块或字段的四个十六进制数（每个块 2 字节，a 到 f 用小写表示），这些被称为块或字段的数由冒号分隔。例如，一个包含 8 个块的 IPv6 地址可写为：

5f05:2000:80ad:5800:0058:0800:2023:1d71

虽然不像用户熟悉的十进制数，但将十六进制数转换为二进制更容易。另外，一些已取得共识的 IPv6 地址简化表示法已被标准化：

1.一个块（字段）中前导的 0 不必书写。在上面的例子中，IPv6 地址可写为：

5f05:2000:80ad:5800:58:0800:2023:1d71

2.全 0 的块（字段）可以省略，并用符号::代替。例如，IPv6 地址 0:0:0:0:0:0:0:1 可简写成::1。同样，地址 2001:0db8:0:0:0:0:0:2 可简写成 2001:0db8::2。为了避免出现歧义，一个 IPv6 地址中符号::只能用一次。一般只能用于影响最大的地方（压缩最多的 0）。

3.在 IPv6 格式中嵌入 IPv4 地址可使用混合符号形式，紧接着 IPv4 部分的地址块的值为 ffff，地址其余部分使用点分四组格式。例如，IPv6 地址::ffff:10.0.0.1 可表示 IPv4 地址 10.0.0.1，它被称为 IPv4 映射的 IPv6 地址。

■特殊地址

在 IPv6 中，许多地址范围和个别地址用于特殊用途，它们都列在下表中：

前缀	特殊用途
::/0	默认路由条目，不用于寻址（全 0 地址）。
::/128	未指定地址，可作为源 IP 地址使用。
::1/128	IPv6 主机回送地址，不用于发送出本地主机的数据报中。
::ffff:0:0/96	IPv4 映射地址。这种地址不会出现在分组头部，只用于内部主机。

2001::/32	Teredo 地址。
2001:10::/28	ORCHI（覆盖可路由加密散列标识符）。这种地址不会出现在公共 Internet 中。
2001:db8::/32	用于文档和实例的地址范围。这种地址不会出现在公共 Internet 中。
2002::/16	6to4 隧道中继的 6to4 地址。
fc00::/7	唯一的本地单播地址，不用于全球的 Internet。
fe80::/10	链路本地单播地址。
ff00::/8	IPv6 组播地址，仅作为目的 IP 地址使用。

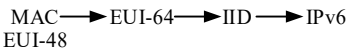
另外，任播地址是一个单播 IPv4 或 IPv6 地址，这些地址根据它所在的网络确定不同的主机。这是通过配置路由器通知 Internet 中多个站点有相同单播路由来实现。因此，一个任播地址不是指 Internet 中的一台主机，而是对于任播地址“最合适”或“最接近”的一台主机。任播地址最常用于发现一台提供了常用服务的计算机。例如，某个数据报发送到一个任播地址，可用于找到 DNS 服务器。

■链路本地地址

IPv6 地址除了比 IPv4 地址长 4 倍这个因素，还有一些额外的特点。IPv6 地址使用特殊前缀表示一个地址范围。一个 IPv6 地址范围是指它可用的网络规模。有关范围的重要例子包括**节点本地**（只用于同一计算机中通信）、**链路本地**（只用于同一网络链路或 IPv6 前缀中的节点）或**全球性**（Internet 范围）。在 IPv6 中，大部分节点通常在同一网络接口上使用多个地址。虽然 IPv4 中也支持这样做，但是并不常见。一个 IPv6 节点中需要一组地址，包括组播地址。

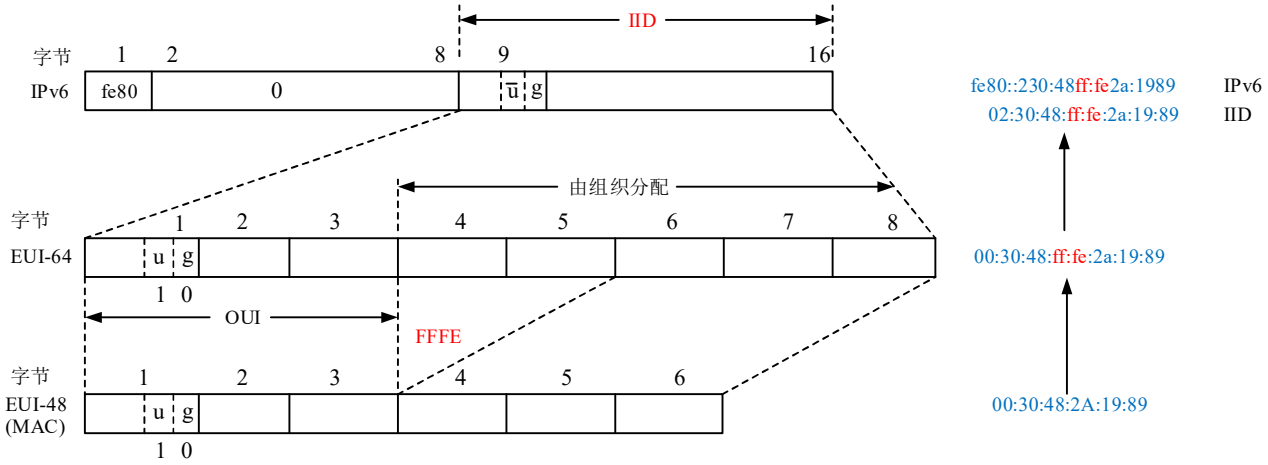
链路本地 IPv6 地址使用接口标识符（IID）作为分配一个单播 IPv6 地址的基础。除了地址是以二进制值 000 开始外，IID 在所有情况下都作为一个 IPv6 地址的低序位，这样它们必须在同一网络中有唯一前缀。链路本地地址形如 fe80::IID，**fe80**::/10 是链路本地地址的前缀，后接 IID 值。

IID 的长度通常是 64 位，并直接由一个网络接口相关的链路层 MAC 地址形成。链路层 MAC 地址使用修改的 EUI-64 格式表示，或者由其它进程随机提供的值形成，以提供可防范地址跟踪的某种程度的隐私保护。网络接口链路层 MAC 地址需要转换成 EUI-64 格式，再转成 IID 值，IID 值再转换成 IPv6 地址，如下图所示：



在 IEEE 标准中，EUI 表示扩展唯一标识符。EUI-64 标识符开始于一个 24 位的组织唯一标识符（OUI），接着是一个由组织分配的 40 位扩展标识符，它由前面 24 位识别，如下图所示。OUI 由 IEEE 注册权威机构来维护和分配。EUI 可能是“统一管理”或“本地管理”的。在 Internet 环境下，这种地址通常是统一管理的。OUI 第一个字节的低两位分别是 u 位和 g 位。当 u 位被设置时，表示该地址是本地管理。当 g 位被设置时，表示该地址是一组或组播类型的地址。目前，我们只关注 g 位未设置的情况。

由修改的 EUI-64 生成 IID 值时，只需要将 u 位取反，其它位不变。IID 值生成 IPv6 地址时前缀设为 **fe80**::/10，低 64 位为 IID 值，如下图所示。



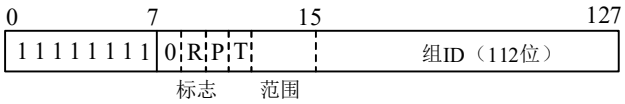
多年来，很多 IEEE 标准兼容的网络接口（如以太网）在使用短格式的地址（MAC 地址），即 48 位的 EUI-48 地址。EUI-48 和 EUI-64 格式之间的显著区别是它们的长度。由 EUI-48 转换成 IPv6 地址时，首先要将 EUI-48 格式转换成 EUI-64，然后再按上面的方法转换成 IPv6 地址。

EUI-48 格式中开始 24 位为 OUI，后 24 位为组织分配位（EUI-64 中为 40 位）。由 EUI-48 格式转换成 EUI-64 格式时，开始的 24 位 OUI 不变，EUI-48 中后 24 位复制到 EUI-64 中高 3 字节，而中间的 4、5 字节用 FFFE 填充，如上图所示。

上图右侧示意了将 48 位 MAC 地址 00:30:48:2A:19:89 依次转换成 EUI-64 格式，IID 值，IPv6 地址的过程。

■组播地址

IPv6 中没有任何广播地址，在 IPv6 中仅使用组播地址。IPv6 对组播的使用相当积极，前缀 ff00::/8 已被预留给组播地址，有 112 位可用于保存组号，可提供的组数为 2^{112}。其一般格式如下图所示：



IPv6 组播地址的第 2 个字节包含一个 4 位**标志**字段和一个 4 位**范围**字段。4 个标志位定义如下：0：保留，R：包含会合点，P：使用单播前缀，T：临时的。4 位范围字段值表示到某些组播地址的数据报的分配限制，即组播的范围（全球、本地等）。组 ID 编码在低序的 112 位中。如果 P 或 R 位被置位，则组 ID 字段将使用一种代替格式，见下文。

4 位范围字段表示到某些组播地址的数据报的分配限制，取值语义如下：

值	范围	值	范围	值	范围
0	保留	4	管理	9~d	未分配
1	接口/机器本地	**5**	**站点本地**	**e**	**全球**
2	**链路/子网本地**	6~7	未分配	f	保留
3	保留	8	组织本地		

很多 IPv6 组播地址由 IANA 分配为永久使用，并且故意跨越多个地址范围。这些组播地址对每个范围都有一定偏移量。例如，可变范围的组播地址 ff0x::101 是为 NTP 服务器预留的，x 表示可变范围。下表显示了一些预留定义的地址。

地址	含义
ff01::101	同一机器中的所有 NTP 服务器
ff02::101	同一链路/子网中所有 NTP 服务器

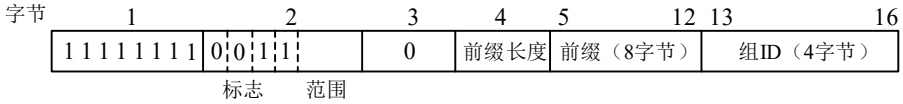
ff04::101	某些管理定义范围内的所有 NTP 服务器
ff05::101	同一站点中的所有 NTP 服务器
ff08::101	同一组织中的的所有 NTP 服务器
ff0e::101	Internet 中的所有 NTP 服务器

在 IPv6 组播地址中，当 P 和 R 标志位为 0 时，组播地址采用上图中的一般格式。当 P 位设置为 1 时，无须基于每个组的全球性许可，对组播地址有两个可选构成方式（组播地址格式），主要是 IPv6 组播地址中组 ID 字段的格式不同。第一种方式称为基于单播前缀的 IPv6 组播地址分配，它由单播前缀生成 IPv6 组播地址中的组 ID 字段。第二种方式称为链路范围的 IPv6 组播地址分配，它使用 IID 值生成 IPv6 组播地址中的组 ID 字段。

为了了解这些不同格式如何工作，首先要了解 IPv6 组播地址中标志字段位的使用细节。它们被定义在下表中：

位字段（标志）	含义
R	会合点标志（0，常规；1，包括 RP 地址）
P	前缀标志（0，常规；1，基于单播前缀的地址）
T	临时标志（0，永久分配的；1，临时的）

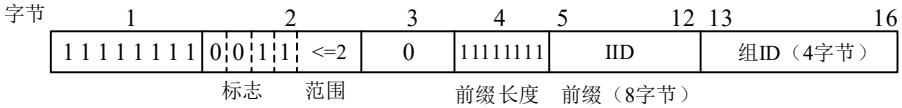
当 T 位字段被设置时，表示组地址是临时或动态分配的，这不是标准地址。当 P 位字段被设置为 1 时，T 字节也必须为 1。当这种情况发生时，使用基于单播地址前缀的特殊格式的 IPv6 组播地址，如下图所示。



上图显示的是第一种可选 IPv6 组播地址格式，它使用基于单播前缀的地址改变组播地址格式（组 ID 字段），包括一个单播前缀及其长度，以及一个更小的组 ID（32 位）。该方案的目的是提供全球唯一的 IPv6 组播地址分配方式，同时不需要提出新的全球性机制。由于 IPv6 单播地址已分配全球性的前缀单元，所以在组播地址中可以使用这个前缀中的位，从而在组播应用中利用现有的单播地址分配方法。

例如，一个组织分配了一个单播前缀 3ffe:ffff:1::/48，那么它随之分配了一个基于单播前缀的组播前缀 ff3x:30:3ffe:ffff:1/96，其中 x 是任何有效范围。

第二种可选的 IPv6 组播地址格式用于创建链路本地范围的组播地址。它是一种基于 IID 的方法，当只需要链路本地范围时，这种方法是基于单播前缀分配的首选。在这种情况下，可使用另一种形式的 IPv6 组播地址格式，如下图所示：



第二种可选格式与第一种可选格式的主要区别是，前缀长度字节值固定为 255，前缀字段（8 字节）由 IID 值代替。这种格式的优点是不需要提供前缀以形成组播地址，在不需要路由器的 Ad hoc（无线自组织）网络中，一台单独的计算机可基于自己的 IID 值形成唯一的组播地址，而无需运行一个复杂的许可协议。如前所述，这种格式只适用于本地链路或节点组播范围。但是，需要更大的范围时，无论是基于单播前缀的地址还是永久组播地址都可以使用。

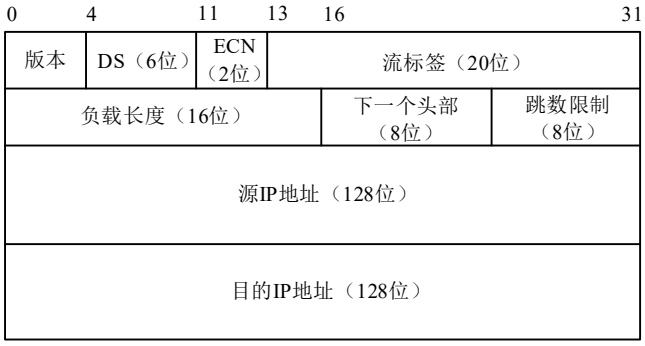
与 IPv4 类似，IPv6 也有一些保留的组播地址。除了前面提到的可变范围地址，这些地址还根据范围划分成组，下表给出了一个 IPv6 组播空间中的保留组播地址列表。

地址	范围	特殊用途
ff01::1	节点	所有节点
ff01::2	节点	所有路由器
ff01::fb	节点	mDNSv6

ff02::1	链路	所有节点
ff02::2	链路	所有路由器
ff02::4	链路	DVMRP 路由器
ff02::5	链路	OSPFIGP
ff02::6	链路	基于 OSPFIGP 设计的路由器
ff02::9	链路	RIPng 路由器
ff02::a	链路	EIGRP 路由器
ff02::d	链路	PIM 路由器
ff02::16	链路	支持 MLDv2 路由器
ff02::6a	链路	所有探测器
ff02::6d	链路	LL-MANET 路由器
ff02::fb	链路	mDNSv6
ff02::1:2	链路	所有 DHCP 代理
ff02::1:3	链路	LLMNR
ff02::1:ffxx:xxxx	链路	请求节点地址范围
ff05::2	站点	所有路由器
ff05::fb	站点	mDNSv6
ff05::1:3	站点	所有 DHCP 服务器
ff0x::	可变的	保留
ff0x::fb	可变的	mDNSv6
ff0x::101	可变的	NTP
ff0x::133	可变的	聚合服务器访问协议
ff0x::18c	可变的	所有 AC 地址（CAPWAP）
ff3x::/32	特殊的	SSM 块

2 报头

IPv6 报头如下图所示：



IPv6报头（40字节）

IPv6 报头大小固定为 40 字节，包含 128 位的源 IP 地址和目的 IP 地址，其余字段语义简介如下：

●**版本**：表示 IP 协议版本号，4bits，IPv6 为 6。

●**DS**：区分服务，6bits，用于支持 Internet 上不同类型的服务，详见下文。

●**ECN**：拥塞标识符，2bits。一台持续拥塞的具有 ECN 感知能力的路由器在转发分组时会设置这两位。

●**流标签**：待续...

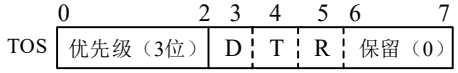
●**负载长度**：IP 数据报长度，不包括 IPv6 报头，但是扩展报头包括在负载长度中。

●**下一个头部**：下一个头部类型，IPv6 扩展报头或传输层报头，见下文。

●**跳数限制**：相当于 IPv4 中的 TTL 字段，用于设置一个数据报可经过的路由器的数量上限。

IPv4 报头第 3 和 IPv6 报头中的第 2 字段都为 DS 字段（区分服务，IPv4 为 TOS）。区分服务是一个框架和一组标准，用于支持 Internet 上不同类型的服务（即不只是尽力而为的服务）。IP 数据报以某种方式被标记，使它们的转发不同于其它的数据报。这样做可能导致网络中排队延时的增加或减少，以及出现其他特殊效果。DS 字段中的数字称为区分服务代码点（DSCP）。“代码点”指的是预定义的具有特定含义的位。在通常情况下，如果数据报拥有一个分配的 DSCP，它在通过网络基础设施交付过程中会保持不变。但是，某些策略可能导致一个数据报中的 DSCP 在交付过程中改变。

DS 字段是代替以前定义在 IPv4 中的服务类型（TOS）和 IPv6 中的流量类别字段。尽管原来的 TOS 和流量类型字段没有得到广泛支持，但 DS 字段结构仍提供了一些对它们的兼容能力。为了对其如何工作有更清楚的了解，我们首先回顾服务类型（TOS）的原始结构，如下图所示：

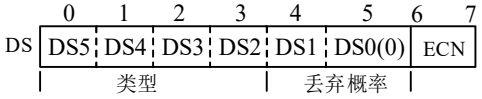


D、T 和 R 子字段表示数据报在延时、吞吐量和可靠性方面得到良好的处理。相应值为 1 表示更好的处理（分别为低延时、高吞吐量和高可靠性）。优先级取值范围是从 000（常规）到 111（网络控制），表示优先级依次递增，见下表。它们都基于一个称为多级优先级与抢占的方案，其中较低的优先级的呼叫可被更高优先级的呼叫抢占。

TOS 的优先级子字段值

值	优先级名称	值	优先级名称
000	常规	100	瞬间覆盖
001	优先	101	严重
010	立即	110	网络控制
011	瞬间	111	网络控制

在定义 DS 字段时，优先级的值已定义，以提供有限的兼容性。在下图中，6 位的 DS 字段用于保存 DSCP，提供对 64 个代码点的支持。特定 DSCP 值可通知路由器对接收的数据报进行转发或特殊处理。不同类型的转发处理表示为每跳行为（PHB），因此 DSCP 值可有效通知路由器哪种 PHB 被应用于数据报。DSCP 的默认值通常为 0，对应于常规尽力而为的 Internet 流量。



64 个可能的 DSCP 值（6bits）分为不同用途，如下表所示：

池	代码点前缀	策略
1	xxxxx0	标准的
2	xxxx11	EXP/LU
3	xxxx01	EXP/LU(*)

DSCP 值被分为 3 个池，分别是标准的、实验/本地用途的（EXP/LU），最终打算标准化的实验/本地用途 EXP/LU(*)。以 0 作为结尾（DS0）的 DSCP 用于标准用途，以 1 作为结尾的 DSCP 用于实验或本地用途。以 01 作为结尾的 DSCP 最初打算用于实验或本地用途，但最终会走向标准化。

前 3 位类型字段基于较早定义的服务类型的优先级子字段。路由器通常先将流量分为不同的类别。常见类别的流量可能有不同的丢弃概率，如果路由器被迫丢弃流量，允许路由器确定首先丢弃哪些流量。3 位的类别选择器提供了 8 个定义的代码点，它们对应于一个指定最小功能集的 PHB，提供与早期的 IP 优先级相似的功能。它们称为类别选择兼容的 PHB，目的是支持部分兼容的最初定义的 IP 优先级子字段。

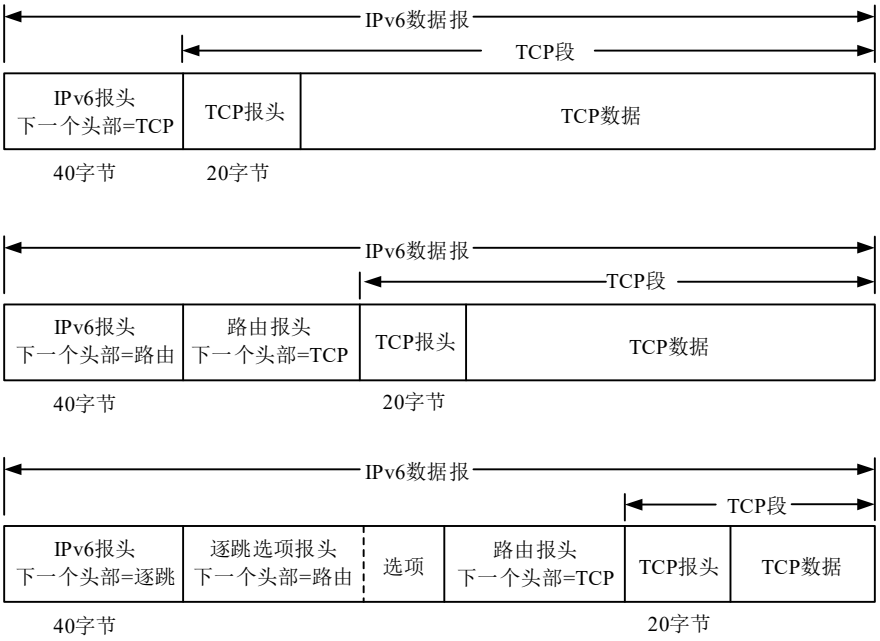
xxx000 形式的代码点总被映射为这种 PHB，但是其他值也可映射到相同 PHB。

在一个流量类型中，数据报被分配一个丢弃优先级。在一个类型中，较高丢弃优先级的数据报优先于那些较低丢弃优先级的数据报处理（即较高优先级的数据报先转发）。

3 扩展报头

在 IPv6 报头中没有选项字段，那些由 IPv4 选项提供的特殊功能，通过在 IPv6 报头之后增加扩展报头实现。IPv4 路由和时间戳功能都采用这种方式，其它功能（如分片和超大分组等）很少在 IPv6 中使用，因此没有为它们在 IPv6 报头部分配相应的位。基于这种设计，IPv6 报头固定为 40 字节，扩展报头仅在需要时添加。在选择 IPv6 报头为固定大小时，要求扩展报头仅由终端系统（仅有一个扩展报头例外）处理，IPv6 设计简化了高性能路由器的设计和实现，这是因为 IPv6 路由器处理分组所需的命令比 IPv4 简单。实际上，分组处理性能受很多因素的影响，包括协议复杂性，路由器硬件和软件功能，以及流量负载等。

扩展报头和更高层协议报头与 IPv6 报头链接起来构成级联的报头，如下图所示。每个报头中的“下一个头部”字段表示紧跟着的报头类型，它可能是一个 IPv6 扩展报头或其它类型，值 59 表示这个报头链的结尾。



上图中第一个 IPv6 数据报中不存在扩展报头，IPv6 报头后紧跟着 TCP 报头。在第二个 IPv6 数据报中，存在路由扩展报头，后面紧接 TCP 报头。在第三个 IPv6 数报中又增加了逐跳选项扩展报头，此扩展报头与 IPv4 报头一样，带有可变长度的选项字段，后面依次是路由扩展报头和 TCP 报头。

下表列出了 IPv6 中扩展报头的类型、在级联报头中的位置、扩展报头值（下一个头部中的值）等信息：

报头类型（名称）	位置（顺序）	值	备注
IPv6 报头	1	41	
逐跳选项	2	0	必须紧跟在 IPv6 报头之后，报头中带选项。
目的地选项	3，8	60	报头中带选项。
路由	4	43	
分片	5	44	
封装安全载荷（ESP）	7	50	
认证（AH）	6	51	
移动（MIPv6）	9	135	
没有头部	最后	59	
ICMPv6	最后	**58**	

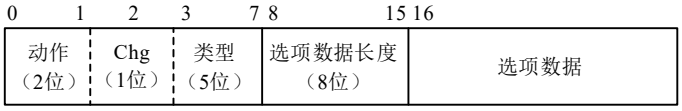
UDP	最后	**17**	
TCP	最后	**6**	
各种其它高层协议	最后	-	

上表中除了“逐跳选项”扩展报头的位置外（强制的），其它扩展报头的位置是建议性的，因此 IPv6 实现必须按接收顺序处理扩展报头。只有“目的地选项”报头可以使用两次，第一次是指出包含在 IPv6 报头中的目的 IPv6 地址，第二次是关于数据报的最终目的地。在“逐跳选项”和“目的地选项”扩展报头内可带 IPv6 选项（类似 IPv4 报头中选项）。

■IPv6 选项

IPv6 选项与 IPv4 选项类似，可放入“逐跳选项”或“目的地选项”扩展报头中。“逐跳选项”（HOPOPT）是唯一由数据报经过的每个路由器处理的扩展报头。“目的地选项”扩展报头仅与接收方相关。这两个扩展报头的编码格式一样。

IPv6 选项被编码为“类型-长度-值”（TLV）集合，如下图所示：



选项中前 2 个字节表示选项类型和选项数据长度，后面是可变长度的选项数据。动作字段表示选项没有被识别时，一个 IPv6 节点是转发还是丢弃该数据报，以及是否向发送方返回一个消息，如下表所示：

值	动作
00	跳过选项，继续处理。
01	丢弃数据报。
10	丢弃数据报，并向源地址发送一个“ICMPv6 参数问题”消息。
11	与 10 相同，但仅在数据报目的地址不是组播时，才发回 ICMP 消息。

当选项数据可能在数据报转发过程中改变时，Chg 位段设置为 1。

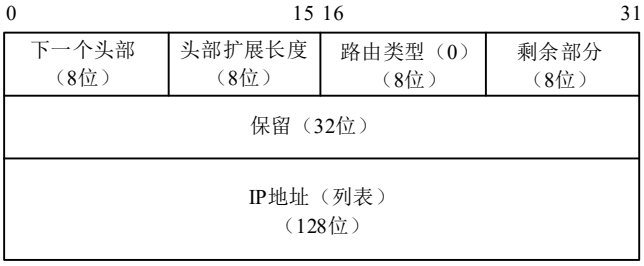
下表列出了 IPv6 选项信息（H：逐跳选项，D：目的地选项）：

选项名称	扩展报头	动作	改变	类型	长度	备注
填充 1	HD	00	0	0	无	用于一字节填充
填充 N	HD	00	0	1	可变	用于 N 字节填充
超大有效载荷	H	11	0	194	4	选项数据为数据报长度值
隧道封装限制	D	00	0	4	4	
路由器警告	H	00	0	5	4	
快速启动	H	00	1	6	8	
CALIPSO	H	00	0	7	8^{+}	
家乡地址	D	11	0	201	16	

■路由扩展报头

IPv6 路由扩展头部为发送方提供了一种 IPv6 数据报控制的机制，以控制数据报通过网络的路径。简单地说，就是在扩展头部中指定了部分路由器的 IPv6 地址，数据报转发过程中必须经过这些路由器。目前路由扩展报头有两个版本，分别为类型 0（RH0）和类型 2（RH2），出于安全考虑 RH0 已经被否决，RH2 被定义成与移动 IP 共同使用。

路由扩展报头结构如下图所示：



路由类型为 2，表示 RH2，因为 RH0 已弃用。扩展报头中只存在一个 IP 地址（RH0 中存在多个 IP 地址）。剩余部分表示还有多少个必须经过的路由器还没有经过。

发送 IPv6 数据报时，IPv6 报头中目的 IP 地址字段设为第一个必经路由器的 IP 地址，数据报到达此路由器时，将路由扩展报头中第一个 IP 地址（下一个必须路由器）与 IPv6 报头中目的 IP 地址交换，数据报再往下传输，到达下一个必经路由器后，再交换 IPv6 中目的 IP 地址与扩展报头中第二个 IP 地址，数据报再往下传输，依此类推。

也就是说，数据报最终目的地 IP 地址保存在路由扩展报头 IP 地址列表的最后面，数据报 IPv6 报头中最开始目的 IP 地址和扩展报头中 IP 地址（除最后一个），表示必经路由器的 IP 地址。

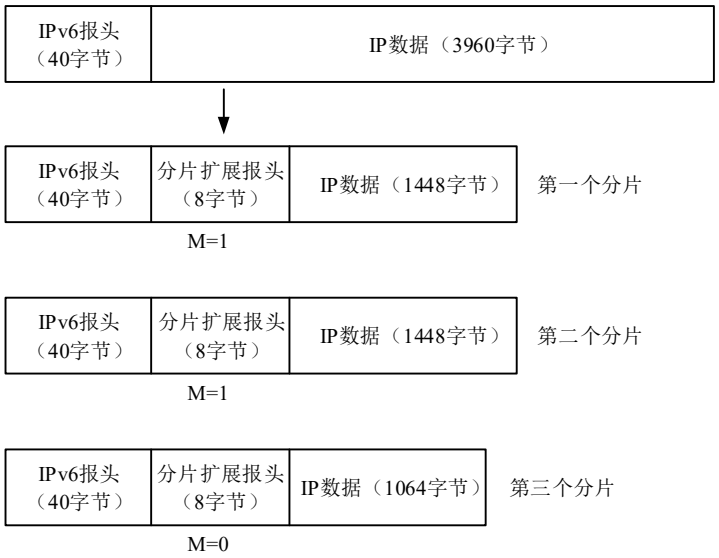
在 RH2 中，路由扩展报头中 IP 地址列表中只有一个 IP 地址，也就是说只能指定一个必经的路由器。

■分片扩展报头

在 IPv6 中只有数据报的发送者可以对数据报进行分片（分段），而中间路由器不能执行分片操作。分片扩展报头用于 IPv6 源节点向目的地发送一个大于路径 MTU 的数据报。在这种情况下需要向 IPv6 数据报（每个分片数据报）添加一个分片扩展报头，其结构如下图所示：

分片扩展报头中字段的语义与 IPv4 报头分片相关的字段语义相似。保留和 Res 字段为 0，接收方会忽略它们。M 位段置位表示后面还有分片数据报。分片偏移量写 IPv4 报头中的分片偏移量语义相同，表示数据报中起始数据在原始数据报中的偏移量（8 字节为单位）。

在分片过程中，输入数据报称为原始数据报，它由两部分组成：不可分片部分和可分片部分。不可分片部分包括 IPv6 报头和任何在到达目的地之前需由中间节点处理的扩展报头。可分片部分包括数据报的其余部分。当原始数据报被分片后，将产生多个分片数据报，每个分片数据报中包含原始数据报中的部分数据，并添加一个分片扩展报头，另外还需要修改 IPv6 报头中负载长度字段值。下图简要示意了 IPv6 数据报分片过程：



前面介绍了 IPv6 地址、IPv6 报头等协议信息，IP 最重要的功能是实现数据报的转发。IPv6 转发与传统 IPv4 转发只有很少改变。除了更长的地址外，IPv6 还使用一种稍微不同的机制（邻居请求消息），以确定它的下一跳的低层地址。另外，IPv6 定义了链路本地地址和全球地址。全球地址的处理方式就像普通的IP地址，链路本地地址只能用于同一链路上。另外，所有的链路本地地址共享相同的IPv6前缀（fe80::/10），在发送一个目的地为链路本地地址的数据报时，一台多宿主主机可能需要用户来决定使用哪个接口。

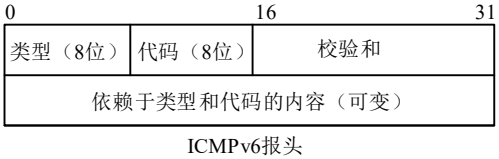
4 ICMPv6

网络层协议版本的升级，由 IPv4 升级到 IPv6，对传输层协的影响并不大。对 UDP、TCP 而言，主要区别是计算检验和时使用的伪报头中的 IP 地址由 32 位变成了 128 位。影响最大的是 ICMP，IPv6 中 ICMP 被赋予更多的任务。下面主要介绍 ICMP 在 IPv6 中的定义（ICMPv6）。

■概述

ICMP（Internet 控制消息协议）用于提供与 IP 协议层配置和 IP 数据包处置相关的诊断和控制信息，主要包括差错报文和信息类报文。ICMP 在 IPv4 中的实现（ICMPv4）在前面介绍过了，在 IPv6 中，ICMP（用 ICMPv6 表示）不仅用于一些简单的错误报告和信令，它也用于邻居发现（ND），与 IPv4 中的 ARP 取着相同的作用。它还用于配置主机（DHCP）和管理组播地址的路由器发现功能（类似 IPv4 中的 IGMP），最后，它还被用来帮助管理移动 IPv6 中的切换。

在 IPv6 报头或最后一个扩展报头中，如果其中下一个头部字段值为 58，表示 IP 数据报封装了 ICMPv6 数据报。ICMPv6 报头格式如下（同 IPv4 中报头格式）：



ICMPv6报头

IPv6 中 8 位的类型、代码字段与 IPv4 中的值是不同的（详见下文）。IPv4 中检验和字段覆盖整个 ICMPv4 报文，在 ICMPv6 中，它将涵盖一个来自 IPv6 报头的伪头部。

下表列出了 ICMPv6 中报文的类型，从 0 至 127 为差错报文，128 至 255 为信息类报文：

类型	正式名称	描述
1	目的不可达	不可达的主机、端口、协议

2	数据包太大（PTB）	需要分片
3	超时	跳数用尽或者重组超时
4	参数问题	畸形数据包或者头部
100、101	为私人实验保留	为实验保留
127	为 ICMPv6 差错报文扩充保留	为更多差错报文保留
128	**回显请求**	**ping 请求，可能包含数据**
129	**回显应答**	**ping 应答，返回数据**
130	组播侦听查询	查询组播订阅者（v1）
131	组播侦听报告	组播订阅者报告（v1）
132	组播侦听完成	组播取消订阅报文（v1）
133	**路由器请求（RS）**	**IPv6 RS 和移动 IPv6 选项**
134	**路由器通告（RA）**	**IPv6 RA 和移动 IPv6 选项**
135	**邻居请求（NS）**	**IPv6 邻居发现（请求）**
136	**邻居通告（NA）**	**IPv6 邻居发现（通告）**
137	重定向报文	使用另一个下一跳路由器
141	反向邻居发现请求报文	反向邻居发现请求：请求给定链路层地址的 IPv6 地址
142	反向邻居发现通告报文	反向邻居发现应答：报告给定的链路层地址的 IPv6 地址
143	组播侦听报告版本 2	组播侦听报告（v2）
144	本地代理地址发现请求报文	请求移动 IPv6 HA 地址，由移动节点发送
145	本地代理地址发现应答报文	包含 MIPv6 HA 地址，在本地网络中由合格的 HA 发送
146	移动前缀请求	当离开时请求本地前缀
147	移动前缀通告	提供从 HA 到移动节点的前缀
148	证书路径请求报文	一条证书路径的保护邻居发现 SEND 请求
149	证书路径通告报文	响应一个证书路径请求的 SEND
151	组播路由器通告	提供组播路由器的地址
152	组播路由器请求	请求组播路由器的地址
153	组播路由器终止	组播路由器使用结束
154	FMIPv6 报文	MIPv6 快速切换报文
200、201	为私人实验保留	为实验保留
255	为ICMPv6信息类报文扩充保留	为更多的信息类报文保留

ICMPv6 中也使用代码（Code）字段，主要是为了完善某些差错报文的含义。在下表中列出了这些标准的 ICMPv6 报文类型（即目的不可达、超时、参数问题），除 0 之外还定义了许多代码值。

类型	代码	名称	用途/注释
1	0	没有到目的地的路由	路由不存在
1	1	管理禁止	策略禁止（如防火墙）
1	2	超出源地址范围	目的范围超出源地址的范围
1	3	地址不可达	当代码 0~2 并不合适时使用
1	4	端口不可达	没有传输层实体在端口监听
1	5	源地址失败策略	违反进/出策略
1	6	拒绝到目的地的路由	特定的拒绝到目的地的路由
3	0	在传输中超过了跳数限制	跳数限制字段值递减为 0

3	1	重组时间超时	在有限的时间内无法重组
4	0	找到错误的头部字段	一般的头部处理错误
4	1	无法识别的下一个头部	未知的下一个头部字段值
4	2	无法识别的 IPv6 选项	未知的“逐跳”或者“目的地”选项

■邻居发现协议

下面简要介绍一下 ICMPv6 中邻居发现协议的实现，其它类型的 ICMPv6 报文请读者参阅相关资料。

IPv6 中的邻居发现协议（NDP 或 ND）将路由器发现和由 ARP 提供的带有地址映射功能的 ICMPv4 重定向机制结合在一起。与 ARP 和 IPv4 普遍使用广播地址不同，ICMPv6 广泛使用组播地址，在网络层和链路层中都使用。

ND 中两个主要部分是：邻居请求/通告（NS/NA），在网络和链路层地址之间提供类似于 ARP 的映射功能；还有路由器请求和通告（RS/RA），提供的功能包括路由器发现、移动 IP 代理发现、重定向，以及一些自动配置的支持。

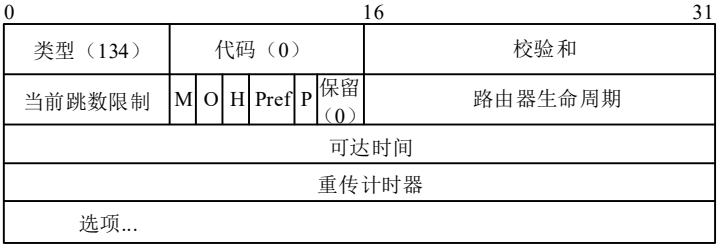
ND 报文就是 ICMPv6 报文，只是发送时 IPv6 的跳数限制字段值设为 255。ND 报文可以携带丰富的选项。下面先讨论报文类型，然后介绍几个选项。

●路由器请求与通告

路由器通告（RA）报文表明附近路由器的存在及其功能。它们定期被路由器发送，或者是响应一个路由器请求（RS）报文。RS 报文用于请求链路上的路由器发送 RA 报文。RS 报文被发送到所有路由器组播地址 ff02::2。如果报文的发送者使用 IPv6 地址，而不是未指定的地址，则应该包括一个源链路层地址选项（选项格式与内容见下文）。对于这样的报文，这是唯一有效的选项。路由器请求（RS）报文结构如下图所示：

ICMPv6路由器请求报文

路由器通告（RA）报文是由路由器发送到所有节点的组播地址 ff02::1 的，或者是发送到请求主机的单播地址（如果通告是为了响应一个请求）。RA 报文通知本地主机和其他路由器关于本地链路的有关配置细节。路由器通告（RA）报文结构如下图所示：



ICMPv6路由器通告报文

当前跳数限制字段指定主机发送 IPv6 数据报的默认跳数限制。值为 0 表示发送路由器并不关心。下一字节包含一个位字段数。M（托管）字段表明本地 IPv6 地址分配是由有状态的配置来处理的，主机应避免使用无状态的自动配置。O（其它）字段表示其他有状态的信息使用一个有状态的配置机制。H（本地代理）字段表示发送路由器愿意充当一个移动 IPv6 节点的本地代理。Pref（优先级）字段给出了将报文发送者作为一个默认路由器来使用的优先级层次：01，高；00，中（默认）；11，低；10，保留（未使用）。当和实验性质的 ND 代理工具配合使用时，将使用 P（代理）标志。它为 IPv6 提供了一个类似代理 ARP

的功能。

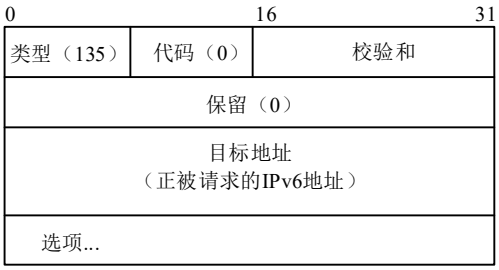
路由器生命周期字段表示发送路由器可以作为默认下一跳的时间，以秒计。如果它被设置为 0，发送路由器不应该用作默认路由器。此字段只适用于使用发送路由器作为默认路由器，它不会影响同一个报文中的其它选项。可达时间字段给出一个节点到达另一个节点所需的毫秒数，假设已经发生了双向通信。这被邻居不可达检测机制使用。重传计时器字段规定主机延迟发送连续 ND 报文的时间，以毫秒为单位。

此报文通常包含源链路层选项，如果链路中使用了可变长度的 MTU 则应包含 MTU 选项。该路由器还应该包括前缀信息选项，表示本地链路上使用了哪些 IPv6 前缀。

●邻居请求和通告

ICMPv6 中的邻居请求（NS）报文，有效地取代了 IPv4 中的 ARP 请求报文。其主要目的是将 IPv6 地址转换成链路层地址。但是，它也被用于检测附近的节点是否可达，它们是否可以双向到达。当用于确定地址映射时，它被发送到目标地址字段中包含的 IPv6 地址所对应的请求节点的组播地址。当这个报文被用来确定到邻居的连接性时，它被发送到该邻居的 IPv6 单播地址，而不是请求节点的地址。

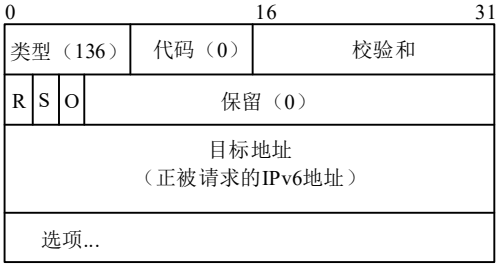
NS 报文包含发送者想设法学习的链路层地址对应的 IPv6 地址。该报文可能包含源链路层地址选项。当请求是被发送到一个组播地址时，该选项必须包含在使用链路层寻址的网络中，对于单播请求而言，该选项应该被包含。如果报文的发送者使用未指定的地址作为源地址，则不应该包括该选项。ICMPv6 中邻居请求（NS）报文结构如下图所示：



ICMPv6邻居请求报文

IPv6 邻居通告（NA）报文和 IPv4 中的 ARP 响应报文的目的一样，还能够有助于邻居不可达检测。它要么作为 NS 报文的响应被发送，要么当一个节点 IPv6 地址变化时被异步发送。它要么被发送到请求节点的单播地址，要么当请求节点使用未指定的地址作为源地址时，它被发送到所有节点的组播地址。

IPv6 邻居通告（NA）报文结构如下图所示：



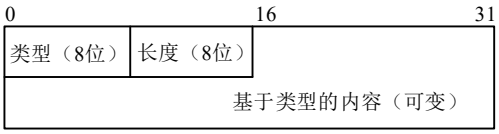
ICMPv6邻居通告报文

R（路由器）字段表示该报文的发送者是一个路由器。这可能会改变，例如当一台路由器不再是路由器，而成为一台主机时。S（请求）字段表示该报文是在响应先前收到的请求。这个字段用来验证已经取得的邻居之间的双向连通性。O（覆盖）字段表示在报文中的信息应覆盖报文发送者之前缓存的任何信息。它不应该在请求通告、任播地址或请求代理通告中设置，而应该在其它通告中设置。

对于请求通告，目标地址字段就是正在被查找的 IPv6 地址。对于主动通告，它是已经改变的链路层地址对应的 IPv6 地址。当通告是通过一个组播地址被请求时，此报文必须包含支持链路层寻址的网络的目标链路层地址。

●邻居发现选项

正如 IPv6 家族的许多协议，它定义了一套标准协议头部，还包含一个或多个选项。ND（邻居发现）报文可能包含零个或多个选项，一些选项可以出现多次。但是，对于某些报文而言，这些选项是必需的。下图给出了 ND 选项的通用格式：



ND选项

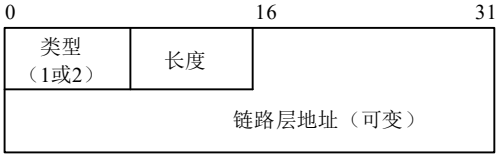
所有 ND 选项以 8 位的类型和 8 位长度字段开始，支持长度可变的选项，最大到 255 字节。选项被填充以形式 8 字节边界，长度字段给出了选项总长度，以 8 字节为单位。类型和长度字段包含在长度字段的值中，最小值为 1。下表列出了在 2011 年年中定义的 25 个标准选项：

类型	名称	用途/注释
1	源链路层地址	发送者的链路层地址；与 NS、RS 及 RA 报文一起使用
2	目标链路层地址	目标链路层地址；与 NA 及定向报文一起使用
3	前缀信息	一个 IPv6 前缀或者地址；与 RA 报文一起使用
4	被重定向的头部	原始 IPv6 报文的部分；与重定向报文一起使用
5	MTU	推荐的 MTU；与 RA 报文、IND 通告报文一起使用
6	NMBA 捷径限制	“捷径尝试”的跳数限制；与 NS 报文一起使用
7	通告间隔	主动 RA 报文的发送间隔；与 RA 报文一起使用
8	本地代理信息	成为一个 MIPv6 HA 的优先级和生命周期；与 RA 报文一起使用（设置 H 位）
9	源地址列表	主机地址；与 IND 报文一起使用
10	目标地址列表	目标地址；与 IND 报文一起使用
11	CGA	基于密码的地址；与安全邻居发现报文（SEND）一起使用
12	RSA 签名	主机签名的证书（SEND）
13	时间戳	反重放时间戳（SEND）
14	随机数	反重随机数（SEND）
15	信任锚	指示证书类型（SEND）
16	证书	编码一个证书（SEND）
17	IP 地址/前缀	移交或者 NAR 地址；与 FMIPv6 PrRtAdv 报文一起使用
19	链路层地址	想要的下一个接入点或者移动节点的地址；与 FMIPv6 RtSolPt 或者 PrRtAdv 报文一起使用
20	邻居通告确认	告诉移动节点下一个有效的 CoA；与 RA 报文一起使用
24	路由信息	路由前缀/首选的路由器列表
25	递归 DNS 服务器	DNS 服务器的 IP 地址；添加到 RA 报文
26	RA 标志扩展	扩展 RA 标志的空间
27	切换密钥请求	FMIPv6--使用 SEND 请求密钥
28	切换密钥应答	FMIPv6--使用 SEND 应答密钥
31	DNS 搜索列表	DNS 域搜索名称；添加到 RA 报文中
253、254	实验性	实验用

下面列举几个典型选项的结构：

（1）源/目标链路层地址选项（类型 1、2）

每当在一个支持链路层选址的网络中使用时，源链路层地址选项（类型 1，如下图所示）就应该被包含在 ICMPv6 RS（路由器请求）报文、NS（邻居请求）报文和 RA（路由器通告）报文中。它指定了一个和报文相关的链路层地址。对于含有多个地址的节点可能包含上述的多个选项。

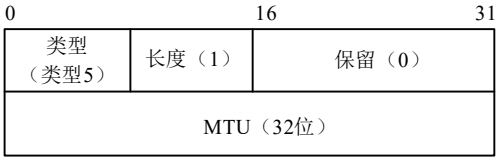


源（1）和目标（2）链路层地址选项

当响应一个组播请求时，采用类似格式的目标链路层地址选项必段包含在 NA（邻居通告）报文中。这个选项通常包含在重定向报文中，但当在一个 NBMA 网络上操作时则必须被包含在这样的报文中。

（2）MTU 选项（类型 5）

MTU 选项仅在 RA 报文中提供，在其它地方被忽略。它提供了主机使用的 MTU，假设能够支持一个可配置的 MTU。MTU 选项结构如下图所示：



ND选项

MTU 选项非常重要，在这个选项中保留了 32 个比特位来存储 MTU 值，支持非常大的 MTU。

13.3.2 传输层协议实现

IPv6 中传输层协议与 IPv4 中传输层协议类似，主要是 ICMPv6 相对于 ICMPv4 被赋予了更多的功能，前面已经介绍过了。

在 Linux 内核中用户进程同样通过套接字访问 IPv6 网络，本小节简要介绍各传输层协议对应套接字的创建、操作等，其操作中包含传输层协议的实现。

1 概述

下在将介绍 IPv6 套接字的创建、实现框架，以及内核相关的初始化工作等。

■创建套接字

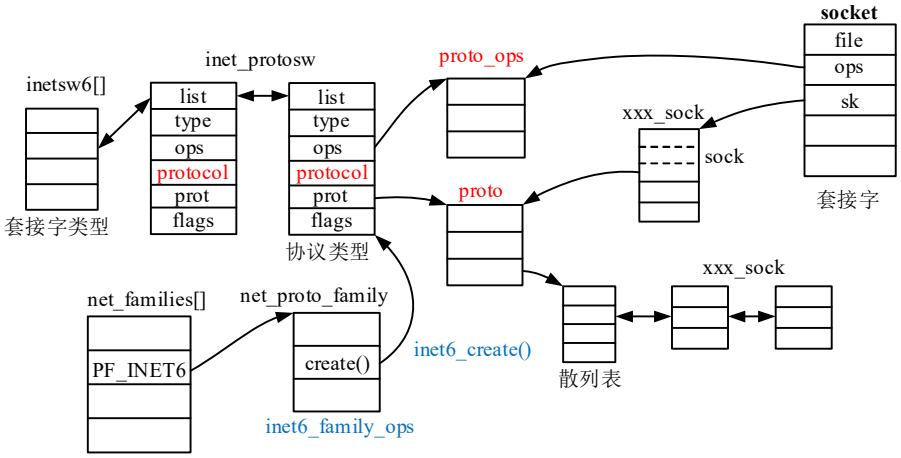
用户创建 IPv6 套接字的方法与创建 IPv4 套接字类似。用户进程创建 IPv6 套接字的系统调用如下所示，IPv6 地址簇标识为 AF_INET6（PF_INET6）。

socket(AF_INET6,SOCK_DGRAM,IPPROTO_UDP);　　/*创建 UDP 套接字*/

内核在/net/ipv6/af_inet6.c 文件内定义了全局双链表数组 inetsw6[SOCK_MAX]，每个链表对应一个套接字类型，用于管理 inet_protosw 实例，如下图所示。inet_protosw 实例由传输层协议定义并注册到双链表数组，其中包含特定传输层协议套接字关联的 proto_ops 和 proto 实例信息等。

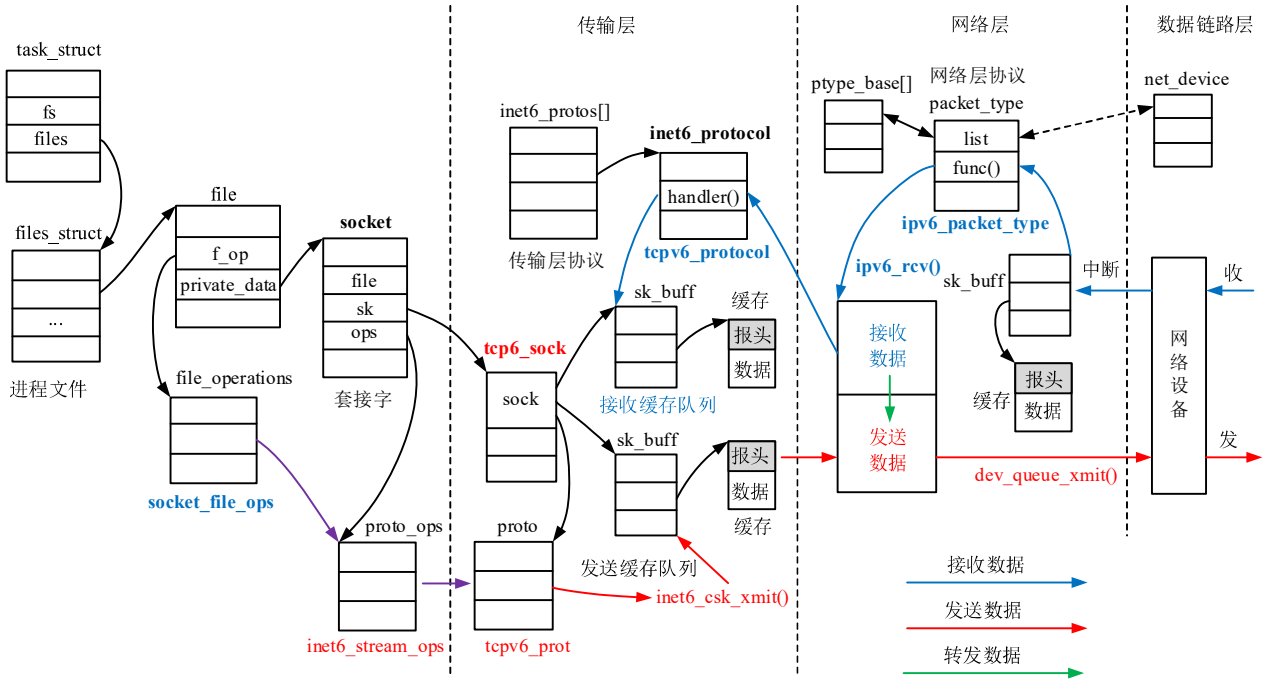
内核在/net/ipv6/af_inet6.c 文件内定义了 IPv6（AF_INET6）对应的 net_proto_family 实例 inet6_family_ops。在创建套接字时，将调用其中的 create()函数设置 socket 实例。inet6_family_ops 实例中此函数为 inet6_create()。

inet6_create()函数将根据套接字类型、传输层协议查找对应的 inet_protosw 实例，依实例指定的 proto 实例创建并设置表示套接字的 xxx_sock 实例，将 inet_protosw 实例指定的 proto_ops 实例赋予 socket 实例等。



■协议实现框架

IPv6 协议实现与 IPv4 协议实现类似，实现框架如下图所示（TCP 套接字）：



上图中所示的是 TCP 套接字，套接字关联的 proto_ops 实例为 inet6_stream_ops，proto 实例为 tcpv6_prot。tcp6_sock 结构体用于在传输层中表示套接字。

传输层协议需要定义并注册 inet6_protocol 结构体实例，其中的处理函数成员用于接收网络层传递的数据包，TCP 协议此实例为 **tcpv6_protocol**，其中接收数据包函数为 **tcp_v6_rcv()**。

IPv6 定义并注册的 packet_type 实例为 **ipv6_packet_type**，其中接收函数为 **ipv6_rcv()**，用于接收数据链路层上传的数据包。

传输层协议中发送、接收数据包的流程与 IPv4 中相同，IPv6 网络层中发送、接收数据包流程与 IPv4 也类似，下一小节将简要介绍。

■初始化

IPv6 协议实现初始化函数为 inet6_init()，定义如下（/net/ipv6/af_inet6.c）：

```
static int __init inet6_init(void)
```

```
{
    struct list_head *r;
    int err = 0;

    sock_skb_cb_check_size(sizeof(struct inet6_skb_parm));

    for (r = &inetsw6[0]; r < &inetsw6[SOCK_MAX]; ++r)
        INIT_LIST_HEAD(r);                  /*初始化 inetsw6[]双链表数组*/
    ...
    err = proto_register(&tcpv6_prot, 1);       /*注册 TCP 套接字关联的 proto 实例*/
    ...
    err = proto_register(&udpv6_prot, 1);       /*注册 UDP 套接字关联的 proto 实例*/
    ...
    err = proto_register(&udplitev6_prot, 1);   /*注册 UDP-Lite 套接字关联的 proto 实例*/
    ...
    err = proto_register(&rawv6_prot, 1);       /*注册原始套接字关联的 proto 实例*/
    ...
    err = proto_register(&pingv6_prot, 1);      /*注册 ping 套接字关联的 proto 实例*/
    ...
    err = rawv6_init();       /*原始套接字初始化，注册对应的 inet_protosw 实例等，/net/ipv6/raw.c*/
    ...
    err = sock_register(&inet6_family_ops);     /*注册 IPv6 对应的 net_proto_family 实例实例*/
    ...
    err = register_pernet_subsys(&inet6_net_ops);
        /*注册 pernet_operations 实例，其初始化函数用于设置网络命名空间中 IPv6 系统控制参数，
          *完成 UDP、TCP 在 proc 文件系统中的初始化等，/net/ipv6/af_inet6.c。*/
    ...
    err = icmpv6_init();      /*ICMPv6 初始化，/net/ipv6/icmp.c*/
    ...
    err = ip6_mr_init();
          /*组播路由初始化，没有选择 IPV6_MROUTE 配置项为空操作，/net/ipv6/ip6mr.c*/
    ...
    err = ndisc_init();     /*邻居子系统初始化等，/net/ipv6/ndisc.c*/
    ...
    err = igmp6_init();     /*组播管理初始化（使用 ICMPv6），/net/ipv6/mcast.c*/
    ...
    ipv6_stub = &ipv6_stub_impl;
    err = ipv6_netfilter_init();        /*Netfilter 子系统初始化，/net/ipv6/netfilter.c*/
    ...
#ifdef CONFIG_PROC_FS
    ...
#endif
    err = ip6_route_init();       /*路由选择子系统初始化，/net/ipv6/route.c*/
    ...
    err = ndisc_late_init();      /*邻居发现协议初始化，/net/ipv6/ndisc.c*/
    ...
```

```
    err = ip6_flowlabel_init();      /*/net/ipv6/ip6_flowlabel.c*/
    ...
    err = addrconf_init();      /*网络设备配置初始化（IPv6 中配置），/net/ipv6/addrconf.c*/
    ...
    err = ipv6_exthdrs_init();    /*扩展报头处理初始化，/net/ipv6/exthdrs.c*/
    ...
    err = ipv6_frag_init();      /*IPv6 分片重组初始化，/net/ipv6/reassembly.c*/
    ...
    err = udpv6_init();        /*UDPv6 初始化，/net/ipv6/udp.c*/
    ...
    err = udplitev6_init();    /*UDP_Litev6 初始化，/net/ipv6/udplite.c*/
    ...
    err = tcpv6_init();      /*TCPv6 初始化，/net/ipv6/tcp_ipv6.c*/
    ...
    err = ipv6_packet_init();    /*注册 ipv6_packet_type 实例，网络层接收数据包，/net/ipv6/af_inet6.c*/
    ...
    err = pingv6_init();      /*ping 套接字初始化，/net/ipv6/ping.c*/
    ...
#ifdef CONFIG_SYSCTL
    err = ipv6_sysctl_register();    /*注册系统控制参数，/net/ipv6/sysctl_net_ipv6.c*/
    ...
#endif
out:
    return err;
     ...    /*错误处理*/
}
module_init(inet6_init);    /*IPv6 实现可配置为模块*/
```

■地址表示

IPv6 地址由 128 个比特位表示，在 Linux 内核中 IPv6 地址由 in6_addr 结构体表示，定义如下：

```
struct in6_addr {          /*/include/uapi/linux/in6.h*/
    union {
        __u8      u6_addr8[16];       /*16 个字节*/
    #if __UAPI_DEF_IN6_ADDR_ALT
        __be16    u6_addr16[8];       /*8 个半字*/
        __be32    u6_addr32[4];       /*4 个字*/
    #endif
    } in6_u;
    #define   s6_addr        in6_u.u6_addr8
    #if __UAPI_DEF_IN6_ADDR_ALT
      #define   s6_addr16    in6_u.u6_addr16
      #define   s6_addr32    in6_u.u6_addr32
    #endif
};
```

in6_addr 结构体中主要包含一个联合体，即 128 位的 IPv6 地址可用 16 个字节、8 个半字或 4 个 32 位字表示。

在传输层中通过端口号来寻址套接字，因此某一主机中的套接字需要由一个 IPv6 地址和一个端口号标识，这里称之为套接字地址。套接字地址由 **sockaddr_in6** 结构体表示，定义如下（/include/uapi/linux/in6.h）：

```
struct sockaddr_in6 {
    unsigned short int  sin6_family;    /* AF_INET6，地址簇标识 */
    __be16              sin6_port;      /*端口号*/
    __be32              sin6_flowinfo;      /* IPv6 flow information */
    struct in6_addr     sin6_addr;          /* IPv6 地址*/
    __u32               sin6_scope_id;      /* scope id (new in RFC2553) */
};
```

2 UDP 实现

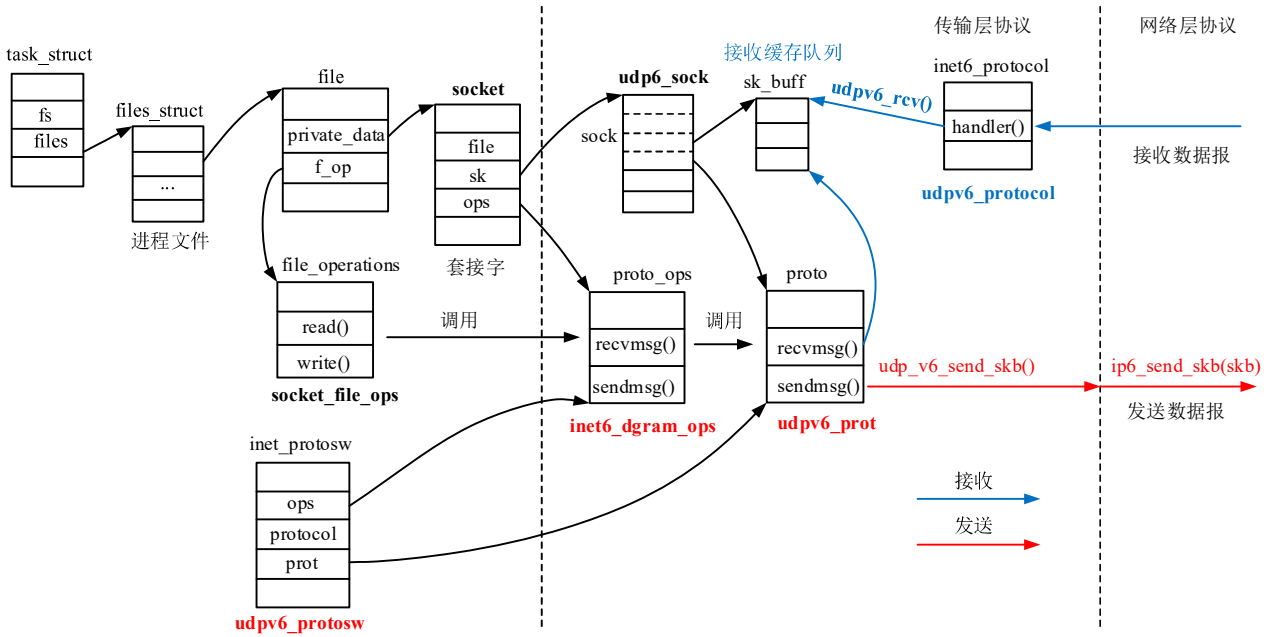
UDP 在前面介绍过了，下面简要介绍一下其在 IPv6 中的实现，实现代码位于/net/ipv6/udp.c 文件内。

■概述

在 IPv6 中 UDP 实现框架如下图所示，与在 IPv4 中的实现类似。UDP 套接字由 udp6_sock 结构体实例表示，关联的 proto_ops 实例为 **inet6_dgram_ops**，关联的 proto 实例为 **udpv6_prot**，这两个实例中的操作函数与 UDPv4 中的操作函数类似，有些是共用的。

UDP 定义并注册了 inet6_protocol 结构体实例 **udpv6_protocol**，其中的接收函数（udpv6_rcv()）用于接收网络层传递的 UDP 数据报。用户读取套接字数据时，从接收缓存队列中获取数据报，读取其中的用户数据。

用户发送 UDP 数据报与 UDPv4 中发送数据报操作类似，如果没有启用抑制或合并功能，则用户的每次写操作都将生成一个数据报，并立即发送。如果启用了抑制或合并功能，则将用户多次写操作合并成一个数据报进行发送。



UDP 初始化函数为 udpv6_init()，定义如下（/net/ipv6/udp.c）：

```
int __init udpv6_init(void)
{
    int ret;

    ret = inet6_add_protocol(&udpv6_protocol, IPPROTO_UDP);      /*注册 inet6_protocol 实例*/
    ...
    ret = inet6_register_protosw(&udpv6_protosw);      /*注册 inet_protosw 实例*/
    ...
out:
    return ret;
    ...
}
```

初始化函数中注册了 inet6_protocol 实例 udpv6_protocol，用于接收网络层数据报，定义如下：

```
static const struct inet6_protocol udpv6_protocol = {
    .handler  =    udpv6_rcv,          /*接收数据包函数*/
    .err_handler  =    udpv6_err,
    .flags        =    INET6_PROTO_NOPOLICY|INET6_PROTO_FINAL,
};
```

初始化函数中注册了 inet_protosw 实例 udpv6_protosw ，用于创建 UDP 套接字，定义如下：

```
static struct inet_protosw udpv6_protosw = {
    .type =       SOCK_DGRAM,
    .protocol =   IPPROTO_UDP,
    .prot =       &udpv6_prot,        /*proto 实例*/
    .ops =        &inet6_dgram_ops,       /*proto_ops 实例*/
    .flags =      INET_PROTOSW_PERMANENT,
};
```

■套接字操作

UDP 套接字操作结构 inet6_dgram_ops 实例定义如下（/net/ipv6/af_inet6.c）：

```
const struct proto_ops inet6_dgram_ops = {
    .family       = PF_INET6,
    .owner        = THIS_MODULE,
    .release      = inet6_release,    /*/net/ipv6/af_inet6.c*/
    .bind         = inet6_bind,    /*绑定操作，/net/ipv6/af_inet6.c*/
    .connect      = inet_dgram_connect,  /*连接操作，/net/ipv6/af_inet6.c*/
    .socketpair    = sock_no_socketpair,
    .accept        = sock_no_accept,
    .getname      = inet6_getname,
    .poll         = udp_poll,
    .ioctl            = inet6_ioctl,        /*/net/ipv6/af_inet6.c*/
    .listen           = sock_no_listen,
    .shutdown         = inet_shutdown,
```

```
    .setsockopt         = sock_common_setsockopt,
    .getsockopt         = sock_common_getsockopt,
    .sendmsg        = inet_sendmsg,
    .recvmsg        = inet_recvmsg,
    .mmap           = sock_no_mmap,
    .sendpage       = sock_no_sendpage,
#ifdef CONFIG_COMPAT
    .compat_setsockopt = compat_sock_common_setsockopt,
    .compat_getsockopt = compat_sock_common_getsockopt,
#endif
};
```

inet6_dgram_ops 实例中的操作函数大部分与 UDPv4 操作结构 inet_dgram_ops 实例中的操作函数相同。IPv6 专用的函数定义在/net/ipv6/af_inet6.c 文件内，源代码请读者自行阅读。

套接字操作 proto_ops 结构体中的函数将调用 proto 结构体中的函数，执行特定于传输层协议的操作。UDP 关联的 proto 实例为 **udpv6_prot**，定义如下（/net/ipv6/udp.c）：

```
struct proto udpv6_prot = {
    .name               = "UDPv6",
    .owner              = THIS_MODULE,
    .close              = udp_lib_close,
    .connect        = ip6_datagram_connect,   /*连接操作，设置目的套接字地址，/net/ipv6/datagram.c*/
    .disconnect         = udp_disconnect,
    .ioctl              = udp_ioctl,
    .destroy        = udpv6_destroy_sock,
    .setsockopt         = udpv6_setsockopt,       /*设置参数值*/
    .getsockopt         = udpv6_getsockopt,       /*获取参数值*/
    .sendmsg        = udpv6_sendmsg,          /*发送数据，/net/ipv6/udp.c*/
    .recvmsg        = udpv6_recvmsg,          /*读取数据，/net/ipv6/udp.c*/
    .backlog_rcv        = __udpv6_queue_rcv_skb,
    .hash               = udp_lib_hash,
    .unhash             = udp_lib_unhash,
    .rehash             = udp_v6_rehash,
    .get_port       = udp_v6_get_port,        /*分配或设置端口号，/net/ipv6/udp.c*/
    .memory_allocated   = &udp_memory_allocated,
    .sysctl_mem         = sysctl_udp_mem,
    .sysctl_wmem        = &sysctl_udp_wmem_min,
    .sysctl_rmem        = &sysctl_udp_rmem_min,
    .obj_size       = sizeof(struct udp6_sock),
    .slab_flags         = SLAB_DESTROY_BY_RCU,
    .h.udp_table        = &udp_table,     /*同 UDPv4 中实例，同时管理 UDPv4 和 UDPv6 套接字*/
#ifdef CONFIG_COMPAT
    .compat_setsockopt = compat_udpv6_setsockopt,
    .compat_getsockopt = compat_udpv6_getsockopt,
#endif
    .clear_sk       = udp_v6_clear_sk,
```

```
};
```

由 udpv6_prot 实例可知，在 UDP 实现中，套接字由 udp6_sock 结构体表示，其实例由 udp_table 散列表管理（同 UDPv4 中散列表）。

udp6_sock 结构体定义如下（/include/linux/ipv6.h）：

```
struct udp6_sock {
    struct udp_sock      udp;      /*UDPv4 套接字表示*/
    struct ipv6_pinfo    inet6;    /*IPv6 中信息*/
};
```

udpv6_prot 实例中的操作函数与 udp_prot 实例中的操作函数类似，下面简要介绍一下发送和读取数据函数的实现，其它函数请读者自行阅读源代码。

●发送数据

udpv6_prot 实例中 udpv6_sendmsg()函数用于发送用户数据，函数代码简列如下（/net/ipv6/udp.c）：

```
int udpv6_sendmsg(struct sock *sk, struct msghdr *msg, size_t len)
{
    struct ipv6_txoptions opt_space;
    struct udp_sock *up = udp_sk(sk);
    struct inet_sock *inet = inet_sk(sk);
    struct ipv6_pinfo *np = inet6_sk(sk);
    DECLARE_SOCKADDR(struct sockaddr_in6 *, sin6, msg->msg_name);      /*目的套接字地址*/
    struct in6_addr *daddr, *final_p, final;      /*IPv6 地址*/
    struct ipv6_txoptions *opt = NULL;
    struct ip6_flowlabel *flowlabel = NULL;
    struct flowi6 fl6;      /*保存路由选择参数*/
    struct dst_entry *dst;
    int addr_len = msg->msg_namelen;
    int ulen = len;
    int hlimit = -1;
    int tclass = -1;
    int dontfrag = -1;
    int corkreq = up->corkflag || msg->msg_flags&MSG_MORE;      /*是否启用抑制或合并功能*/
    int err;
    int connected = 0;
    int is_udplite = IS_UDPLITE(sk);      /*是否是 UDP-Litev6*/
    int (*getfrag)(void *, char *, int, int, int, struct sk_buff *);      /*复制用户数据至数据包的函数指针*/

    /*检查目的地址*/
    if (sin6) {
        ...
    } else if (!up->pending) {
        if (sk->sk_state != TCP_ESTABLISHED)
            return -EDESTADDRREQ;
        daddr = &sk->sk_v6_daddr;
    } else
```

```
        daddr = NULL;

if (daddr) {
        if (ipv6_addr_v4mapped(daddr)) {
                ...         /*IPv4 映射的 IPv6 地址*/
        }
}

if (up->pending == AF_INET)          /*UDPv4*/
        return udp_sendmsg(sk, msg, len);

if (len > INT_MAX - sizeof(struct udphdr))
        return -EMSGSIZE;

getfrag  =  is_udplite ?  udplite_getfrag : ip_generic_getfrag;
if (up->pending) {              /*有挂起数据*/
        lock_sock(sk);
        if (likely(up->pending)) {
                if (unlikely(up->pending != AF_INET6)) {
                        release_sock(sk);
                        return -EAFNOSUPPORT;
                }
                dst = NULL;
                goto do_append_data;        /*合并数据*/
        }
        release_sock(sk);
}

/*没有挂起数据*/
ulen += sizeof(struct udphdr);
memset(&fl6, 0, sizeof(fl6));

if (sin6) {       /*指定的目的地址*/
        if (sin6->sin6_port == 0)
                return -EINVAL;

        fl6.fl6_dport = sin6->sin6_port;
        daddr = &sin6->sin6_addr;

        if (np->sndflow) {
                fl6.flowlabel = sin6->sin6_flowinfo&IPV6_FLOWINFO_MASK;
                if (fl6.flowlabel&IPV6_FLOWLABEL_MASK) {
                        flowlabel = fl6_sock_lookup(sk, fl6.flowlabel);
                        if (!flowlabel)
                                return -EINVAL;
                }
```

```
        }

        if (sk->sk_state == TCP_ESTABLISHED &&ipv6_addr_equal(daddr, &sk->sk_v6_daddr))
            daddr = &sk->sk_v6_daddr;

        if (addr_len >= sizeof(struct sockaddr_in6) &&sin6->sin6_scope_id &&
            __ipv6_addr_needs_scope_id(__ipv6_addr_type(daddr)))
            fl6.flowi6_oif = sin6->sin6_scope_id;
    } else {     /*未指定目的地址，sin6 为 NULL，使用连接操作中设置的地址*/
        if (sk->sk_state != TCP_ESTABLISHED)
            return -EDESTADDRREQ;

        fl6.fl6_dport = inet->inet_dport;
        daddr = &sk->sk_v6_daddr;
        fl6.flowlabel = np->flow_label;
        connected = 1;
    }

    if (!fl6.flowi6_oif)
        fl6.flowi6_oif = sk->sk_bound_dev_if;

    if (!fl6.flowi6_oif)
        fl6.flowi6_oif = np->sticky_pktinfo.ipi6_ifindex;

    fl6.flowi6_mark = sk->sk_mark;

    if (msg->msg_controllen) {            /*处理控制消息*/
        ...
    }
    if (!opt)
        opt = np->opt;
    if (flowlabel)
        opt = fl6_merge_options(&opt_space, flowlabel, opt);
    opt = ipv6_fixup_options(&opt_space, opt);

    fl6.flowi6_proto = sk->sk_protocol;
    if (!ipv6_addr_any(daddr))
        fl6.daddr = *daddr;
    else
        fl6.daddr.s6_addr[15] = 0x1;        /*环回地址*/
    if (ipv6_addr_any(&fl6.saddr) && !ipv6_addr_any(&np->saddr))
        fl6.saddr = np->saddr;
    fl6.fl6_sport = inet->inet_sport;

    final_p = fl6_update_dst(&fl6, opt, &final);
    if (final_p)
```

```
            connected = 0;

        if (!fl6.flowi6_oif && ipv6_addr_is_multicast(&fl6.daddr)) {
            fl6.flowi6_oif = np->mcast_oif;
            connected = 0;
        } else if (!fl6.flowi6_oif)
            fl6.flowi6_oif = np->ucast_oif;

        security_sk_classify_flow(sk, flowi6_to_flowi(&fl6));

        dst = ip6_sk_dst_lookup_flow(sk, &fl6, final_p);       /*路由选择查找，/net/ipv6/ip6_output.c*/
        ...

        if (hlimit < 0)
            hlimit = ip6_sk_dst_hoplimit(np, &fl6, dst);

        if (tclass < 0)
            tclass = np->tclass;

        if (msg->msg_flags&MSG_CONFIRM)
            goto do_confirm;
back_from_confirm:

        if (!corkreq) {        /*没有启用抑制（或合并）功能*/
            struct sk_buff *skb;

            skb = ip6_make_skb(sk, getfrag, msg, ulen,sizeof(struct udphdr), hlimit, tclass, opt,
                        &fl6, (struct rt6_info *)dst,msg->msg_flags, dontfrag);
                                                    /*生成数据包，/net/ipv6/ip6_output.c*/
            err = PTR_ERR(skb);
            if (!IS_ERR_OR_NULL(skb))
                err = udp_v6_send_skb(skb, &fl6); /*调用 ip6_send_skb(skb)发送数据包，/net/ipv6/udp.c*/
            goto release_dst;
        }

        /*启用了抑制（或合并）功能*/
        lock_sock(sk);
        if (unlikely(up->pending)) {
            ...
        }

        up->pending = AF_INET6;

do_append_data:       /*合并数据后，再发送*/
        if (dontfrag < 0)
            dontfrag = np->dontfrag;
```

```
        up->len += ulen;
        err = ip6_append_data(sk, getfrag, msg, ulen,
              sizeof(struct udphdr), hlimit, tclass, opt, &fl6,struct rt6_info *)dst,
              corkreq ? msg->msg_flags|MSG_MORE : msg->msg_flags, dontfrag);
                                                        /*合并数据，/net/ipv6/ip6_output.c*/
        if (err)
              udp_v6_flush_pending_frames(sk);
        else if (!corkreq)
              err = udp_v6_push_pending_frames(sk);   /*发送数据报，/net/ipv6/udp.c*/
        else if (unlikely(skb_queue_empty(&sk->sk_write_queue)))
              up->pending = 0;

        if (err > 0)
              err = np->recverr ? net_xmit_errno(err) : 0;
        release_sock(sk);

    release_dst:
        if (dst) {
              if (connected) {
                    ip6_dst_store(sk, dst,ipv6_addr_equal(&fl6.daddr, &sk->sk_v6_daddr) ?
                                &sk->sk_v6_daddr : NULL,
              #ifdef CONFIG_IPV6_SUBTREES
                                ipv6_addr_equal(&fl6.saddr, &np->saddr) ?
                                &np->saddr :
              #endif
                                NULL);
              } else {
                    dst_release(dst);
              }
              dst = NULL;
        }

    out:
        dst_release(dst);
        fl6_sock_release(flowlabel);
        if (!err)
              return len;         /*返回发送数据长度*/
        ...
    }
```

UDPv6 中发送数据操作与 UDPv4 中发送数据操作类似，请读者自行阅读源代码。

●读取数据

读取数据是指用户进程从 UDPv6 套接字接收缓存队列中读取数据报中的数据至用户空间，后面将介绍 UDPv6 从网络层接收数据报至套接字接收缓存队列的操作。

udpv6_prot 实例中 **udpv6_recvmsg()**函数用于从套接字接收缓存队列中读取数据至用户空间，函数定义如下（/net/ipv6/udp.c）：

```
int udpv6_recvmsg(struct sock *sk, struct msghdr *msg, size_t len,int noblock, int flags, int *addr_len)
{
    struct ipv6_pinfo *np = inet6_sk(sk);
    struct inet_sock *inet = inet_sk(sk);
    struct sk_buff *skb;
    unsigned int ulen, copied;
    int peeked, off = 0;
    int err;
    int is_udplite = IS_UDPLITE(sk);
    int is_udp4;
    bool slow;

    if (flags & MSG_ERRQUEUE)
        return ipv6_recv_error(sk, msg, len, addr_len);

    if (np->rxpmtu && np->rxopt.bits.rxpmtu)
        return ipv6_recv_rxpmtu(sk, msg, len, addr_len);

try_again:
    skb = __skb_recv_datagram(sk, flags | (noblock ? MSG_DONTWAIT : 0),&peeked, &off, &err);
                                                        /*获取数据报，/net/core/datagram.c*/
    ...
    ulen = skb->len - sizeof(struct udphdr);
    copied = len;
    if (copied > ulen)
        copied = ulen;
    else if (copied < ulen)
        msg->msg_flags |= MSG_TRUNC;

    is_udp4 = (skb->protocol == htons(ETH_P_IP));

    if (copied < ulen || UDP_SKB_CB(skb)->partial_cov) {
        if (udp_lib_checksum_complete(skb))
            goto csum_copy_err;
    }

    /*复制数据报数据至用户空间*/
    if (skb_csum_unnecessary(skb))
        err = skb_copy_datagram_msg(skb, sizeof(struct udphdr),msg, copied);
    else {
        err = skb_copy_and_csum_datagram_msg(skb, sizeof(struct udphdr), msg);
        ...
    }
    ...
    sock_recv_ts_and_drops(msg, sk, skb);
```

```
        /*复制数据报源地址（发送方地址）至用户空间*/
        if (msg->msg_name) {
            DECLARE_SOCKADDR(struct sockaddr_in6 *, sin6, msg->msg_name);
            sin6->sin6_family = AF_INET6;
            sin6->sin6_port = udp_hdr(skb)->source;
            sin6->sin6_flowinfo = 0;

            if (is_udp4) {
                ipv6_addr_set_v4mapped(ip_hdr(skb)->saddr,&sin6->sin6_addr);
                sin6->sin6_scope_id = 0;
            } else {
                sin6->sin6_addr = ipv6_hdr(skb)->saddr;
                sin6->sin6_scope_id =ipv6_iface_scope_id(&sin6->sin6_addr,inet6_iif(skb));
            }
            *addr_len = sizeof(*sin6);
        }

        if (np->rxopt.all)
            ip6_datagram_recv_common_ctl(sk, msg, skb);

        if (is_udp4) {
            if (inet->cmsg_flags)
                ip_cmsg_recv(msg, skb);
        } else {
            if (np->rxopt.all)
                ip6_datagram_recv_specific_ctl(sk, msg, skb);
        }

        err = copied;        /*复制数据长度*/
        if (flags & MSG_TRUNC)
            err = ulen;

    out_free:
        skb_free_datagram_locked(sk, skb);      /*释放数据报*/
    out:
        return err;      /*返回复制数据长度*/
        ...
    }
```

udpv6_recvmsg()函数从套接字接收缓存队列中取出一个数据报，复制其数据至用户空间，并获取发送方地址返回给调用者，函数源代码请读者自行阅读。

■接收数据报

前面介绍的是用户从套接字接收缓存队列中读取数据报中的数据，这里要介绍的是 UDPv6 从网络层接收数据报，添加到套接字的接收缓存队列。

在 IPv6 中传输层协议需要定义并注册 inet6_protocol 实例，用于接收网络层数据包。UDPv6 定义并注

册的实例为 **udpv6_protocol**，其接收网络层数据包的函数为 **udpv6_rcv()**。

udpv6_rcv()函数定义如下（/net/ipv6/udp.c）：

```
static __inline__ int udpv6_rcv(struct sk_buff *skb)
{
    return   __udp6_lib_rcv(skb, &udp_table, IPPROTO_UDP);      /*/net/ipv6/udp.c*/
}
```

__udp6_lib_rcv()函数定义如下：

```
int __udp6_lib_rcv(struct sk_buff *skb, struct udp_table *udptable,int proto)
{
    struct net *net = dev_net(skb->dev);
    struct sock *sk;
    struct udphdr *uh;
    const struct in6_addr *saddr, *daddr;      /*IPv6 地址*/
    u32 ulen = 0;

    if (!pskb_may_pull(skb, sizeof(struct udphdr)))
        goto discard;

    saddr = &ipv6_hdr(skb)->saddr;        /*源 IPv6 地址*/
    daddr = &ipv6_hdr(skb)->daddr;      /*目的 IPv6 地址*/
    uh = udp_hdr(skb);          /*UDPv6 报头*/

    ulen = ntohs(uh->len);      /*数据报长度*/
    if (ulen > skb->len)
        goto short_packet;

    if (proto == IPPROTO_UDP) {
        if (ulen == 0)
            ulen = skb->len;

        if (ulen < sizeof(*uh))
            goto short_packet;

        if (ulen < skb->len) {
            if (pskb_trim_rcsum(skb, ulen))
                goto short_packet;
            saddr = &ipv6_hdr(skb)->saddr;
            daddr = &ipv6_hdr(skb)->daddr;
            uh = udp_hdr(skb);      /*指向 UDP 报头*/
        }
    }

    if (udp6_csum_init(skb, uh, proto))
        goto csum_error;
```

```
    if (ipv6_addr_is_multicast(daddr))      /*接收组播数据报*/
        return __udp6_lib_mcast_deliver(net, skb,saddr, daddr, udptable, proto);

    /*接收单播数据报*/
    sk = __udp6_lib_lookup_skb(skb, uh->source, uh->dest, udptable);      /*查找目的套接字*/
    if (sk) {       /*查找到了目的套接字*/
        int ret;

        if (!uh->check && !udp_sk(sk)->no_check6_rx) {
            sock_put(sk);
            udp6_csum_zero_error(skb);
            goto csum_error;
        }

        if (inet_get_convert_csum(sk) && uh->check && !IS_UDPLITE(sk))
            skb_checksum_try_convert(skb, IPPROTO_UDP, uh->check,ip6_compute_pseudo);

        ret = udpv6_queue_rcv_skb(sk, skb);    /*将数据报添加到接收缓存队列，/net/ipv6/udp.c*/
        sock_put(sk);

        if (ret > 0)
            return -ret;
        return 0;
    }

    /*下面是处理目的套接字不存在的情形*/
    if (!uh->check) {
        udp6_csum_zero_error(skb);
        goto csum_error;
    }

    if (!xfrm6_policy_check(NULL, XFRM_POLICY_IN, skb))
        goto discard;

    if (udp_lib_checksum_complete(skb))
        goto csum_error;

    UDP6_INC_STATS_BH(net, UDP_MIB_NOPORTS, proto == IPPROTO_UDPLITE);
    icmpv6_send(skb, ICMPV6_DEST_UNREACH, ICMPV6_PORT_UNREACH, 0);
                                                /*发回目的不可达 ICMPv6 消息*/
    kfree_skb(skb);
    return 0;
    ...
}
```

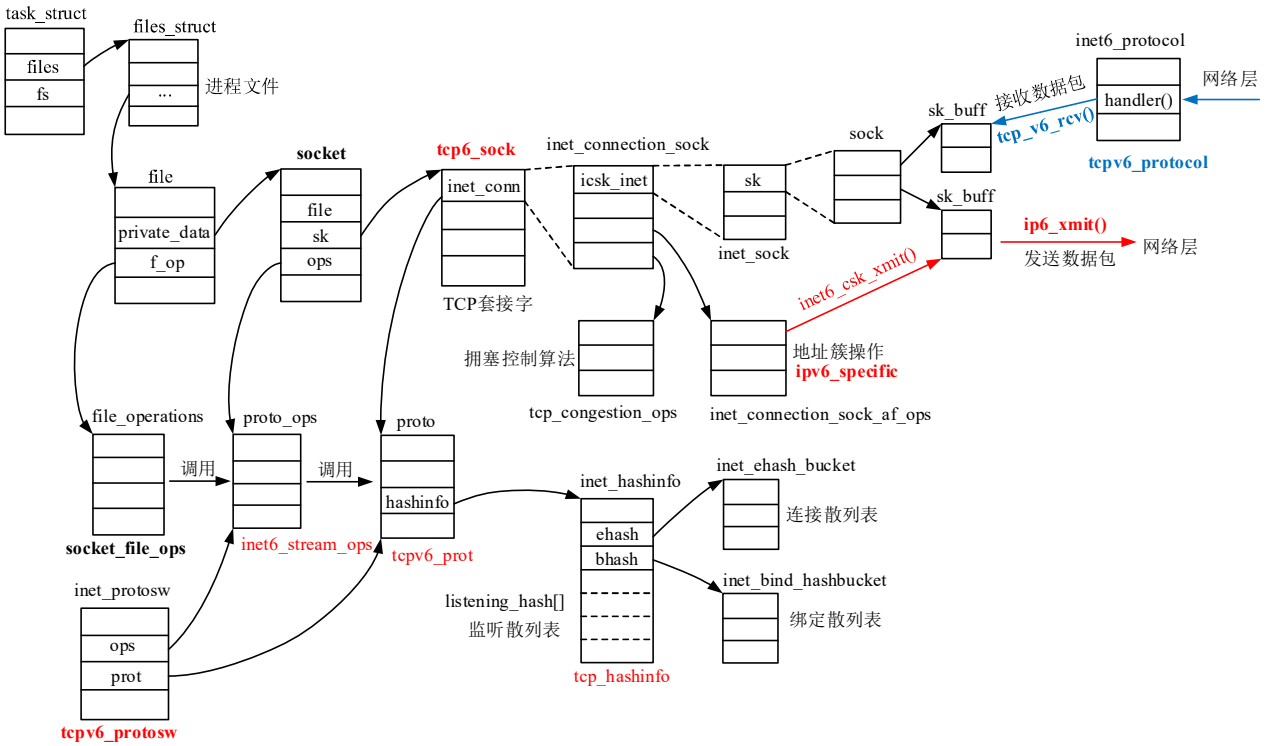
udpv6_rcv()函数主要工作是根据目的 IP 地址、端口号查找目的套接字，然后将数据报添加到套接字接收缓存队列。如果目的套接字不存在，则向发送者发回目的不可达 ICMP 消息。

3 TCP 实现

TCP 在前面也介绍过了，这里简要介绍一下 TCP 在 IPv6 中的实现（TCPv6），TCPv6 实现代码主要在/net/ipv6/tcp_ipv6.c 文件内。

■概述

IPv6 中 TCP 实现框架如下图所示，IPv6 中 TCP 套接字与 IPv4 中 TCP 套接字框架几乎相同，主要区别是少数几个数据结构定义或具体操作函数不同。



TCPv6 套接字由 tcp6_sock 结构体表示，定义如下（/include/linux/ipv6.h）：

```
struct tcp6_sock {
    struct tcp_sock    tcp;          /*同 TCPv4 中结构*/
    struct ipv6_pinfo    inet6;      /*IPv6 信息*/
};
```

TCP 套接字关联的 proto_ops 实例为 **inet6_stream_ops**，关联的 proto 实例为 **tcpv6_prot**。tcp6_sock 实例同样由 tcp_hashinfo 实例管理（同 TCPv4 套接字）。

套接字关联的地址簇操作结构实例为 **ipv6_specific**，其中的 **inet6_csk_xmit()**函数用于发送缓存队列中数据包至网络层。TCPv6 定义并注册的 inet6_protocol 实例为 **tcpv6_protocol**，其中的 **tcp_v6_rcv()**函数用于接收网络层传递的数据包，添加到套接字接收缓存队列。

TCPv6 初始化函数为 tcpv6_init()，定义如下（/net/ipv6/tcp_ipv6.c）：

```
int __init tcpv6_init(void)
{
    int ret;
    ret = inet6_add_protocol(&tcpv6_protocol, IPPROTO_TCP);      /*注册 tcpv6_protocol 实例*/
    ...
```

```
    ret = inet6_register_protosw(&tcpv6_protosw);   /*注册 tcpv6_protosw 实例，用于设置套接字*/
    ...
    ret = register_pernet_subsys(&tcpv6_net_ops);
                /*tcpv6_net_ops 初始化函数中为网络命名空间创建用于控制的原始 TCP 套接字*/
    ...
out:
    return ret;
    ...
}
```

■套接字操作

TCPv6 中 inet6_stream_ops 和 tcpv6_prot 实例中的操作函数与 TCPv4 中的操作函数大部分相同，这两个实例定义如下：

```
const struct proto_ops inet6_stream_ops = {        /*/net/ipv6/af_inet6.c*/
    .family          = PF_INET6,
    .owner         = THIS_MODULE,
    .release       = inet6_release,
    .bind            = inet6_bind,
    .connect       = inet_stream_connect,
    .socketpair     = sock_no_socketpair,
    .accept          = inet_accept,
    .getname       = inet6_getname,
    .poll            = tcp_poll,
    .ioctl               = inet6_ioctl,
    .listen              = inet_listen,
    .shutdown            = inet_shutdown,
    .setsockopt          = sock_common_setsockopt,
    .getsockopt          = sock_common_getsockopt,
    .sendmsg       = inet_sendmsg,
    .recvmsg       = inet_recvmsg,
    .mmap          = sock_no_mmap,
    .sendpage      = inet_sendpage,
    .splice_read         = tcp_splice_read,
#ifdef CONFIG_COMPAT
    .compat_setsockopt = compat_sock_common_setsockopt,
    .compat_getsockopt = compat_sock_common_getsockopt,
#endif
};

struct proto tcpv6_prot = {        /*/net/ipv6/tcp_ipv6.c*/
    .name                = "TCPv6",
    .owner               = THIS_MODULE,
    .close               = tcp_close,
    .connect         = tcp_v6_connect,        /*连接操作*/
    .disconnect          = tcp_disconnect,
```

```
    .accept             = inet_csk_accept,
    .ioctl              = tcp_ioctl,
    .init               = tcp_v6_init_sock,      /*初始化套接字*/
    .destroy            = tcp_v6_destroy_sock,
    .shutdown           = tcp_shutdown,
    .setsockopt         = tcp_setsockopt,
    .getsockopt         = tcp_getsockopt,
    .recvmsg            = tcp_recvmsg,       /*接收数据*/
    .sendmsg            = tcp_sendmsg,       /*发送数据*/
    .sendpage           = tcp_sendpage,
    .backlog_rcv        = tcp_v6_do_rcv,
    .release_cb         = tcp_release_cb,
    .hash               = inet_hash,
    .unhash             = inet_unhash,
    .get_port           = inet_csk_get_port,     /*分配或设置端口号*/
    .enter_memory_pressure= tcp_enter_memory_pressure,
    .stream_memory_free   = tcp_stream_memory_free,
    .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),      /*套接字由 tcp6_sock 结构体表示*/
    .slab_flags         = SLAB_DESTROY_BY_RCU,
    .twsk_prot          = &tcp6_timewait_sock_ops,
    .rsk_prot           = &tcp6_request_sock_ops,
    .h.hashinfo         = &tcp_hashinfo,         /*同 TCPv4 中实例*/
    .no_autobind        = true,
#ifdef CONFIG_COMPAT
    .compat_setsockopt= compat_tcp_setsockopt,
    .compat_getsockopt      = compat_tcp_getsockopt,
#endif
#ifdef CONFIG_MEMCG_KMEM
    .proto_cgroup       = tcp_proto_cgroup,
#endif
    .clear_sk           = tcp_v6_clear_sk,
};
```

以上两个实例中，除了绑定、连接、初始化等几个函数外，其余与 TCPv4 中函数都相同，包括发送、读取数据函数，套接字管理结构实例与 TCPv4 相同，因此这里不再介绍了。

需要注意的是，在发送数据包时，将调用 inet_connection_sock_af_ops 实例中的 queue_xmit()函数发送数据包，这里一般为 **inet6_csk_xmit()**函数。inet6_csk_xmit()函数将调用 **ip6_xmit()**函数发送数据包，这是数据包进入网络层的入口函数，在后面介绍 IPv6 网络层协议实现时再介绍。

■接收数据包

TCPv6 定义并注册了 net6_protocol 实例 **tcpv6_protocol**，用于接收网络层传递的数据包，如下所示：

```
static const struct inet6_protocol tcpv6_protocol = {          /*/net/ipv6/tcp_ipv6.c*/
    .early_demux  =    tcp_v6_early_demux,
    .handler  =    tcp_v6_rcv,          /*接收数据包函数*/
    .err_handler    =    tcp_v6_err,
    .flags          =    INET6_PROTO_NOPOLICY|INET6_PROTO_FINAL,
};
```

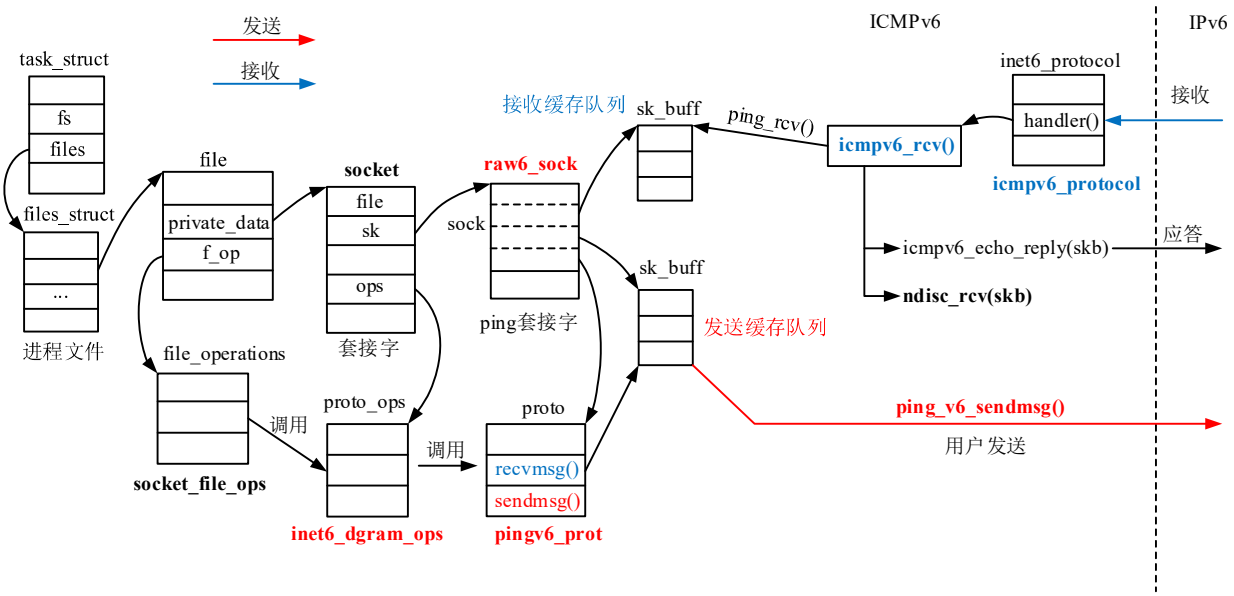
tcp_v6_rcv()函数用于接收 IPv6 数据包，函数定义在 net/ipv6/tcp_ipv6.c 文件内，函数代码与 tcp_v4_rcv()函数类似，请读者自行阅读源代码。

4 ICMP 实现

ICMP 在 IPv6 中被赋予更多的功用。在 IPv6 报头“下一个头部”字段中用 58（IPPROTO_ICMPV6，不同于 IPv4 中值）标识下一个头部是 ICMPv6 报头。用户进程也可通过 ping 套接字与 ICMPv6 通信。

■ICMPv6 实现

ICMPv6 与 ICMPv4 实现类似，其实现框架如下图所示：



ICMPv6 实现代码主要位于/net/ipv6/icmp.c 文件内。ICMPv6 中定义并注册了 inet6_protocol 结构体实例 **icmpv6_protocol**，其接收函数为 **icmpv6_rcv()**。icmpv6_rcv()函数根据 ICMPv6 报头中的**类型**值，分别调用相应的处理函数，例如，回显应答数据报由 ping_rcv(skb)函数处理（数据包交给 ping 套接字），邻居请求和应答、重定向等数据报由 ndisc_rcv(skb)函数处理。

使用 ICMPv6 的协议，如邻居发现等，需要定义接收相应数据报的处理函数。发送数据时，构建 ICMPv6 数据报，并发送到网络层。

■ping 套接字

用户进程可通过 ping 套接字与 ICMPv6 通信。ping 套接字实现代码位于/net/ipv6/ping.c 文件内。用于设置 ping 套接字的 inet_protosw 实例定义如下：

```
static struct inet_protosw pingv6_protosw = {
        .type =          SOCK_DGRAM,
        .protocol =    IPPROTO_ICMPV6,
        .prot =          &pingv6_prot,          /*proto 实例*/
        .ops =           &inet6_dgram_ops,      /*同 UDPv6 套接字*/
        .flags =         INET_PROTOSW_REUSE,
};
```

ping 套接字关联的 proto_ops 实例同 UDPv6 套接字关联的实例，proto 实例为 **pingv6_prot**，定义如下：

```
struct proto pingv6_prot = {
        .name =          "PINGv6",
        .owner =  THIS_MODULE,
        .init =          ping_init_sock,
        .close =    ping_close,
        .connect =       ip6_datagram_connect_v6_only,
        .disconnect =   udp_disconnect,
        .setsockopt =   ipv6_setsockopt,
        .getsockopt =   ipv6_getsockopt,
        .sendmsg =       ping_v6_sendmsg,       /*发送 ICMPv6 数据报*/
        .recvmsg =       ping_recvmsg,          /*接收 ICMPv6 数据报，同 ICMPv4*/
        .bind =          ping_bind,
        .backlog_rcv =       ping_queue_rcv_skb,
        .hash =          ping_hash,
        .unhash =ping_unhash,
        .get_port =      ping_get_port,
        .obj_size =      sizeof(struct raw6_sock),      /*套接字由 raw6_sock 结构体表示*/
};
```

ping 套接字由 raw6_sock 结构体实例表示，接收数据报的函数为 ping_recvmsg()，同 ICMPv4。发送数据报时，用户进程需要构建 ICMPv6 数据报报头（由 icmp6hdr 结构体表示，/include/uapi/linux/icmpv6.h），用户数据最终由 **ping_v6_sendmsg()**函数发送到网络层。

13.3.3 IPv6 协议实现

本小节简要介绍 IPv6 网络层协议在 Linux 内核中的实现。IPv6 实现在内核中是一个可选择的配置选项（IPV6 选项），IPv6 实现可配置成模块。

1 概述

下面先介绍 IPv6 报头的表示，以及 IPv6 中发送、接收数据包的流程。后面将介绍 IPv6 网络层协议实现中各子系统的实现，以及发送、接收数据包函数的实现。

■IPv6 报头

IPv6 报头结构在前面介绍过了，在内核中 IPv6 报头由 ipv6hdr 结构体表示，定义如下：

```
struct ipv6hdr {                /*/include/uapi/linux/ipv6.h*/
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u8        priority:4,          /*优先级*/
```

```
                version:4;          /*版本号*/
#elif defined(__BIG_ENDIAN_BITFIELD)
    __u8        version:4,
                priority:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
    __u8        flow_lbl[3];    /*流标签*/

    __be16      payload_len;        /*负载长度，IP 数据报长度，不包括 IPv6 报头*/
    __u8        nexthdr;            /*下一个头部*/
    __u8        hop_limit;          /*跳数限制*/

    struct in6_addr     saddr;      /*源 IP 地址，/include/uapi/linux/in6.h*/
    struct in6_addr     daddr;      /*目的 IP 地址*/
};
```

ipv6hdr 结构体部分成员简介如下：

●**nexthdr**：标识下一个报头。IPv6 报头之后可以有若干个扩展报头，最后是传输层协议报头。下一个头部字段标识下一个扩展报头的类型或传输层报头。其中传输层报头标识同 IPv4 中传输层报头标识（除 ICMP 外）。扩展报头标识定义如下（/include/uapi/linux/in6.h）：

```
#if __UAPI_DEF_IPPROTO_V6
  #define   IPPROTO_HOPOPTS         0   /* IPv6 hop-by-hop options，逐跳选项扩展报头*/
  #define   IPPROTO_ROUTING         43  /* IPv6 routing header，路由扩展报头*/
  #define   IPPROTO_FRAGMENT        44  /* IPv6 fragmentation header，分片扩展报头*/
  #define   IPPROTO_ICMPV6          58  /* ICMPv6，不同于 ICMPv4*/
  #define   IPPROTO_NONE            59  /* IPv6 no next header，没有下一个扩展报头*/
  #define   IPPROTO_DSTOPTS         60  /* IPv6 destination options，目的地选项扩展报头*/
  #define   IPPROTO_MH              135 /* IPv6 mobility header，移动扩展报头*/
#endif      /* __UAPI_DEF_IPPROTO_V6 */
```

●**saddr、daddr**：源、目的 IP 地址，由 in6_addr 结构体表示，结构体定义见上一小节。

■IPv6 资源

在网络命名空间 net 结构体中，包含 netns_ipv6 结构体成员，用于记录 IPv6 资源和参数等，如下所示：

```
struct net {
    ...
  #if IS_ENABLED(CONFIG_IPV6)
    struct netns_ipv6   ipv6;       /*记录 IPv6 网络协议资源和参数*/
  #endif
    ...
}
```

netns_ipv6 结构体定义如下（/include/net/netns/ipv6.h）：

```
struct netns_ipv6 {
```

```
    struct netns_sysctl_ipv6   sysctl;      /*系统控制参数，/include/net/netns/ipv6.h*/
    struct ipv6_devconf  *devconf_all;      /*网络设备配置参数（网络层参数）*/
    struct ipv6_devconf  *devconf_dflt;     /*默认的网络设备配置参数*/
    struct inet_peer_base     *peers;
    struct netns_frags   frags;
#ifdef CONFIG_NETFILTER          /*xt_table 表*/
    struct xt_table        *ip6table_filter;
    struct xt_table        *ip6table_mangle;
    struct xt_table        *ip6table_raw;
  #ifdef CONFIG_SECURITY
    struct xt_table        *ip6table_security;
  #endif
    struct xt_table        *ip6table_nat;
#endif
    struct rt6_info          *ip6_null_entry;      /*空的路由选择表项，内核设置的默认表项*/
                                                   /*指向内核定义的 ip6_null_entry_template 实例*/
    struct rt6_statistics     *rt6_stats;
    struct timer_list          ip6_fib_timer;      /*定时器*/
    struct hlist_head          *fib_table_hash;     /*管理路由选择表的散列表*/
    struct fib6_table          *fib6_main_tbl;      /*主路由选择表*/
    struct dst_ops         ip6_dst_ops;
                           /*dst_entry 实例操作结构实例，复制于内核定义的 ip6_dst_ops_template 实例*/
    unsigned int             ip6_rt_gc_expire;
    unsigned long            ip6_rt_last_gc;

#ifdef CONFIG_IPV6_MULTIPLE_TABLES      /*支持策略路由（多个路由选择表）*/
    struct rt6_info             *ip6_prohibit_entry;
    struct rt6_info             *ip6_blk_hole_entry;
    struct fib6_table           *fib6_local_tbl;
    struct fib_rules_ops        *fib6_rules_ops;
#endif
    struct sock             **icmp_sk;       /*用于系统控制的各传输层协议套接字*/
    struct sock            *ndisc_sk;
    struct sock            *tcp_sk;
    struct sock            *igmp_sk;
    struct sock             *mc_autojoin_sk;
#ifdef CONFIG_IPV6_MROUTE      /*机器配置为组播路由器*/
  #ifndef CONFIG_IPV6_MROUTE_MULTIPLE_TABLES
    struct mr6_table      *mrt6;
  #else
    struct list_head       mr6_tables;
    struct fib_rules_ops    *mr6_rules_ops;
  #endif
#endif
    atomic_t         dev_addr_genid;
    atomic_t         fib6_sernum;        /*路由选择条目当前编号*/
```

};

在初始化函数 inet6_init()中将注册 pernet_operations 实例，实例的初始化函数将设置网络命名空间中 IPv6 系统控制参数，如下所示：

```
static int __init inet6_init(void)
{
    ...
    err = register_pernet_subsys(&inet6_net_ops);
                                        /*注册 pernet_operations 实例，/net/ipv6/af_inet6.c*/
    ...
}
```

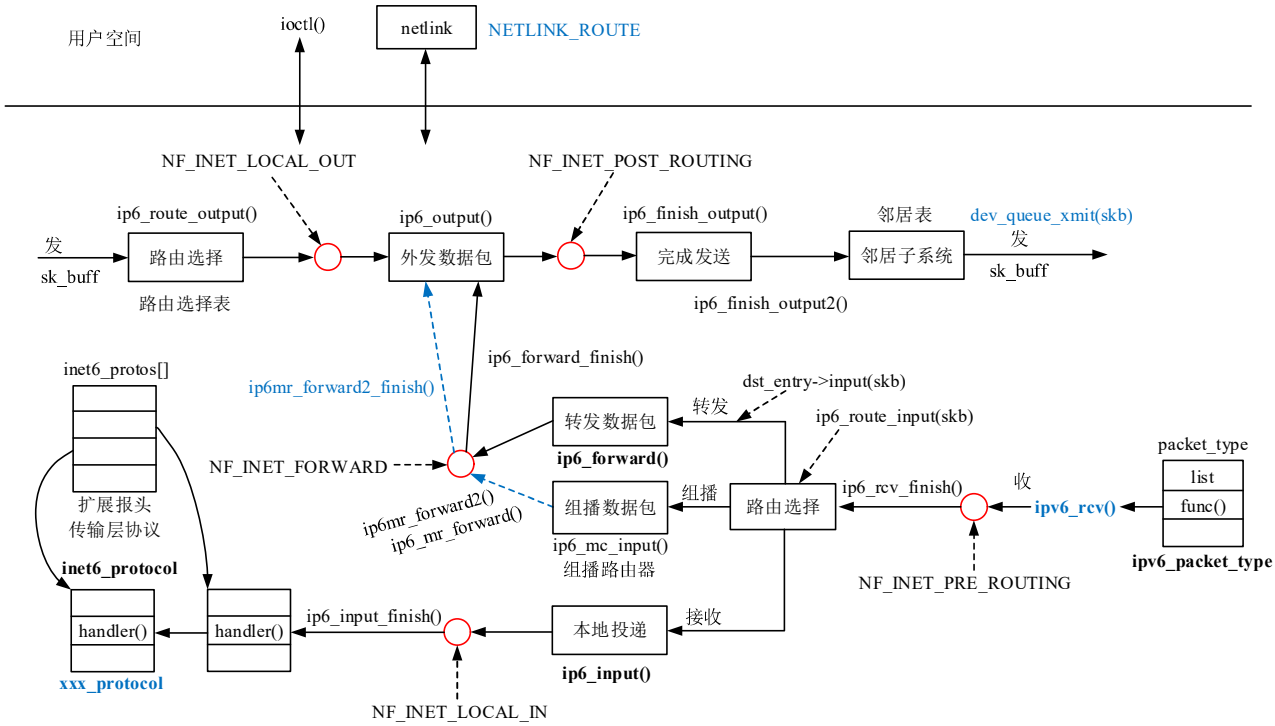
inet6_net_ops 实例定义如下：

```
static struct pernet_operations inet6_net_ops = {
    .init = inet6_net_init,      /*初始化函数*/
    .exit = inet6_net_exit,
};
```

初始化函数 inet6_net_init()中将初始化 net->ipv6.sysctl 成员内各参数。net->ipv6 成员中其它的资源、参数将在各子系统初始化时设置。

■发送接收数据包流程

下面先简要概述 IPv6 中发送、接收数据包流程，后面将介绍 IPv6 中各子系统，如路由选择子系统、邻居子系统等，以及发送/接收数据包流程的实现。



IPv6 发送接收数据包流程与 IPv4 发送接收流程相似。

发送通道：在套接字发送数据包时（或执行连接操作时），将执行路由选择操作，它根据目的 IPv6 地址确定数据包的下一步走向。发送通道路由选择的接口函数为 ip6_route_output()，此函数在路由选择表

中查找最长前缀匹配的表项，以确定外发的网络设备和下一跳 IPv6 地址，并将数据包的下一步处理函数设为 ip6_output()。ip6_output()函数最终在邻居表中查找表示下一跳的邻居实例（不存在则创建），如果邻居中缓存了 L2 层报头，则直接将 L2 层报头写入数据包，将数据包传递给数据链路层。如果未缓存 L2 层报头（或邻居物理地址），则通过邻居发现协议（ND 协议，由 ICMPv6 承载）解析邻居物理地址后，再发送数据包。

接收通道：IPv6 定义并注册了 packet_type 结构体实例 **ipv6_packet_type**，其中的 func()函数成员为 **ipv6_rcv()**，这是数据包进入 IPv6 网络层的入口函数。接收通道也要进行路由选择，ip6_route_output()为接口函数。路由选择结果中，对于投递到本机的数据包处理函数为 **ip6_input()**，转发数据包为 **ip6_forward()**，转发组播数据包函数为 ip6_mc_input()（本机配置为组播路由器）。对于转发数据包，最终都由 ip6_output()函数发出。对于投递到本机的数据包，ip6_input()函数将数据包交给 ip6_input_finish()函数处理。

在最终接收数据包的 ip6_input_finish()函数中将逐级扫描数据包中扩展报头、传输层报头，分别调用各报头对应 inet6_protocol 实例中的处理函数，处理数据包（各扩展报头、传输层协议需要注册 inet6_protocol 实例）。最后一级报头是传输层报头，即数据包最终将传递给传输层协议处理。

Netfilter 子系统同样也适用于 IPv6，在发送、接收通道中嵌入了与 IPv4 相同的钩子，内核或用户可向钩子注册回调函数。

3 路由选择子系统

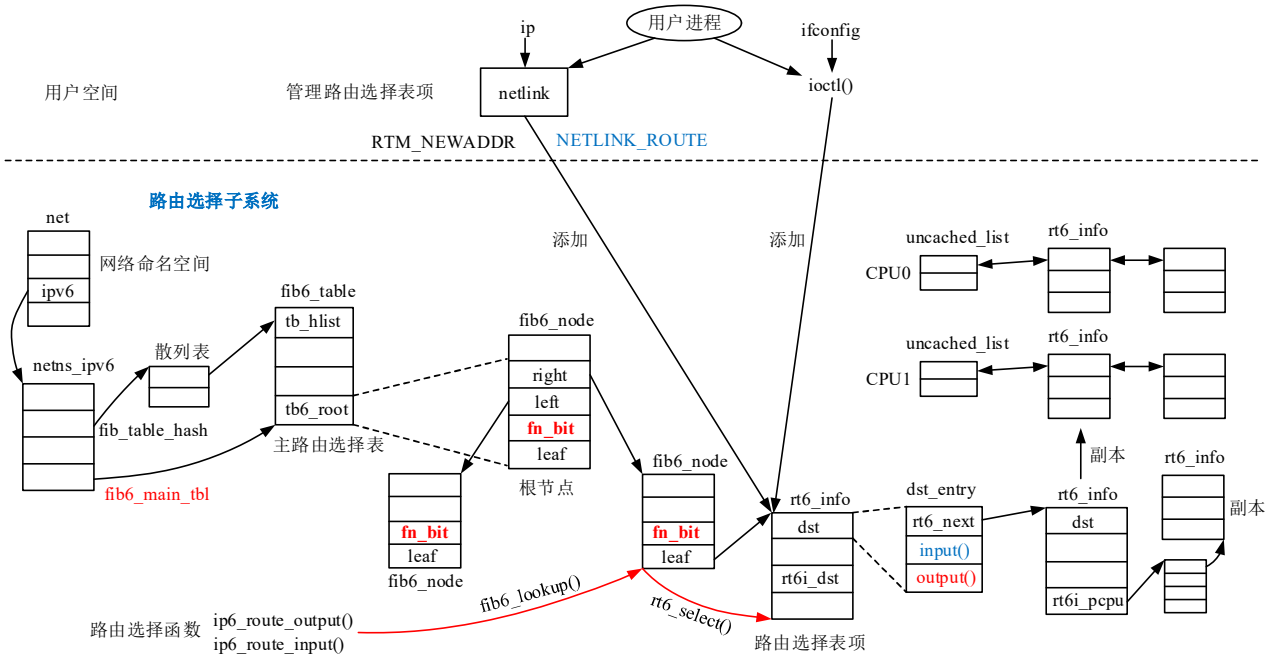
IPv6 中路由选择子系统与 IPv4 中的功能相同，结构也类似，主要不同之处是表示路由选择表、表项等的数据结构不同。

■概述

下面简要概述 IPv6 路由选择子系统框架，以及路由选择子系统的初始化，后面再分别介绍路由选择表的结构、用户进程接口，以及 IPv6 中路由选择函数的实现。

●框架

IPv6 中路由选择子系统实现框架如下图所示（与 IPv4 中路由选择子系统类似，但又有不同）：



IPv6 中路由选择表由网络命名空间管理，在初始化网络命名空间时为其创建主路由选择表（没有配置

策略路由）。路由选择表由 fib6_table 结构体表示，网络命名空间中定义了散列表，用于管理路由选择表。路由选择表利用一种类似于 IPv4 中的 TRIE 树来管理路由选择表项，树的节点由 fib6_node 结构体表示，路由选择表项由 **rt6_info** 结构体表示（不同于 IPv4 中的 rtable 结构体）。

每个 fib6_node 节点都关联到一个 rt6_info 实例，不同节点可以关联相同的 rt6_info 实例。目的 IPv6 地址、输出网络设备等表项信息保存在 rt6_info 实例中。fib6_node 节点中主要保存了匹配前缀长度参数，节点按匹配前缀长度从小到大在树中从上至下排列，上层节点是父节点，下层节点是子节点。每个节点表示一个网络地址范围（由目的 IPv6 地址和匹配前缀长度确定），子节点表示的范围是父节点范围的子集，即是父节点表示网络的子网（子节点匹配前缀更长）。匹配前缀长度及比特位相等（表示相同网络地址范围）的路由选择表项（参数不同），由同一个 fib6_node 节点管理，各表项 rt6_info 实例组成单链表。

用户进程可以通过 netlink 套接字、ioctl()系统调用等向内核添加路由选择表项、本机 IPv6 地址等，这些操作会在路由选择表中添加路由选择表项。

在 IPv6 发送和接收数据包的路径中都将执行路由选择操作，**ip6_route_output()/ip6_route_input()**为接口函数。这两个函数中，将调用 fib6_lookup()函数在路由选择表中查找匹配节点的 fib6_node 实例，调用函数 rt6_select()在 fib6_node 节点中选择合适的表项 rt6_info 实例，返回给调用者。另外，路由选择函数中还会复制 rt6_info 实例的一个副本，添加到 CPU 核对应的 uncached_list 链表或赋予 rt6_info 实例中的 percpu 指针变量。

rt6_info 结构体中嵌入了 dst_entry 结构体成员，表示路由选择结果。rt6_info 实例由 net->ipv6.ip6_dst_ops 表示的 dst_ops 结构体实例管理，此实例复制于内核定义的 ip6_dst_ops_template 实例。

在创建网络命名空间的初始化时会为 net->ipv6.ip6_dst_ops 实例创建 rt6_info 结构体的 slab 缓存，在添加路由时将由此 slab 缓存分配 rt6_info 实例，并初始化且添加到路由选择表。分配 rt6_info 实例时，同时会设置其内嵌的 dst_entry 实例，主要是其中的 input()和 output()函数指针，表示数据包下一步的处理函数。

●初始化

内核在/net/ipv6/route.c 文件内，定义了 dst_ops 实例 **ip6_dst_ops_template**，用于设置网络命名空间中的 net->ipv6.ip6_dst_ops 成员，如下所示：

```
static struct dst_ops ip6_dst_ops_template = {
    .family             =   AF_INET6,
    .gc             =   ip6_dst_gc,
    .gc_thresh          =   1024,
    .check              =   ip6_dst_check,
    .default_advmss     =   ip6_default_advmss,
    .mtu            =   ip6_mtu,
    .cow_metrics        =   ipv6_cow_metrics,
    .destroy        =   ip6_dst_destroy,
    .ifdown             =   ip6_dst_ifdown,
    .negative_advice    =   ip6_negative_advice,
    .link_failure       =   ip6_link_failure,
    .update_pmtu        =   ip6_rt_update_pmtu,
    .redirect       =   rt6_do_redirect,
    .local_out      =   __ip6_local_out,
    .neigh_lookup       =   ip6_neigh_lookup,
};
```

另外，还定义了 dst_ops 实例 ip6_dst_blackhole_ops。

在/net/ipv6/route.c 文件内，定义了默认的空路由选择表项，作为系统初始的默认路由选择表项，如下

所示：

```
static const struct rt6_info ip6_null_entry_template = {            /*目的地址为 0*/
    .dst = {
        .__refcnt  = ATOMIC_INIT(1),
        .__use          = 1,
        .obsolete  = DST_OBSOLETE_FORCE_CHK,
        .error          = -ENETUNREACH,
        .input          = ip6_pkt_discard,    /*丢弃数据包，发送目的不可达 ICMPv6 消息*/
        .output         = ip6_pkt_discard_out,    /*丢弃数据包，发送目的不可达 ICMPv6 消息*/
    },
    .rt6i_flags      = (RTF_REJECT | RTF_NONEXTHOP),
    .rt6i_protocol   = RTPROT_KERNEL,
    .rt6i_metric     = ~(u32) 0,
    .rt6i_ref   = ATOMIC_INIT(1),
};
```

IPv6 中路由选择子系统初始化函数为 ip6_route_init()（在 inet6_init()函数中被调用），函数定义如下：

```
int __init ip6_route_init(void)        /*/net/ipv6/route.c*/
{
    int ret;
    int cpu;

    ret = -ENOMEM;
    ip6_dst_ops_template.kmem_cachep =kmem_cache_create("ip6_dst_cache", sizeof(struct rt6_info), 0,
                                        SLAB_HWCACHE_ALIGN, NULL);
                                                    /*创建 rt6_info 结构体 slab 缓存*/
    ...
    ret = dst_entries_init(&ip6_dst_blackhole_ops);       /*初始化 ip6_dst_blackhole_ops 实例*/
    ...
    ret = register_pernet_subsys(&ipv6_inetpeer_ops);     /*初始化函数中初始化 net->ipv6.peers 成员*/
    ...
    ret = register_pernet_subsys(&ip6_route_net_ops); /*初始化函数为 ip6_route_net_init()，见下文*/
    ...
    ip6_dst_blackhole_ops.kmem_cachep = ip6_dst_ops_template.kmem_cachep;
    /*设置空路由选择表项*/
    init_net.ipv6.ip6_null_entry->dst.dev = init_net.loopback_dev;          /*指向环回设备*/
    init_net.ipv6.ip6_null_entry->rt6i_idev = in6_dev_get(init_net.loopback_dev);

#ifdef CONFIG_IPV6_MULTIPLE_TABLES        /*支持策略路由*/
    init_net.ipv6.ip6_prohibit_entry->dst.dev = init_net.loopback_dev;
    init_net.ipv6.ip6_prohibit_entry->rt6i_idev = in6_dev_get(init_net.loopback_dev);
    init_net.ipv6.ip6_blk_hole_entry->dst.dev = init_net.loopback_dev;
    init_net.ipv6.ip6_blk_hole_entry->rt6i_idev = in6_dev_get(init_net.loopback_dev);
#endif

    ret = fib6_init();      /*路由选择表初始化，创建路由选择表等，/net/ipv6/ip6_fib.c*/
```

```
    ...
    ret = xfrm6_init();
    ...
    ret = fib6_rules_init();        /*策略路由初始化*/
    ...
    ret = register_pernet_subsys(&ip6_route_net_late_ops); /*初始化函数在 proc 文件系统中创建文件*/
    ...
    ret = -ENOBUFS;

    /*注册添加路由、删除路由、获取路由消息的处理函数*/
    if (__rtnl_register(PF_INET6, RTM_NEWROUTE, inet6_rtm_newroute, NULL, NULL) ||
    __rtnl_register(PF_INET6, RTM_DELROUTE, inet6_rtm_delroute, NULL, NULL) ||
    __rtnl_register(PF_INET6, RTM_GETROUTE, inet6_rtm_getroute, NULL, NULL))
        goto out_register_late_subsys;

    ret = register_netdevice_notifier(&ip6_route_dev_notifier);
                /*向 netdev_chain 通知链注册通知，/net/ipv6/route.c*/
                /*通知回调函数为 ip6_route_dev_notify()，函数内判断如果注册的是环回设备，
                *则将此网络设备赋予 net->ipv6.ip6_null_entry 路由条目。
                */
    ...
    for_each_possible_cpu(cpu) {        /*初始化各 CPU 的 uncached_list 链表*/
        struct uncached_list *ul = per_cpu_ptr(&rt6_uncached_list, cpu);

        INIT_LIST_HEAD(&ul->head);
        spin_lock_init(&ul->lock);
    }
out:
    return ret;
    ...
}
```

ip6_route_init()函数中为 ip6_dst_ops_template 实例创建路由选择条目 rt6_info 结构体 slab 缓存，初始化 ip6_dst_blackhole_ops 实例；注册了 3 个 pernet_operations 实例，其中 **ip6_route_net_ops** 实例的初始化函数为 ip6_route_net_init()，见下文；设置了初始的默认路由选择条目；调用 fib6_init()函数初始化路由选择表（创建 fib6_node 结构体缓存，创建路由选择表等）；调用 fib6_rules_init()函数初始化策略路由（多个路由选择表）；注册了添加路由、删除路由、获取路由消息的处理函数；最后向 netdev_chain 通知链注册通知 ip6_route_dev_notifier，其回调函数将注册的环回设备赋予初始的默认路由选择表项。

下面看一下 ip6_route_net_ops 实例的初始化函数 ip6_route_net_init()的定义，代码简列如下：

```
static int __net_init ip6_route_net_init(struct net *net)
{
    int ret = -ENOMEM;

    memcpy(&net->ipv6.ip6_dst_ops, &ip6_dst_ops_template,sizeof(net->ipv6.ip6_dst_ops));
                        /*复制 ip6_dst_ops_template 实例至 net->ipv6.ip6_dst_ops*/

    if (dst_entries_init(&net->ipv6.ip6_dst_ops) < 0)        /*初始化 net->ipv6.ip6_dst_ops 成员*/
```

```
        goto out_ip6_dst_ops;

    net->ipv6.ip6_null_entry = kmemdup(&ip6_null_entry_template,sizeof(*net->ipv6.ip6_null_entry),
                                       GFP_KERNEL);
                            /*网络命名空间初始默认路由选择表项为 ip6_null_entry_template*/
    if (!net->ipv6.ip6_null_entry)
        goto out_ip6_dst_entries;
    net->ipv6.ip6_null_entry->dst.path =(struct dst_entry *)net->ipv6.ip6_null_entry;
    net->ipv6.ip6_null_entry->dst.ops = &net->ipv6.ip6_dst_ops;
    dst_init_metrics(&net->ipv6.ip6_null_entry->dst,ip6_template_metrics, true);

#ifdef CONFIG_IPV6_MULTIPLE_TABLES      /*支持策略路由*/
    net->ipv6.ip6_prohibit_entry = kmemdup(&ip6_prohibit_entry_template,
                                    sizeof(*net->ipv6.ip6_prohibit_entry),GFP_KERNEL);
    if (!net->ipv6.ip6_prohibit_entry)
        goto out_ip6_null_entry;
    net->ipv6.ip6_prohibit_entry->dst.path =(struct dst_entry *)net->ipv6.ip6_prohibit_entry;
    net->ipv6.ip6_prohibit_entry->dst.ops = &net->ipv6.ip6_dst_ops;
    dst_init_metrics(&net->ipv6.ip6_prohibit_entry->dst,ip6_template_metrics, true);

    net->ipv6.ip6_blk_hole_entry = kmemdup(&ip6_blk_hole_entry_template,
                                    sizeof(*net->ipv6.ip6_blk_hole_entry),GFP_KERNEL);
    if (!net->ipv6.ip6_blk_hole_entry)
        goto out_ip6_prohibit_entry;
    net->ipv6.ip6_blk_hole_entry->dst.path =
        (struct dst_entry *)net->ipv6.ip6_blk_hole_entry;
    net->ipv6.ip6_blk_hole_entry->dst.ops = &net->ipv6.ip6_dst_ops;
    dst_init_metrics(&net->ipv6.ip6_blk_hole_entry->dst, ip6_template_metrics, true);
#endif      /*支持策略路由结束*/

    /*设置系统控制参数*/
    net->ipv6.sysctl.flush_delay = 0;
    net->ipv6.sysctl.ip6_rt_max_size = 4096;
    net->ipv6.sysctl.ip6_rt_gc_min_interval = HZ / 2;
    net->ipv6.sysctl.ip6_rt_gc_timeout = 60*HZ;
    net->ipv6.sysctl.ip6_rt_gc_interval = 30*HZ;
    net->ipv6.sysctl.ip6_rt_gc_elasticity = 9;
    net->ipv6.sysctl.ip6_rt_mtu_expires = 10*60*HZ;
    net->ipv6.sysctl.ip6_rt_min_advmss = IPV6_MIN_MTU - 20 - 40;

    net->ipv6.ip6_rt_gc_expire = 30*HZ;

    ret = 0;
out:
    return ret;
    ...
```

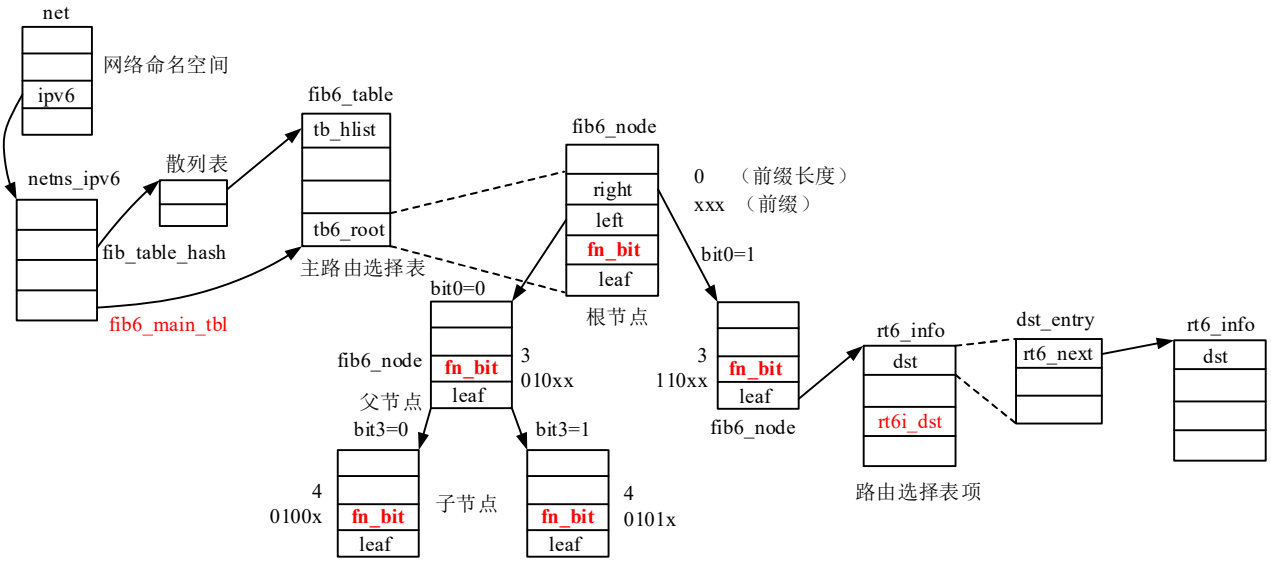
}

■路由选择表

下面介绍 IPv6 中路由选择表的结构、主要的数据结构，以及向路由选择表添加、查找表项等的接口函数。

●路由选择表结构

IPv6 路由选择表结构与 IPv4 中路由选择表结构类似，如下图所示：



路由选择表由 fib6_table 结构体表示，网络命名空间中包含散列表用于管理路由选择表，初始化时将创建主路由选择表（若没有配置支持策略路由，只有一个表）。路由选择表中节点由 fib6_node 结构体表示，fib6_node 实例组成父子关系的层次结构。fib6_node 结构体中 fn_bit 成员表示匹配前缀长度（比特位数），前缀值（目的地址）保存在 fib6_node 实例关联的 rt6_info 实例的 rt6i_dst 成员中。fib6_node 实例按前缀长度从小到大，在层次结构中从上至下排列。每个父节点的前缀与其子节点前缀的关系是包含与被包含的关系，也就是说子节点前缀定义的网络范围是父节点前缀定义网络范围的子集（子节点表示父节点的子网）。前缀值（二进制表示的值）大的子节点在右边，小的在左边。在添加新节点时，可能需要增加额外的中间节点。

rt6_info 结构体表示路由选择表项，每个 fib6_node 实例都需要关联一个 rt6_info 实例，父子节点可能关联相同的 rt6_info 实例。

若支持多路径路由选择，相同目的地址的路由表项 rt6_info 实例由同一个 fib6_node 实例管理，rt6_info 实例组成单链表。

●数据结构

IPv6 中路由选择表由 fib6_table 结构体表示，定义如下（/include/net/ip6_fib.h）：

```
struct fib6_table {
    struct hlist_node    tb6_hlist;      /*散列表节点，将表添加到网络命名空间中散列表*/
    u32                  tb6_id;         /*路由选择表标识*/
    rwlock_t             tb6_lock;
    struct fib6_node     tb6_root;       /*基数树根节点，*/
    struct inet_peer_base    tb6_peers;
};
```

IPv6 路由选择表中通过基数树管理路由选择表项，基数树每个节点由 fib6_node 结构体表示，定义如下（/include/net/ip6_fib.h）：

```
struct fib6_node {
    struct fib6_node    *parent;        /*父节点*/
    struct fib6_node    *left;          /*左边节点*/
    struct fib6_node    *right;         /*右边节点*/
#ifdef CONFIG_IPV6_SUBTREES
    struct fib6_node    *subtree;       /*以源 IP 地址建立子树*/
#endif
    struct rt6_info     *leaf;          /*指向表示路由选择表项的 rt6_info 实例*/

    __u16               fn_bit;         /*前缀长度（比特位）*/
    __u16               fn_flags;       /*标志*/
    int                 fn_sernum;      /*编号，按顺序编号*/
    struct rt6_info     *rr_ptr;
};
```

fib6_node 结构体实例在基数树中组成父子关系的层次结构，主要成员简介如下：

●**parent**：父节点。

●**left**：左边节点。

●**right**：右边节点。

●**fn_bit**：目的地址前缀长度，比特位数。

●**fn_sernum**：节点编号，顺序编号。

●**fn_flags**：标志成员，取值如下（/include/net/ip6_fib.h）：

```
#define   RTN_TL_ROOT     0x0001
#define   RTN_ROOT        0x0002      /*根节点*/
#define   RTN_RTINFO      0x0004      /*节点具有有效的路由选择表项信息*/
```

●**leaf**：指向表示路由选择表项的 rt6_info 实例，结构体定义见下文。

rt6_info 结构体表示路由选择表项，定义如下（/include/net/ip6_fib.h）：

```
struct rt6_info {
    struct dst_entry    dst;        /*路由选择结果*/

    struct fib6_table       *rt6i_table;    /*指向路由选择表*/
    struct fib6_node        *rt6i_node;     /*指向所属的基数树节点*/

    struct in6_addr     rt6i_gateway;   /*网关（下一跳）IPv6 地址*/

    /*多路径路由选择：rt6i_siblings 链接 rt6_info 实例，表示目的地址相同，参数不同的条件目*/
    struct list_head        rt6i_siblings;
    unsigned int            rt6i_nsiblings;

    atomic_t            rt6i_ref;   /*引用计数*/

    struct rt6key       rt6i_dst    ____cacheline_aligned_in_smp;   /*目的 IPv6 地址*/
    u32                 rt6i_flags;     /*标志*/
```

```
    struct rt6key           rt6i_src;       /*源 IPv6 地址*/
    struct rt6key           rt6i_prefsrc;

    struct list_head        rt6i_uncached;      /*双链表成员*/
    struct uncached_list    *rt6i_uncached_list;

    struct inet6_dev        *rt6i_idev;         /*输出网络设备*/
    struct rt6_info * __percpu  *rt6i_pcpu;     /*percpu 指针数组*/

    u32                 rt6i_metric;
    u32                 rt6i_pmtu;
    unsigned short      rt6i_nfheader_len;
    u8                  rt6i_protocol;
};
```

rt6_info 结构体中主要成员简介如下：

●**dst**：dst_entry 结构体成员，表示路由选择结果，其中包含数据包下一步的处理函数指针。

●**rt6i_gateway**：网关（下一跳）IPv6 地址。

●**rt6i_dst**：目的 IPv6 地址，由 rt6key 结构体表示，定义如下：

```
struct rt6key {
    struct in6_addr     addr;       /*IPv6 地址*/
    int         plen;           /*前缀长度*/
};
```

●**rt6i_idev**：指向 inet6_dev 结构体实例，在网络层表示网络设备，结构体定义在/include/net/if_inet6.h 头文件。

●**初始化**

IPv6 路由选择表初始化函数为 **fib6_init()**，函数调用关系为 inet6_init()->ip6_route_init()->**fib6_init()**。函数定义如下（/net/ipv6/ip6_fib.c）：

```
int __init fib6_init(void)
{
    int ret = -ENOMEM;

    fib6_node_kmem = kmem_cache_create("fib6_nodes",sizeof(struct fib6_node),
                        0, SLAB_HWCACHE_ALIGN,NULL);        /*创建 fib6_node 结构体缓存*/
    ...
    ret = register_pernet_subsys(&fib6_net_ops);        /*初始化函数中创建主路由选择表等*/
    ...
    ret = __rtnl_register(PF_INET6, RTM_GETROUTE, NULL, inet6_dump_fib,NULL);
    ...
    __fib6_flush_trees = fib6_flush_trees;
out:
    return ret;
    ...
}
```

fib6_init()函数内为 fib6_node 结构体创建了 slab 缓存，注册了 pernet_operations 实例 fib6_net_ops。fib6_net_ops 实例定义如下（/net/ipv6/ip6_fib.c）：

```
static struct pernet_operations fib6_net_ops = {
    .init = fib6_net_init,        /*初始化函数，创建主路由选择表等*/
    .exit = fib6_net_exit,
};

初始化函数 fib6_net_init()定义如下：
static int __net_init fib6_net_init(struct net *net)
{
    size_t size = sizeof(struct hlist_head) * FIB6_TABLE_HASHSZ;

    setup_timer(&net->ipv6.ip6_fib_timer, fib6_gc_timer_cb, (unsigned long)net);

    net->ipv6.rt6_stats = kzalloc(sizeof(*net->ipv6.rt6_stats), GFP_KERNEL);
    ...
    size = max_t(size_t, size, L1_CACHE_BYTES);
    net->ipv6.fib_table_hash = kzalloc(size, GFP_KERNEL);      /*分配管理路由选择表的散列表*/
    ...

    net->ipv6.fib6_main_tbl = kzalloc(sizeof(*net->ipv6.fib6_main_tbl), GFP_KERNEL);
                                                    /*分配网络命名空间中的主路由选择表*/
    ...
    /*初始化主路由选择表*/
    net->ipv6.fib6_main_tbl->tb6_id = RT6_TABLE_MAIN;
    net->ipv6.fib6_main_tbl->tb6_root.leaf = net->ipv6.ip6_null_entry;
    net->ipv6.fib6_main_tbl->tb6_root.fn_flags =RTN_ROOT | RTN_TL_ROOT | RTN_RTINFO;
    inet_peer_base_init(&net->ipv6.fib6_main_tbl->tb6_peers);

#ifdef CONFIG_IPV6_MULTIPLE_TABLES    /*支持策略路由，多个路由选择表*/
    net->ipv6.fib6_local_tbl = kzalloc(sizeof(*net->ipv6.fib6_local_tbl),GFP_KERNEL);
    ...
    net->ipv6.fib6_local_tbl->tb6_id = RT6_TABLE_LOCAL;
    net->ipv6.fib6_local_tbl->tb6_root.leaf = net->ipv6.ip6_null_entry;
    net->ipv6.fib6_local_tbl->tb6_root.fn_flags =RTN_ROOT | RTN_TL_ROOT | RTN_RTINFO;
    inet_peer_base_init(&net->ipv6.fib6_local_tbl->tb6_peers);
#endif
    fib6_tables_init(net);        /*初始化路由选择表，主要是将表添加到网络命名空间中的散列表*/
    return 0;
    ...
}
```

fib6_net_init()函数主要工作是为网络命名空间分配管理路由选择表的散列表，创建并初始化主路由选择表等。

●接口函数

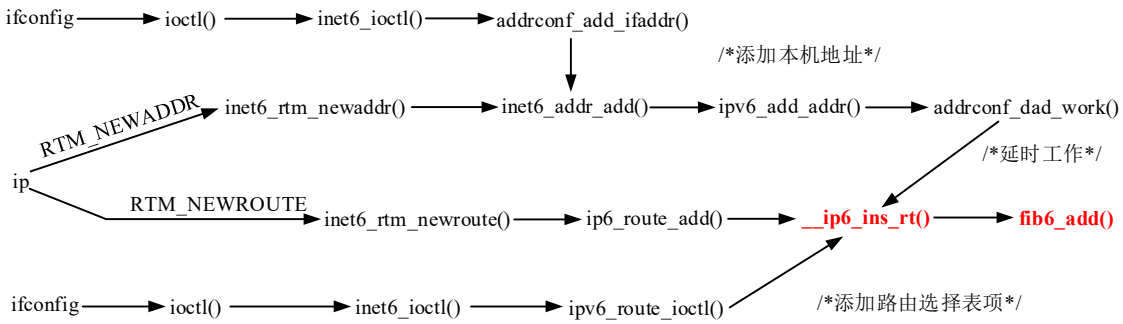
int **fib6_add**(struct fib6_node *root, struct **rt6_info** *rt,struct nl_info *info, struct mx6_config *mxc)：向路由选择表中添加 rt 指向的表项。此函数中主要分两步，一是查找或添加节点 fib6_node 实例，二是将 rt 指向 rt6_info 实例关联到节点 fib6_node 实例。

struct fib6_node ***fib6_lookup**(struct fib6_node *root, const struct in6_addr ***daddr**,const struct in6_addr *saddr)：根据目的地址（和源地址）在路由选择表中查找最长匹配前缀的 fib6_node 实例。

这两个函数都定义在/net/ipv6/ip6_fib.c 文件内，源代码请读者自行阅读。

■添加路由

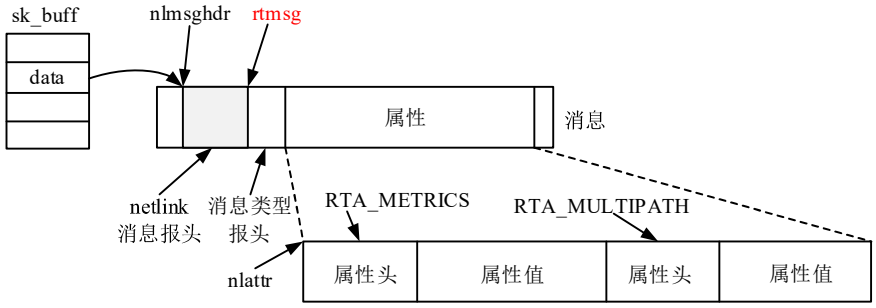
用户可以通过 ip、ifconfig 等命令直接向路由选择表添加表项（路由），在向本机网络接口配置地址时，也会在路由选择表中添加表项，函数调用关系简列如下图所示：



在直接添加路由选择表项的操作中，将创建、设置 rt6_info 实例并添加到路由选择表。在配置本机网络设备 IPv6 地址时，也将创建并设置 rt6_info 实例，在延时工作中会将路由选择表项 rt6_info 实例添加到路由选择表。

下面先以 NETLINK_ROUTE 类型 netlink 套接字 RTM_NEWROUTE 消息为例，说明添加路由选择表项的过程。在后面介绍配置网络设备本地地址时，也会向路由选择表添加表项。

IPv6 中 RTM_NEWROUTE 消息的结构与 IPv4 中消息结构相同，如下图所示，具体消息定义，属性定义请读者参考 13.1.2 小节。



在初始化函数 ip6_route_init()中注册了 RTM_NEWROUTE 消息的处理函数为 **inet6_rtm_newroute()**，定义如下（/net/ipv6/route.c）：

```
static int inet6_rtm_newroute(struct sk_buff *skb, struct nlmsghdr *nlh)
{
    struct fib6_config cfg;     /*暂存路由信息，/include/net/ip6_fib.h*/
    int err;

    err = rtm_to_fib6_config(skb, nlh, &cfg);        /*RTM_NEWROUTE 消息转 fib6_config 实例*/
    ...
    if (cfg.fc_mp)
        return ip6_route_multipath_add(&cfg);
    else
        return ip6_route_add(&cfg);     /*添加路由，/net/ipv6/route.c*/
}
```

ip6_route_add()函数用于添加路由，函数定义如下：

```
int ip6_route_add(struct fib6_config *cfg)
{
    struct mx6_config mxc = { .mx = NULL, };
    struct rt6_info *rt = NULL;
    int err;

    err = ip6_route_info_create(cfg, &rt);      /*分配设置 rt6_info 实例，/net/ipv6/route.c*/
    ...
    err = ip6_convert_metrics(&mxc, cfg);
    ...
    err = __ip6_ins_rt(rt, &cfg->fc_nlinfo, &mxc);
                    /*调用 fib6_add()函数，将 rt6_info 实例添加到路由选择表，/net/ipv6/route.c*/
    kfree(mxc.mx);
    return err;
    ...
}
```

ip6_route_add()函数主要工作是调用 ip6_route_info_create()函数分配并设置路由选择条目 rt6_info 实例，调用__ip6_ins_rt()函数将 rt6_info 实例添加到路由选择表，后者实现比较简单，请读者自行阅读源代码。下面简要看一下 ip6_route_info_create()函数的实现。

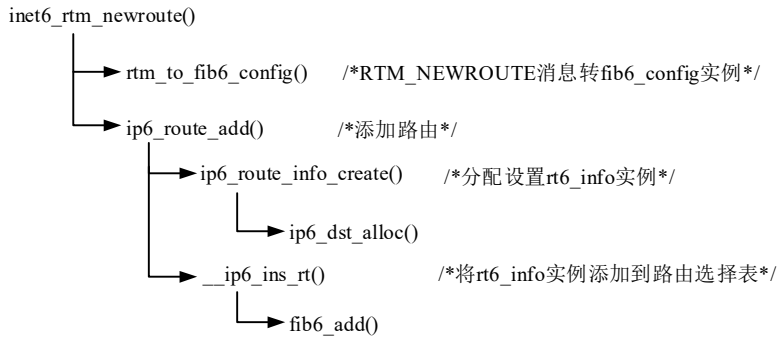
```
int ip6_route_info_create(struct fib6_config *cfg, struct rt6_info **rt_ret)
{
    int err;
    struct net *net = cfg->fc_nlinfo.nl_net;
    struct rt6_info *rt = NULL;
    struct net_device *dev = NULL;
    struct inet6_dev *idev = NULL;
    struct fib6_table *table;
    int addr_type;

    ...     /*地址有效性判断*/
    if (cfg->fc_ifindex) {
        err = -ENODEV;
        dev = dev_get_by_index(net, cfg->fc_ifindex);       /*查找网络设备 net_device 实例*/
        ...
        idev = in6_dev_get(dev);      /*查找在 IPv6 中表示网络设备的 inet6_dev 实例*/
        ...
    }

    if (cfg->fc_metric == 0)
        cfg->fc_metric = IP6_RT_PRIO_USER;

    err = -ENOBUFS;
    if (cfg->fc_nlinfo.nlh &&!(cfg->fc_nlinfo.nlh->nlmsg_flags & NLM_F_CREATE)) {
```

```
        table = fib6_get_table(net, cfg->fc_table);       /*获取路由选择表*/
        ...
    } else {
        table = fib6_new_table(net, cfg->fc_table);
    }
    ...
    rt = ip6_dst_alloc(net, NULL,(cfg->fc_flags & RTF_ADDRCONF) ? 0 : DST_NOCOUNT);
            /*从 net->ipv6.ip6_dst_ops 成员 dst_ops 实例的缓存中分配 rt6_info 实例，/net/ipv6/route.c*/

    ...
    if (cfg->fc_flags & RTF_EXPIRES)
        rt6_set_expires(rt, jiffies +clock_t_to_jiffies(cfg->fc_expires));
    else
        rt6_clean_expires(rt);

    if (cfg->fc_protocol == RTPROT_UNSPEC)
        cfg->fc_protocol = RTPROT_BOOT;
    rt->rt6i_protocol = cfg->fc_protocol;

    addr_type = ipv6_addr_type(&cfg->fc_dst);        /*目的地址类型*/

    if (addr_type & IPV6_ADDR_MULTICAST)       /*目的地址为组播地址*/
        rt->dst.input = ip6_mc_input;
    else if (cfg->fc_flags & RTF_LOCAL)        /*目的地址为本机地址*/
        rt->dst.input = ip6_input;
    else
        rt->dst.input = ip6_forward;        /*转发函数*/

    rt->dst.output = ip6_output;       /*输出本机数据包函数*/

    ipv6_addr_prefix(&rt->rt6i_dst.addr, &cfg->fc_dst, cfg->fc_dst_len);
    rt->rt6i_dst.plen = cfg->fc_dst_len;       /*前缀长度*/
    if (rt->rt6i_dst.plen == 128)
        rt->dst.flags |= DST_HOST;

#ifdef CONFIG_IPV6_SUBTREES
    ipv6_addr_prefix(&rt->rt6i_src.addr, &cfg->fc_src, cfg->fc_src_len);
    rt->rt6i_src.plen = cfg->fc_src_len;
#endif

    rt->rt6i_metric = cfg->fc_metric;

    if ((cfg->fc_flags & RTF_REJECT) ||(dev && (dev->flags & IFF_LOOPBACK) &&
         !(addr_type & IPV6_ADDR_LOOPBACK) &&
         !(cfg->fc_flags & RTF_LOCAL))) {            /*环回设备*/
        ...
```

```
        }

        if (cfg->fc_flags & RTF_GATEWAY) {      /*目的地址是网关*/
            ...
        }
        ...
        if (!ipv6_addr_any(&cfg->fc_prefsrc)) {
            if (!ipv6_chk_addr(net, &cfg->fc_prefsrc, dev, 0)) {
                err = -EINVAL;
                goto out;
            }
            rt->rt6i_prefsrc.addr = cfg->fc_prefsrc;
            rt->rt6i_prefsrc.plen = 128;
        } else
            rt->rt6i_prefsrc.plen = 0;

        rt->rt6i_flags = cfg->fc_flags;

    install_route:    /*安装路由*/
        rt->dst.dev = dev;
        rt->rt6i_idev = idev;
        rt->rt6i_table = table;
        cfg->fc_nlinfo.nl_net = dev_net(dev);
        *rt_ret = rt;
        return 0;
        ...
    }
```

ip6_route_info_create()函数调用 ip6_dst_alloc()函数从 net->ipv6.ip6_dst_ops 成员 dst_ops 实例的缓存中分配 rt6_info 结构体实例，根据目的地址类型设置 rt6_info 实例。

下图列出了 RTM_NEWROUTE 消息处理函数 inet6_rtm_newroute()的函数调用关系：



需要注意的是，在创建路由选择表项 rt6_info 实例时，就会对其中的 dst_entry 结构体成员进行设置，包括其中的 input()和 output()函数指针成员，而不是在路由选择操作中设置 dst_entry 结构体成员。input()和 output()函数是数据包下一步的处理函数。

■配置网络设备

这里说的配置网络设备，是指配置网络设备在 IPv6 网络层协议中的参数。由于一个网络设备可以传输

多种网络层协议的数据包，因此每个网络层协议需要配置网络设备特定于本网络层协议的参数。

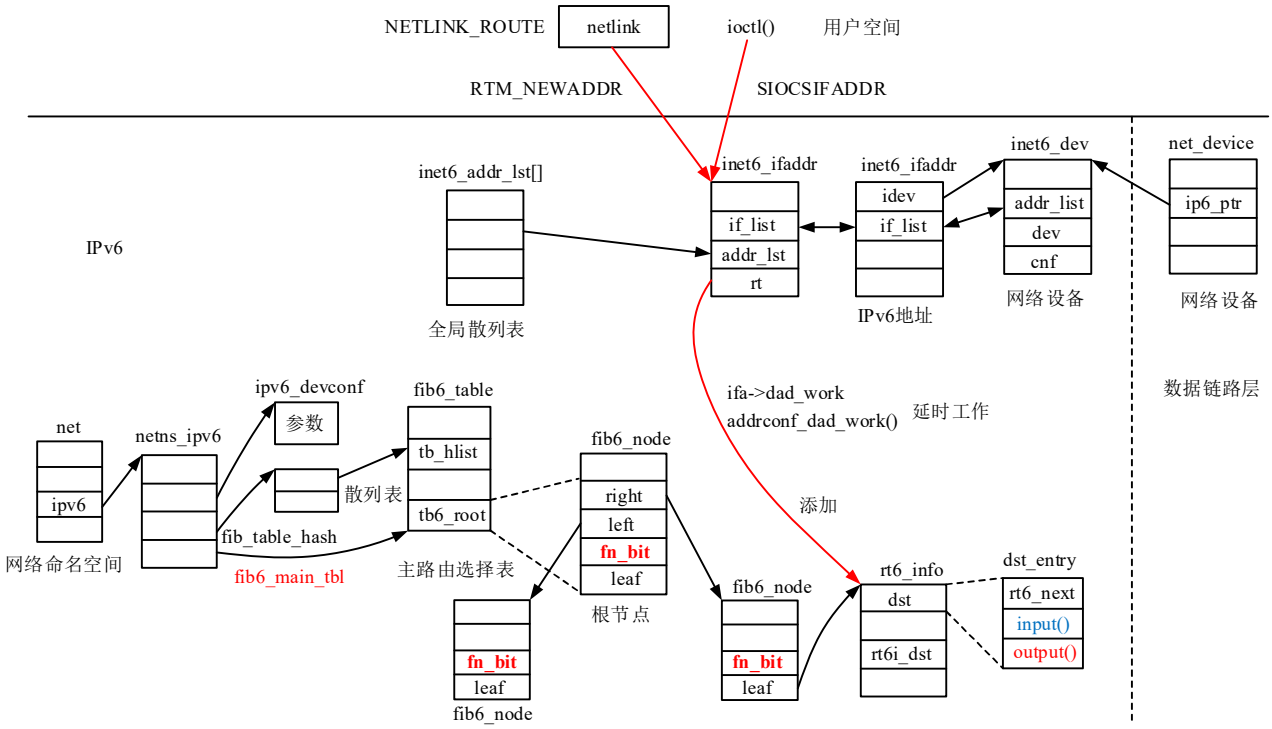
由于在向网络设备配置 IPv6 地址时，会在路由选择表中添加表项，因此在这里介绍网络设备在 IPv6 中的配置。

●概述

如下图所示，网络设备在 IPv6 中由 inet6_dev 结构体表示，在注册网络设备 net_device 实例时，将执行 netdev_chain 通知链中通知，其中 ipv6_dev_notf 通知的回调函数将为网络设备创建 inet6_dev 实例。网络设备 IPv6 地址由 inet6_ifaddr 结构体表示。一个网络设备可以有多个 IPv6 地址，因此 inet6_dev 实例中管理着一个 inet6_ifaddr 实例双链表。所有的 inet6_ifaddr 实例还由全局散列表管理。用户进程可通过 netlink 套接字或 ioctl()系统调用等向网络设备配置 IPv6 地址。

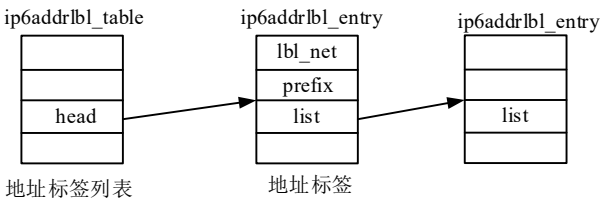
在向网络设备配置地址时，将创建、设置表示地址的 inet6_ifaddr 实例以及对应路由选择表项的 rt6_info 实例。inet6_ifaddr 实例将添加到 inet6_ifaddr 实例中的双链表和全局散列表。在延时工作中会将 rt6_info 实例添加到路由选择表。

另外，网络设备的一些元参数由 ipv6_devconf 结构体表示，它由网络命名空间管理，在创建 inet6_dev 实例时，将赋予此实例。



IPv6 中定义了地址标签，用于标识什么前缀的地址属于什么类型（用于什么用途），例如，::1/128 地址为环回地址，其标签值为 1。

地址标签相关代码位于/net/ipv6/addrlabel.c 文件内。地址标签由地址标签列表管理，ip6addrlbl_table 结构体表示地址标签列表，地址标签由 ip6addrlbl_entry 结构体表示，这两个数据结构都定义在 addrlabel.c 文件内。地址标签列表如下图所示：



地址标签列表 ip6addrlbl_table 结构体定义如下：

```
static struct ip6addrlbl_table
{
    struct hlist_head head;    /*单链表成员，管理地址标签*/
    spinlock_t lock;
    u32 seq;
}ip6addrlbl_table;
```

全局地址标签列表 ip6addrlbl_table 实例，管理了所有网络命名空间中的地址标签。

地址标签 ip6addrlbl_entry 结构体定义如下：

```
struct ip6addrlbl_entry {
    possible_net_t lbl_net;        /*网络命名空间*/
    struct in6_addr prefix;        /*地址前缀*/
    int prefixlen;          /*前缀长度*/
    int ifindex;           /*网络设备编号*/
    int addrtype;          /*地址类型*/
    u32 label;             /*地址标签值*/
    struct hlist_node list;          /*单链表成员，将实例链入地址标签列表*/
    atomic_t refcnt;
    struct rcu_head rcu;
};
```

static int **ip6addrlbl_add**(struct net *net,const struct in6_addr *prefix, int prefixlen,int ifindex, u32 label, int replace)函数用于创建地址标签实例，并添加到标签列表。net 表示网络命名空间，prefix 指向前缀地址，prefixlen 表示前缀长度，ifindex 表示网络设备编号，label 表示地址标签数值，replace 表示是否替换已有的地址标签。

在初始化函数 ipv6_addr_label_init()中会为网络命名空间添加一组由内核定义的地址标签，源代码请读者自行阅读。

用户进程还可以通过 NETLINK_ROUTE 消息，添加、删除、获取地址标签信息。ipv6_addr_label_rtnl_register()函数中注册了这些消息的处理函数，如下所示（/net/ipv6/addrlabel.c）：

```
void __init ipv6_addr_label_rtnl_register(void)
{
    __rtnl_register(PF_INET6, RTM_NEWADDRLABEL, ip6addrlbl_newdel,NULL, NULL);
    __rtnl_register(PF_INET6, RTM_DELADDRLABEL, ip6addrlbl_newdel,NULL, NULL);
    __rtnl_register(PF_INET6, RTM_GETADDRLABEL, ip6addrlbl_get,ip6addrlbl_dump, NULL);
}
```

●设备与地址

在 IPv6 网络层协议中，网络设备由 inet6_dev 结构体表示。一个网络设备既可传输 IPv4 数据包，也可传输 IPv6 数据包，因此一个网络设备既有在 IPv4 中的表示 in_device 实例，又有在 IPv6 中的表示 inet6_dev 实例。

inet6_dev 结构体定义如下（/include/net/if_inet6.h）：

```
struct inet6_dev {
    struct net_device    *dev;          /*指向网络设备 net_device 实例*/
```

```
    struct list_head    addr_list;      /*单播地址 inet6_ifaddr 实例双链表*/

    struct ifmcaddr6    *mc_list;       /*组播地址列表*/
    struct ifmcaddr6    *mc_tomb;
    spinlock_t          mc_lock;

    unsigned char       mc_qrv;         /* Query Robustness Variable */
    unsigned char       mc_gq_running;
    unsigned char       mc_ifc_count;
    unsigned char       mc_dad_count;

    unsigned long       mc_v1_seen;   /* Max time we stay in MLDv1 mode */
    unsigned long       mc_qi;          /* Query Interval */
    unsigned long       mc_qri;         /* Query Response Interval */
    unsigned long       mc_maxdelay;

    struct timer_list   mc_gq_timer; /* general query timer */
    struct timer_list   mc_ifc_timer; /* interface change timer */
    struct timer_list   mc_dad_timer;       /* dad complete mc timer */

    struct ifacaddr6    *ac_list;       /*任播地址列表*/
    rwlock_t        lock;
    atomic_t        refcnt;
    __u32           if_flags;
    int             dead;

    u8              rndid[8];
    struct timer_list   regen_timer;
    struct list_head    tempaddr_list;      /*临时时地址列表*/

    struct in6_addr     token;

    struct neigh_parms *nd_parms;       /*邻居参数*/
    struct ipv6_devconf   cnf;      /*网络设备元参数，来源于 net->ipv6.devconf_dflt*/
    struct ipv6_devstat   stats;

    struct timer_list   rs_timer;
    __u8                rs_probes;

    __u8                addr_gen_mode;
    unsigned long       tstamp;     /* ipv6InterfaceTable update timestamp */
    struct rcu_head         rcu;
};
```

inet6_dev 结构体中主要包含网络设备的各类型地址列表和一些元参数。下面主要介绍其中的单播地址管理和元参数，它们分别由 inet6_ifaddr 和 ipv6_devconf 结构体表示。

网络设备单播 IPv6 地址由 inet6_ifaddr 结构体表示，定义如下（/include/net/if_inet6.h）：

```
struct inet6_ifaddr {
    struct in6_addr     addr;       /*IPv6 地址*/
    __u32               prefix_len;     /*前缀长度*/

    __u32           valid_lft;
    __u32           prefered_lft;
    atomic_t        refcnt;     /*引用计数*/
    spinlock_t      lock;
    int             state;      /*状态*/
    __u32           flags;      /*标志*/
    __u8            dad_probes;
    __u8            stable_privacy_retry;
    __u16           scope;      /*范围*/

    unsigned long       cstamp;   /* created timestamp，时间戳*/
    unsigned long       tstamp;   /* updated timestamp，时间戳  */

    struct delayed_work     dad_work;       /*延时工作*/

    struct inet6_dev    *idev;      /*指向 inet6_dev 实例*/
    struct rt6_info     *rt;        /*指向路由选择表项 rt6_info 实例*/

    struct hlist_node   addr_lst;       /*将实例链入全局散列表*/
    struct list_head    if_list;        /*将实例链入 inet6_dev 实例中双链表*/

    struct list_head    tmp_list;       /*定时器*/
    struct inet6_ifaddr  *ifpub;
    int             regen_count;

    bool            tokenized;

    struct rcu_head         rcu;
    struct in6_addr         peer_addr;
};
```

inet6_dev 结构体中 addr_list 双链表成员管理了网络设备单播地址 inet6_ifaddr 实例。

网络设备元参数由 ipv6_devconf 结构体表示，结构体定义在/include/linux/ipv6.h 头文件，主要包含 IPv6 连接的一些元参数（整型数），结构体定义请读者自行阅读。

内核在/net/ipv6/addrconf.c 文件内定义了 ipv6_devconf 结构体实例 ipv6_devconf 和 ipv6_devconf_dflt。在创建（初始化）网络命名空间时，将会复制这两个实例的副本赋予网络命名空间 net->ipv6 成员，如下所示：

```
net->ipv6.devconf_all = all;       /*ipv6_devconf 副本*/
net->ipv6.devconf_dflt = dflt;     /*ipv6_devconf_dflt 副本*/
```

在创建 inet6_dev 实例时，net->ipv6.devconf_dflt 成员将复制给 inet6_dev.cnf 成员。

●初始化

网络设备在 IPv6 中的配置初始化函数为 addrconf_init()，它由 inet6_init()函数调用，函数定义如下：

```
int __init addrconf_init(void)        /*/net/ipv6/addrconf.c*/
{
    struct inet6_dev *idev;
    int i, err;

    err = ipv6_addr_label_init();
                        /*为网络命名空间添加一组由内核定义的地址标签，/net/ipv6/addrlabel.c*/
    ...    /*错误处理*/

    err = register_pernet_subsys(&addrconf_ops);
        /*初始化函数中复制 ipv6_devconf 和 ipv6_devconf_dflt 实例分别赋予 net->ipv6.devconf_all
          *和 net->ipv6.devconf_dflt 成员，并设置部分参数。
          */
    ...     /*错误处理*/

    addrconf_wq = create_workqueue("ipv6_addrconf");        /*创建延时工作*/
    ...       /*错误处理*/
    rtnl_lock();
    idev = ipv6_add_dev(init_net.loopback_dev);       /*添加表示环回设备的 inet6_dev 实例*/
    rtnl_unlock();
    ...      /*错误处理*/

    for (i = 0; i < IN6_ADDR_HSIZE; i++)
        INIT_HLIST_HEAD(&inet6_addr_lst[i]);       /*初始化管理 inet6_ifaddr 实例的散列表*/

    register_netdevice_notifier(&ipv6_dev_notf);
                                        /*向 netdev_chain 通知链注册 ipv6_dev_notf 通知，
                                        *通知回调函数中将创建 inet6_dev 实例（注册网络设备时）。
                                        */
    addrconf_verify();
    rtnl_af_register(&inet6_ops);       /*注册地址簇 rtnl_af_ops 结构体实例*/
    err = __rtnl_register(PF_INET6, RTM_GETLINK, NULL, inet6_dump_ifinfo,NULL);
    ...

    /*注册消息处理函数*/
    __rtnl_register(PF_INET6, RTM_NEWADDR, inet6_rtm_newaddr, NULL, NULL);
    __rtnl_register(PF_INET6, RTM_DELADDR, inet6_rtm_deladdr, NULL, NULL);
    __rtnl_register(PF_INET6, RTM_GETADDR, inet6_rtm_getaddr,inet6_dump_ifaddr, NULL);
    __rtnl_register(PF_INET6, RTM_GETMULTICAST, NULL,inet6_dump_ifmcaddr, NULL);
    __rtnl_register(PF_INET6, RTM_GETANYCAST, NULL,inet6_dump_ifacaddr, NULL);
    __rtnl_register(PF_INET6, RTM_GETNETCONF, inet6_netconf_get_devconf,
                                            inet6_netconf_dump_devconf, NULL);
```

```
    ipv6_addr_label_rtnl_register();
                    /*注册 RTM_NEWADDRLABEL 等消息处理函数，/net/ipv6/addrlabel.c*/
    return 0;
    ...
}
```

addrconf_init()函数中完成网络设备在 IPv6 中配置的初始化工作，上面源代码请读者自行阅读。这里主要介绍两项工作，一是向 netdev_chain 通知链注册 ipv6_dev_notf 通知，二是注册地址操作 netlink 消息的处理函数。

ipv6_dev_notf 通知回调函数如果是在注册网络设备时调用，将为网络设备创建 inet6_dev 实例。当用户进程通过 RTM_NEWADDR 消息向网络设备添加地址时，将为网络设备创建并注册 inet6_ifaddr 实例，并向路由选择表添加路由选择表项。下面分别介绍这两项工作的实现。

●添加设备

在注册网络设备 net_device 实例时，将执行 netdev_chain 通知链中通知，前面介绍的 ipv6_dev_notf 通知，在此时将为网络设备在 IPv6 中创建 inet6_dev 实例。

ipv6_dev_notf 通知定义如下（/net/ipv6/addrconf.c）：

```
static struct notifier_block ipv6_dev_notf = {
    .notifier_call = addrconf_notify,
};
```

通知回调函数为 addrconf_notify()，定义如下：

```
static int addrconf_notify(struct notifier_block *this, unsigned long event,void *ptr)
{
    struct net_device *dev = netdev_notifier_info_to_dev(ptr);
    struct inet6_dev *idev = __in6_dev_get(dev);
    int run_pending = 0;
    int err;

    switch (event) {
    case NETDEV_REGISTER:        /*注册网络设备*/
        if (!idev && dev->mtu >= IPV6_MIN_MTU) {
            idev = ipv6_add_dev(dev);          /*创建 inet6_dev 实例，/net/ipv6/addrconf.c*/
            ...      /*错误处理*/
        }
        break;

    case NETDEV_UP:        /*开启网络设备*/
    case NETDEV_CHANGE:
        ...
        break;

    case NETDEV_CHANGEMTU:     /*修改 MTU*/
        ...
    case NETDEV_DOWN:          /*关闭网络设备*/
    case NETDEV_UNREGISTER:
```

```
            /*移除地址*/
            addrconf_ifdown(dev, event != NETDEV_DOWN);
            break;

        case NETDEV_CHANGENAME:
            ...
            break;

        case NETDEV_PRE_TYPE_CHANGE:
        case NETDEV_POST_TYPE_CHANGE:
            addrconf_type_change(dev, event);
            break;
        }
        return NOTIFY_OK;
    }
```

在注册网络设备 net_device 实例，执行 ipv6_dev_notf 通知时，将调用 ipv6_add_dev()函数为网络设备创建 inet6_dev 实例，函数定义如下：

```
    static struct inet6_dev *ipv6_add_dev(struct net_device *dev)
    {
        struct inet6_dev *ndev;
        int err = -ENOMEM;

        ASSERT_RTNL();

        if (dev->mtu < IPV6_MIN_MTU)
            return ERR_PTR(-EINVAL);

        ndev = kzalloc(sizeof(struct inet6_dev), GFP_KERNEL);       /*分配 inet6_dev 实例（全零）*/
        ...       /*错误处理*/

        rwlock_init(&ndev->lock);
        ndev->dev = dev;              /*指向 net_device 实例*/
        INIT_LIST_HEAD(&ndev->addr_list);       /*初始化地址双链表*/
        setup_timer(&ndev->rs_timer, addrconf_rs_timer,(unsigned long)ndev);       /*设置定时器*/
        memcpy(&ndev->cnf, dev_net(dev)->ipv6.devconf_dflt, sizeof(ndev->cnf));     /*复制配置参数*/
        ndev->cnf.mtu6 = dev->mtu;
        ndev->cnf.sysctl = NULL;
        ndev->nd_parms = neigh_parms_alloc(dev, &nd_tbl);       /*邻居参数*/
        ...         /*错误处理*/
        if (ndev->cnf.forwarding)
            dev_disable_lro(dev);

        dev_hold(dev);

        if (snmp6_alloc_dev(ndev) < 0) {       /*为 idev->stats 成员分配空间，/net/ipv6/addrconf.c*/
```

```
        ...         /*如果以上分配操作出错，释放 inet6_dev 实例，返回错误码*/
    }

    if (snmp6_register_dev(ndev) < 0) {       /*在 proc 文件系统中创建目录/文件，/net/ipv6/proc.c*/
        ...       /*错误处理*/
    }

    in6_dev_hold(ndev);         /*增加引用计数 idev->refcnt*/

    if (dev->flags & (IFF_NOARP | IFF_LOOPBACK))
        ndev->cnf.accept_dad = -1;

#if IS_ENABLED(CONFIG_IPV6_SIT)
    if (dev->type == ARPHRD_SIT && (dev->priv_flags & IFF_ISATAP)) {
        pr_info("%s: Disabled Multicast RS\n", dev->name);
        ndev->cnf.rtr_solicits = 0;
    }
#endif

    INIT_LIST_HEAD(&ndev->tempaddr_list);
    setup_timer(&ndev->regen_timer, ipv6_regen_rndid, (unsigned long)ndev);
    if ((dev->flags&IFF_LOOPBACK) ||dev->type == ARPHRD_TUNNEL ||
    dev->type == ARPHRD_TUNNEL6 || dev->type == ARPHRD_SIT ||dev->type == ARPHRD_NONE)
    {
          ndev->cnf.use_tempaddr = -1;
    } else {
        in6_dev_hold(ndev);
        ipv6_regen_rndid((unsigned long) ndev);
    }

    ndev->token = in6addr_any;         /*全零地址*/

    if (netif_running(dev) && addrconf_qdisc_ok(dev))
        ndev->if_flags |= IF_READY;

    ipv6_mc_init_dev(ndev);       /*组播初始化，/net/ipv6/mcast.c*/
    ndev->tstamp = jiffies;
    err = addrconf_sysctl_register(ndev);       /*注册系统控制参数，/net/ipv6/addrconf.c*/
    ...
    rcu_assign_pointer(dev->ip6_ptr, ndev);       /*net_device->ip6_ptr=ndev*/

    /*加入接口本地所有节点组播组*/
    ipv6_dev_mc_inc(dev, &in6addr_interfacelocal_allnodes);

    /*加入所有节点组播组*/
    ipv6_dev_mc_inc(dev, &in6addr_linklocal_allnodes);
```

```
    /*加入所有路由器节点组播组（如果机器可转发流量）*/
    if (ndev->cnf.forwarding && (dev->flags & IFF_MULTICAST))
        ipv6_dev_mc_inc(dev, &in6addr_linklocal_allrouters);

    return ndev;      /*返回 inet6_dev 实例指针*/
    ...
}
```

●添加本机地址

用户进程可以通过 NETLINK_ROUTE 类型 netlink 套接字的 **RTM_NEWADDR** 消息，添加本地网络设备的 IPv6 地址，也可以通过 ioctl()系统调用添加。在添加本机地址时，会在路由选择表中添加相应的表项。下面以 RTM_NEWADDR 消息为例，介绍添加本机地址及路由选择表项的过程。

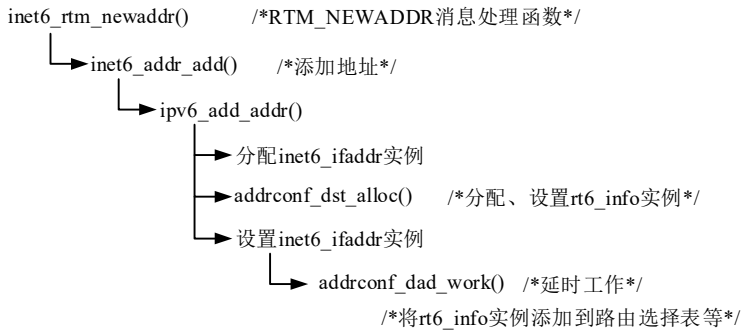
与 IPv4 中 RTM_NEWADDR 消息一样，消息类型报头由 ifaddrmsg 结构体表示，结构体定义如下：

```
struct ifaddrmsg {        /*/include/uapi/linux/if_addr.h*/
    __u8          ifa_family;      /*协议簇标识，必须是第一个成员*/
    __u8          ifa_prefixlen;   /*匹配前缀长度（掩码中 1 的位数）*/
    __u8          ifa_flags;       /*标记*/
    __u8          ifa_scope;       /*地址范围*/
    __u32         ifa_index;       /*网络接口编号（索引值）*/
};
```

RTM_NEWADDR 消息类型中属性类型定义如下（/include/uapi/linux/if_addr.h）：

```
enum {
    IFA_UNSPEC,
    IFA_ADDRESS,
    IFA_LOCAL,      /*本地地址*/
    IFA_LABEL,
    IFA_BROADCAST,    /*广播地址*/
    IFA_ANYCAST,
    IFA_CACHEINFO,      /*/include/uapi/linux/if_addr.h*/
            /*缓存信息，由 ifa_cacheinfo 结构体实例表示，可设置更详细的信息，如有效期等*/
    IFA_MULTICAST,
    IFA_FLAGS,
    __IFA_MAX,      /*属性类型最大值*/
};
```

在初始化函数 addrconf_init()中注册了 RTM_NEWADDR 消息的处理函数 **inet6_rtm_newaddr()**，函数调用关系简列如下：



在此消息处理函数中将创建、设置表示地址的 inet6_ifaddr 实例和表示对应路由选择表项的 rt6_info 实例。inet6_ifaddr 实例将添加到 inet6_dev 实例中的地址列表和全局散列表，在延时工作中会将 rt6_info 实例添加到路由选择表。

inet6_rtm_newaddr()函数定义如下（/net/ipv6/addrconf.c）：

```
static int inet6_rtm_newaddr(struct sk_buff *skb, struct nlmsghdr *nlh)
{
    struct net *net = sock_net(skb->sk);
    struct ifaddrmsg *ifm;          /*RTM_NEWADDR 消息报头*/
    struct nlattr *tb[IFA_MAX+1];       /*属性列表*/
    struct in6_addr *pfx, *peer_pfx;
    struct inet6_ifaddr *ifa;
    struct net_device *dev;
    u32 valid_lft = INFINITY_LIFE_TIME, preferred_lft = INFINITY_LIFE_TIME;
    u32 ifa_flags;
    int err;

    err = nlmsg_parse(nlh, sizeof(*ifm), tb, IFA_MAX, ifa_ipv6_policy);        /*解析消息中属性*/
    ...
    ifm = nlmsg_data(nlh);          /*消息类型报头，ifaddrmsg 实例指针*/
    pfx = extract_addr(tb[IFA_ADDRESS], tb[IFA_LOCAL], &peer_pfx);      /*前缀*/
    ...

    if (tb[IFA_CACHEINFO]) {       /*IFA_CACHEINFO 属性传递的信息*/
        struct ifa_cacheinfo *ci;

        ci = nla_data(tb[IFA_CACHEINFO]);
        valid_lft = ci->ifa_valid;
        preferred_lft = ci->ifa_prefered;
    } else {
        preferred_lft = INFINITY_LIFE_TIME;
        valid_lft = INFINITY_LIFE_TIME;
    }

    dev =    __dev_get_by_index(net, ifm->ifa_index);       /*由编号查找网络设备 net_device 实例*/
    ...

    ifa_flags = tb[IFA_FLAGS] ? nla_get_u32(tb[IFA_FLAGS]) : ifm->ifa_flags;     /*标志*/
```

```
        ifa_flags &= IFA_F_NODAD | IFA_F_HOMEADDRESS | IFA_F_MANAGETEMPADDR |
                    IFA_F_NOPREFIXROUTE | IFA_F_MCAUTOJOIN;

        ifa = ipv6_get_ifaddr(net, pfx, dev, 1);        /*查找表示地址的 inet6_ifaddr 实例*/
        if (!ifa) {      /*未找到 inet6_ifaddr 实例*/
            return inet6_addr_add(net, ifm->ifa_index, pfx, peer_pfx,ifm->ifa_prefixlen, ifa_flags,
                                  preferred_lft, valid_lft);
                                /*创建、设置 inet6_ifaddr 实例，/net/ipv6/addrconf.c*/
        }

        /*下面是处理 inet6_ifaddr 实例已存在的情形*/
        if (nlh->nlmsg_flags & NLM_F_EXCL ||!(nlh->nlmsg_flags & NLM_F_REPLACE))
            err = -EEXIST;
        else
            err = inet6_addr_modify(ifa, ifa_flags, preferred_lft, valid_lft);      /*修改地址参数*/

        in6_ifa_put(ifa);
        return err;
    }
```

由以上代码可知，当地址对应的 inet6_ifaddr 实例不存在时，将调用 **inet6_addr_add()**函数创建并设置表示地址的 inet6_ifaddr 实例，函数定义如下（/net/ipv6/addrconf.c）：

```
    static int inet6_addr_add(struct net *net, int ifindex,const struct in6_addr *pfx,const struct in6_addr
                              *peer_pfx,unsigned int plen, __u32 ifa_flags, __u32 prefered_lft, __u32 valid_lft)
    {
        struct inet6_ifaddr *ifp;
        struct inet6_dev *idev;
        struct net_device *dev;
        unsigned long timeout;
        clock_t expires;
        int scope;
        u32 flags;

        ASSERT_RTNL();

        if (plen > 128)
            return -EINVAL;

        /*检查有效期*/
        if (!valid_lft || prefered_lft > valid_lft)
            return -EINVAL;

        if (ifa_flags & IFA_F_MANAGETEMPADDR && plen != 64)
            return -EINVAL;

        dev = __dev_get_by_index(net, ifindex);       /*查找 net_device 实例*/
```

```
        ...         /*如果网络设备不存在*/

        idev = addrconf_add_dev(dev);     /*查找 inet6_dev 实例，不存在则创建，/net/ipv6/addrconf.c*/
        ...

        if (ifa_flags & IFA_F_MCAUTOJOIN) {
            int ret = ipv6_mc_config(net->ipv6.mc_autojoin_sk,true, pfx, ifindex);
            ...
        }

        scope = ipv6_addr_scope(pfx);          /*地址范围*/

        timeout = addrconf_timeout_fixup(valid_lft, HZ);
        if (addrconf_finite_timeout(timeout)) {
            expires = jiffies_to_clock_t(timeout * HZ);
            valid_lft = timeout;
            flags = RTF_EXPIRES;
        } else {
            expires = 0;
            flags = 0;
            ifa_flags |= IFA_F_PERMANENT;
        }

        timeout = addrconf_timeout_fixup(prefered_lft, HZ);
        if (addrconf_finite_timeout(timeout)) {
            if (timeout == 0)
                ifa_flags |= IFA_F_DEPRECATED;
            prefered_lft = timeout;
        }

        ifp = ipv6_add_addr(idev, pfx, peer_pfx, plen, scope, ifa_flags,valid_lft, prefered_lft);
                                            /*创建并设置 inet6_ifaddr 实例，/net/ipv6/addrconf.c*/

        if (!IS_ERR(ifp)) {
            ...
        } else if (ifa_flags & IFA_F_MCAUTOJOIN) {
            ...
        }
        return PTR_ERR(ifp);
    }
```

inet6_addr_add()函数调用 **ipv6_add_addr**()函数创建、设置 inet6_ifaddr 和 rt6_info 实例，函数代码简列如下（/net/ipv6/addrconf.c）：

```
    static struct inet6_ifaddr *ipv6_add_addr(struct inet6_dev *idev, const struct in6_addr *addr,
                const struct in6_addr *peer_addr, int pfxlen, int scope, u32 flags, u32 valid_lft, u32 prefered_lft)
    {
```

```
struct inet6_ifaddr *ifa = NULL;
struct rt6_info *rt;
unsigned int hash;
int err = 0;
int addr_type = ipv6_addr_type(addr);       /*地址类型*/

...     /*地址有效性检查*/

rcu_read_lock_bh();
...       /*有效性检查*/

spin_lock(&addrconf_hash_lock);

/*忽略相同的地址*/
if (ipv6_chk_same_addr(dev_net(idev->dev), addr, idev->dev)) {
        ...
}

ifa = kzalloc(sizeof(struct inet6_ifaddr), GFP_ATOMIC);         /*分配 inet6_ifaddr 实例*/
...
rt = addrconf_dst_alloc(idev, addr, false);
                                    /*分配、设置 rt6_info 实例，关联到环回设备，/net/ipv6/route.c*/
...
neigh_parms_data_state_setall(idev->nd_parms);       /*邻居参数*/

/*设置 inet6_ifaddr 实例*/
ifa->addr = *addr;
if (peer_addr)
        ifa->peer_addr = *peer_addr;

spin_lock_init(&ifa->lock);
INIT_DELAYED_WORK(&ifa->dad_work, addrconf_dad_work);       /*/net/ipv6/addrconf.c*/
                                                /*延时工作，将 rt6_info 实例添加到路由选择表等*/
INIT_HLIST_NODE(&ifa->addr_lst);
ifa->scope = scope;
ifa->prefix_len = pfxlen;
ifa->flags = flags | IFA_F_TENTATIVE;
ifa->valid_lft = valid_lft;
ifa->prefered_lft = prefered_lft;
ifa->cstamp = ifa->tstamp = jiffies;
ifa->tokenized = false;

ifa->rt = rt;         /*指向 rt6_info 实例*/
ifa->idev = idev;       /*指向 inet6_dev 实例*/
in6_dev_hold(idev);
in6_ifa_hold(ifa);
```

```
    hash = inet6_addr_hash(addr);     /*计算散列值*/
    hlist_add_head_rcu(&ifa->addr_lst, &inet6_addr_lst[hash]); /*将 net6_ifaddr 实例添加到散列表*/
    spin_unlock(&addrconf_hash_lock);

    write_lock(&idev->lock);
    ipv6_link_dev_addr(idev, ifa);
                    /*将 inet6_ifaddr 添加到 idev->addr_list 单播地址列表，/net/ipv6/addrconf.c*/

    if (ifa->flags&IFA_F_TEMPORARY) {        /*如果需要，添加到临时地址列表*/
        list_add(&ifa->tmp_list, &idev->tempaddr_list);
        in6_ifa_hold(ifa);
    }

    in6_ifa_hold(ifa);
    write_unlock(&idev->lock);
out2:
    ...
    return ifa;        /*返回 net6_ifaddr 实例指针*/
    ...
}
```

■路由选择函数

IPv6 在发送通道执行路由选择的函数为 **ip6_route_output()**，接收通道函数为 **ip6_route_input(skb)**。在这两个函数中都通过 flowi6 结构体来暂存路由查找过程中的参数。

flowi6 结构体定义如下（/include/net/flow.h）：

```
struct flowi6 {
    struct flowi_common     __fl_common;
    ...          /*__fl_commonk 中成员宏定义*/
    struct in6_addr          daddr;     /*目的 IPv6 地址*/
    struct in6_addr          saddr;      /*源 IPv6 地址*/
    __be32                flowlabel;
    union flowi_uli          uli;
    ...      /*uli 中成员宏定义*/
} __attribute__((__aligned__(BITS_PER_LONG/8)));
```

flowi6 结构体成员简介如下：

● **__fl_common**：flowi_common 结构体成员，定义如下：

```
struct flowi_common {
    int    flowic_oif;          /*输出网络设备编号*/
    int    flowic_iif;          /*输入网络设备编号*/
    __u32     flowic_mark;
    __u8      flowic_tos;        /*服务类型*/
    __u8      flowic_scope;      /*范围*/
    __u8      flowic_proto;
```

```
    __u8    flowic_flags;
#define FLOWI_FLAG_ANYSRC               0x01
#define FLOWI_FLAG_KNOWN_NH             0x02
    __u32   flowic_secid;
};
```

●**uli**：flowi_uli 结构体成员，定义如下：

```
union flowi_uli {
    struct {            /*端口号*/
        __be16  dport;
        __be16  sport;
    } ports;

    struct {            /*ICMP 中类型、代码*/
        __u8    type;
        __u8    code;
    } icmpt;

    struct {
        __le16  dport;
        __le16  sport;
    } dnports;

    __be32          spi;
    __be32          gre_key;

    struct {
        __u8    type;
    } mht;
};
```

●**发送路由选择函数**

发送通道路由选择函数为 **ip6_route_output()**，代码简介如下（/net/ipv6/route.c）：

```
struct dst_entry *ip6_route_output(struct net *net, const struct sock *sk,struct flowi6 *fl6)
{
    int flags = 0;

    fl6->flowi6_iif = LOOPBACK_IFINDEX;

    if ((sk && sk->sk_bound_dev_if) || rt6_need_strict(&fl6->daddr))
        flags |= RT6_LOOKUP_F_IFACE;

    if (!ipv6_addr_any(&fl6->saddr))
        flags |= RT6_LOOKUP_F_HAS_SADDR;
    else if (sk)
        flags |= rt6_srcprefs2flags(inet6_sk(sk)->srcprefs);
```

```
    return fib6_rule_lookup(net, fl6, flags, ip6_pol_route_output);      /*/net/ipv6/ip6_fib.c*/
            /*没有选择 IPV6_MULTIPLE_TABLES 配置选项，直接调用 ip6_pol_route_output()函数*/
}
```

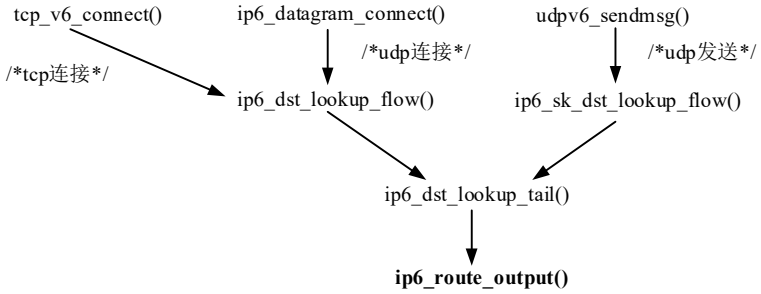
如果没有选择 IPV6_MULTIPLE_TABLES 配置选项（策略路由），函数最后调用 ip6_pol_route_output()函数（/net/ipv6/route.c）执行路由选择，此函数内又调用 ip6_pol_route()函数执行路由选择。

ip6_pol_route()函数定义如下（/net/ipv6/route.c）：

```
static struct rt6_info *ip6_pol_route(struct net *net, struct fib6_table *table, int oif,
                                      struct flowi6 *fl6, int flags)
/*table：路由选择表，oif：输出网络设备编号*/
{
    struct fib6_node *fn, *saved_fn;
    struct rt6_info *rt;
    int strict = 0;

    strict |= flags & RT6_LOOKUP_F_IFACE;
    if (net->ipv6.devconf_all->forwarding == 0)
        strict |= RT6_LOOKUP_F_REACHABLE;

    read_lock_bh(&table->tb6_lock);

    fn = fib6_lookup(&table->tb6_root, &fl6->daddr, &fl6->saddr);
                                                    /*在路由选择表中查找 fib6_node 实例*/
    saved_fn = fn;

redo_rt6_select:
    rt = rt6_select(fn, oif, strict);   /*在节点中查找合适的路由选择表项 rt6_info 实例，/net/ipv6/route.c*/
    if (rt->rt6i_nsiblings)
        rt = rt6_multipath_select(rt, fl6, oif, strict);      /*多路径路由选择*/
    if (rt == net->ipv6.ip6_null_entry) {          /*查找到的是空表项*/
        ...
    }

    if (rt == net->ipv6.ip6_null_entry || (rt->rt6i_flags & RTF_CACHE)) {
        dst_use(&rt->dst, jiffies);
        read_unlock_bh(&table->tb6_lock);

        rt6_dst_from_metrics_check(rt);
        return rt;
    } else if (unlikely((fl6->flowi6_flags & FLOWI_FLAG_KNOWN_NH) &&
                        !(rt->rt6i_flags & RTF_GATEWAY))) {

        struct rt6_info *uncached_rt;

        dst_use(&rt->dst, jiffies);
        read_unlock_bh(&table->tb6_lock);
```

```
            uncached_rt = ip6_rt_cache_alloc(rt, &fl6->daddr, NULL);       /*复制 rt6_info 实例*/
            dst_release(&rt->dst);

            if (uncached_rt)
                rt6_uncached_list_add(uncached_rt);       /*添加到 rt6_uncached_list 链表（percpu）*/
            else
                uncached_rt = net->ipv6.ip6_null_entry;

            dst_hold(&uncached_rt->dst);
            return uncached_rt;         /*返回复制的副本*/

        } else {
            struct rt6_info *pcpu_rt;

            rt->dst.lastuse = jiffies;
            rt->dst.__use++;
            pcpu_rt = rt6_get_pcpu_route(rt);     /*rt->rt6i_pcpu*/

            if (pcpu_rt) {
                read_unlock_bh(&table->tb6_lock);
            } else {
                dst_hold(&rt->dst);
                read_unlock_bh(&table->tb6_lock);
                pcpu_rt = rt6_make_pcpu_route(rt);       /*复制一个副本至 rt->rt6i_pcpu*/
                dst_release(&rt->dst);
            }

            return pcpu_rt;

        }
    }
```

下图示意了 UDP、TCP 发送通道路由选择操作过程中的函数调用关系：



在 UDP、TCP 的连接操作中将执行路由选择操作，如果 UDP 在发送数据前没有执行连接操作，则在发送操作中执行路由选择。

●**接收路由选择函数**

接收通道路由选择函数为 **ip6_route_input(skb)**，函数定义如下（/net/ipv6/route.c）：

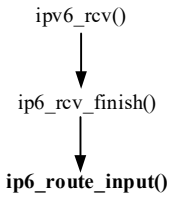
```
void ip6_route_input(struct sk_buff *skb)
{
    const struct ipv6hdr *iph = ipv6_hdr(skb);
    struct net *net = dev_net(skb->dev);
    int flags = RT6_LOOKUP_F_HAS_SADDR;
    struct flowi6 fl6 = {           /*构建 flowi6 实例*/
        .flowi6_iif = skb->dev->ifindex,
        .daddr = iph->daddr,
        .saddr = iph->saddr,
        .flowlabel = ip6_flowinfo(iph),
        .flowi6_mark = skb->mark,
        .flowi6_proto = iph->nexthdr,
    };

    skb_dst_set(skb, ip6_route_input_lookup(net, skb->dev, &fl6, flags));
                                /*调用 ip6_route_input_lookup()函数执行路由选择*/
}
```

ip6_route_input(skb)函数内将构建 flowi6 实例，然后调用 ip6_route_input_lookup()函数执行路由选择操作，函数定义如下（/net/ipv6/route.c）：

```
static struct dst_entry *ip6_route_input_lookup(struct net *net,struct net_device *dev,
                                                struct flowi6 *fl6, int flags)
{
    if (rt6_need_strict(&fl6->daddr) && dev->type != ARPHRD_PIMREG)
        flags |= RT6_LOOKUP_F_IFACE;

    return fib6_rule_lookup(net, fl6, flags, ip6_pol_route_input);      /*同发送通道路由选择函数*/
}
```

接收通道最后调用 fib6_rule_lookup()函数执行路由选择操作，同发送通道路由选择。

下图示意了 IPv6 接收通道中路由选择操作的函数调用关系：



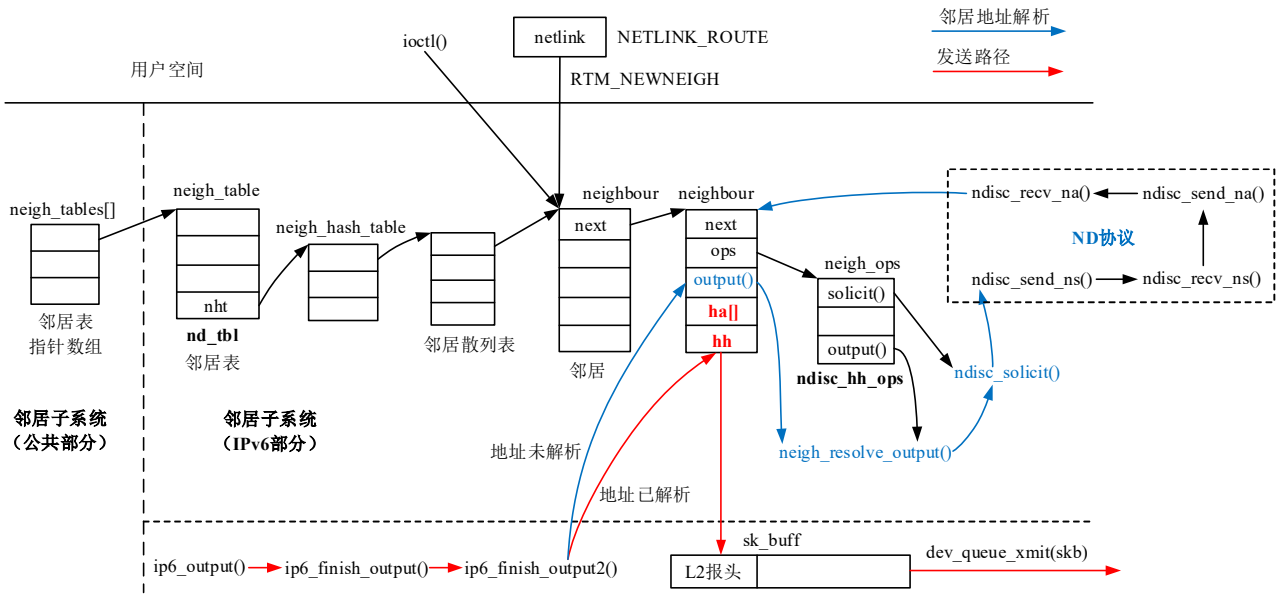
4 邻居子系统

IPv6 中邻居子系统与 IPv4 中邻居子系统结构、功能相似。邻居表、邻居等数据结构，以及创建邻居表、邻居操作等函数都是共用的，主要区别是 IPv6 通过邻居发现协议（不是 ARP 协议）来解析邻居物理地址，邻居发现协议由 ICMPv6 承担数据传输功能。

下面简要介绍 IPv6 中邻居子系统的实现，其中与 IPv4 邻居子系统相同的部分请读者参考 13.1.3 小节。

■概述

IPv6 邻居子系统框架如下图所示：



IPv6 中邻居表实例为 nd_tbl，邻居操作结构实例通常为 ndisc_hh_ops。如果邻居地址未解析，则通过邻居发现（ND）协议解析邻居物理地址，ND 协议通过 ICMPv6 的邻居请求（NS）和邻居通告（NA）类型报文实现。

IPv6 邻居子系统、邻居发现协议实现代码位于/net/ipv6/ndisc.c 文件内。

IPv6 邻居子系统初始化函数为 ndisc_init()，由 inet6_init()函数调用，函数定义如下：

```
int __init ndisc_init(void)
{
    int err;

    err = register_pernet_subsys(&ndisc_net_ops);
                        /*初始化函数为网络命名空间创建用于控制的 ICMPv6 原始套接字*/
    ...
    neigh_table_init(NEIGH_ND_TABLE, &nd_tbl);      /*初始化 IPv6 邻居表 nd_tbl*/

#ifdef CONFIG_SYSCTL
    err = neigh_sysctl_register(NULL, &nd_tbl.parms,ndisc_ifinfo_sysctl_change);
    ...
out:
#endif
    return err;
    ...
}
```

■构造邻居

内核在/net/ipv6/ndisc.c 文件内定义了 IPv6 邻居表实例 **nd_tbl**，以及邻居操作结构实例 ndisc_hh_ops 等，如下所示：

```
struct neigh_table nd_tbl = {
```

```
    .family = AF_INET6,
    .key_len =      sizeof(struct in6_addr),      /*键值长度，IPv6 地址作为键值*/
    .protocol =     cpu_to_be16(ETH_P_IPV6),
    .hash =         ndisc_hash,
    .key_eq =ndisc_key_eq,      /*键值比较函数*/
    .constructor =  ndisc_constructor,      /*邻居构造函数，创建邻居时调用*/
    .pconstructor =pndisc_constructor,
    .pdestructor =  pndisc_destructor,
    .proxy_redo = pndisc_redo,
    .id =       "ndisc_cache",
    .parms = {
        .tbl                = &nd_tbl,
        .reachable_time             = ND_REACHABLE_TIME,
        .data = {
            [NEIGH_VAR_MCAST_PROBES] = 3,
            [NEIGH_VAR_UCAST_PROBES] = 3,
            [NEIGH_VAR_RETRANS_TIME] = ND_RETRANS_TIMER,
            [NEIGH_VAR_BASE_REACHABLE_TIME] = ND_REACHABLE_TIME,
            [NEIGH_VAR_DELAY_PROBE_TIME] = 5 * HZ,
            [NEIGH_VAR_GC_STALETIME] = 60 * HZ,
            [NEIGH_VAR_QUEUE_LEN_BYTES] = 64 * 1024,
            [NEIGH_VAR_PROXY_QLEN] = 64,
            [NEIGH_VAR_ANYCAST_DELAY] = 1 * HZ,
            [NEIGH_VAR_PROXY_DELAY] = (8 * HZ) / 10,
        },
    },
    .gc_interval =    30 * HZ,
    .gc_thresh1 =   128,
    .gc_thresh2 =   512,
    .gc_thresh3 =  1024,
};
```

IPv6 邻居操作结构 ndisc_hh_ops 定义如下：

```
static const struct neigh_ops ndisc_hh_ops = {
    .family =       AF_INET6,
    .solicit =      ndisc_solicit,      /*发送邻居请求报文，解析邻居地址*/
    .error_report =     ndisc_error_report,
    .output =       neigh_resolve_output,     /*调用 solicit()函数*/
    .connected_output =     neigh_resolve_output,
};
```

另外，内核还定义了 neigh_ops 结构体 ndisc_generic_ops 和 ndisc_direct_ops 实例，在创建邻居时，将根据数据链路层报头操作结构情况将不同的实例赋予邻居实例。

IPv6 中邻居操作、发送数据包流程等与 IPv4 相同，下面主要介绍一下 IPv6 中邻居构造函数的实现，在创建邻居时，此函数用于设置邻居实例。

IPv6 中邻居构造函数 ndisc_constructor()定义如下：

```
static int ndisc_constructor(struct neighbour *neigh)
{
    struct in6_addr *addr = (struct in6_addr *)&neigh->primary_key;      /*IPv6 地址作为键值*/
    struct net_device *dev = neigh->dev;
    struct inet6_dev *in6_dev;
    struct neigh_parms *parms;
    bool is_multicast = ipv6_addr_is_multicast(addr);    /*是否是组播地址*/

    in6_dev = in6_dev_get(dev);      /*inet6_dev 实例*/
    ...
    parms = in6_dev->nd_parms;       /*邻居参数*/
    __neigh_parms_put(neigh->parms);
    neigh->parms = neigh_parms_clone(parms);

    neigh->type = is_multicast ? RTN_MULTICAST : RTN_UNICAST;
    if (!dev->header_ops) {      /*没有数据链路层报头操作结构*/
        neigh->nud_state = NUD_NOARP;
        neigh->ops = &ndisc_direct_ops;
        neigh->output = neigh_direct_output;
    } else {
        if (is_multicast) {
            neigh->nud_state = NUD_NOARP;
            ndisc_mc_map(addr, neigh->ha, dev, 1);
        } else if (dev->flags&(IFF_NOARP|IFF_LOOPBACK)) {
            neigh->nud_state = NUD_NOARP;
            memcpy(neigh->ha, dev->dev_addr, dev->addr_len);
            if (dev->flags&IFF_LOOPBACK)
                neigh->type = RTN_LOCAL;
        } else if (dev->flags&IFF_POINTOPOINT) {
            neigh->nud_state = NUD_NOARP;
            memcpy(neigh->ha, dev->broadcast, dev->addr_len);
        }
        if (dev->header_ops->cache)      /*报头操作结构中定义了 cache()函数*/
            neigh->ops = &ndisc_hh_ops;
        else
            neigh->ops = &ndisc_generic_ops;
        if (neigh->nud_state&NUD_VALID)      /*邻居有效*/
            neigh->output = neigh->ops->connected_output;
        else            /*邻居无效，需要先解析物理地址*/
            neigh->output = neigh->ops->output;
    }
    in6_dev_put(in6_dev);
    return 0;
}
```

■ND 协议实现

IPv6 中通过邻居发现（ND）协议解析邻居物理地址，而不是使用 IPv4 中的 ARP 协议。ND 协议通过 ICMPv6 报文来实现，主要包括邻居请求（NS）和邻居通告（NA）报文。

如果邻居物理地址未解析，发送主机需要发送 NS 报文，邻居主机接收到 NS 报文后，发回 NA 报文，发送主机收到 NA 报文后，依此将邻居物理地址、生成的数据链路层报头写入邻居实例。此后，发送数据包可直接使用邻居实例中缓存的数据链路层报头。

●发送邻居请求

在前面介绍的邻居操作结构 ndisc_hh_ops 实例中，solicit 函数指针成员为 **ndisc_solicit()**，此函数用于发送 NS 报文。

ndisc_solicit()函数代码简列中如下：

```
static void ndisc_solicit(struct neighbour *neigh, struct sk_buff *skb)
{
    struct in6_addr *saddr = NULL;
    struct in6_addr    mcaddr;
    struct net_device *dev = neigh->dev;
    struct in6_addr *target = (struct in6_addr *)&neigh->primary_key;        /*邻居 IPv6 地址*/
    int probes = atomic_read(&neigh->probes);       /*探测次数*/

    if (skb && ipv6_chk_addr_and_flags(dev_net(dev), &ipv6_hdr(skb)->saddr,
                                dev, 1,IFA_F_TENTATIVE|IFA_F_OPTIMISTIC))
        saddr = &ipv6_hdr(skb)->saddr;
    probes -= NEIGH_VAR(neigh->parms, UCAST_PROBES);
    if (probes < 0) {
        if (!(neigh->nud_state & NUD_VALID)) {
            ...
        }
        ndisc_send_ns(dev, neigh, target, target, saddr);
    } else if ((probes -= NEIGH_VAR(neigh->parms, APP_PROBES)) < 0) {
        neigh_app_ns(neigh);
    } else {
        addrconf_addr_solict_mult(target, &mcaddr);
                                        /*生成链路本地组播地址，/include/net/addrconf.h*/
        ndisc_send_ns(dev, NULL, target, &mcaddr, saddr); /*构建、发送 NS 报文（ICMPv6 报文）*/
    }
}
```

ndisc_solicit()函数调用 ndisc_send_ns()函数生成 NS 报文，并发送出去，源代码请读者自行阅读。

●接收邻居请求

发送主机发出 NS 报文后，邻居主机需要接收 NS 报文。由前面介绍的 ICMPv6 实现可知，ICMPv6 报文件由 icmpv6_rcv()函数接收，在此函数中，对于 NS、NA、RS、RA 及重定向报文将调用 **ndisc_rcv()**函数接收。在 ndisc_rcv()函数中又将调用 **ndisc_recv_ns()**函数接收 NS 报文。

ndisc_recv_ns()函数代码简列如下：

```
static void ndisc_recv_ns(struct sk_buff *skb)
{
```

```
struct nd_msg *msg = (struct nd_msg *)skb_transport_header(skb);
const struct in6_addr *saddr = &ipv6_hdr(skb)->saddr;
const struct in6_addr *daddr = &ipv6_hdr(skb)->daddr;
u8 *lladdr = NULL;
u32 ndoptlen = skb_tail_pointer(skb) - (skb_transport_header(skb) +offsetof(struct nd_msg, opt));
struct ndisc_options ndopts;
struct net_device *dev = skb->dev;
struct inet6_ifaddr *ifp;
struct inet6_dev *idev = NULL;
struct neighbour *neigh;
int dad = ipv6_addr_any(saddr);
bool inc;
int is_router = -1;

...      /*数据包有效性检查*/

if (ipv6_addr_is_multicast(&msg->target)) {
      ...    /*组播地址，直接返回*/
}

if (dad && !ipv6_addr_is_solict_mult(daddr)) {
      ...        /*返回*/
}

if (!ndisc_parse_options(msg->opt, ndoptlen, &ndopts)) {
      ...        /*返回*/
}

if (ndopts.nd_opts_src_lladdr) {
      lladdr = ndisc_opt_addr_data(ndopts.nd_opts_src_lladdr, dev);        /*链路层地址*/
      ...
}

inc = ipv6_addr_is_multicast(daddr);      /*是否是组播地址*/

ifp = ipv6_get_ifaddr(dev_net(dev), &msg->target, dev, 1);      /*检找地址对应的 inet6_ifaddr 实例*/
if (ifp) {        /*inet6_ifaddr 实例存在*/
      if (ifp->flags & (IFA_F_TENTATIVE|IFA_F_OPTIMISTIC)) {
            if (dad) {
                  addrconf_dad_failure(ifp);
                  return;
            } else {
                  if (!(ifp->flags & IFA_F_OPTIMISTIC))
                        goto out;
            }
      }
```

```
        idev = ifp->idev;          /*inet6_dev 实例*/
    } else {              /*inet6_ifaddr 实例不存在*/
        struct net *net = dev_net(dev);

        idev = in6_dev_get(dev);
        if (!idev) {
            return;
        }

        if (ipv6_chk_acast_addr(net, dev, &msg->target) ||
            (idev->cnf.forwarding &&(net->ipv6.devconf_all->proxy_ndp || idev->cnf.proxy_ndp) &&
             (is_router = pndisc_is_router(&msg->target, dev)) >= 0)) {
            if (!(NEIGH_CB(skb)->flags & LOCALLY_ENQUEUED) &&
                skb->pkt_type != PACKET_HOST &&inc &&
                NEIGH_VAR(idev->nd_parms, PROXY_DELAY) != 0) {
                /*任播或代理*/
                struct sk_buff *n = skb_clone(skb, GFP_ATOMIC);
                if (n)
                    pneigh_enqueue(&nd_tbl, idev->nd_parms, n);
                goto out;
            }
        } else
            goto out;
    }

    if (is_router < 0)
        is_router = idev->cnf.forwarding;

    if (dad) {
        ndisc_send_na(dev, NULL, &in6addr_linklocal_allnodes, &msg->target,
                                                !!is_router, false, (ifp != NULL), true);
        goto out;
    }

    if (inc)
        NEIGH_CACHE_STAT_INC(&nd_tbl, rcv_probes_mcast);
    else
        NEIGH_CACHE_STAT_INC(&nd_tbl, rcv_probes_ucast);

    /*为源 IPv6 地址查找或创建邻居实例*/
    neigh = __neigh_lookup(&nd_tbl, saddr, dev,!inc || lladdr || !dev->addr_len);
    if (neigh)       /*更新邻居实例*/
        neigh_update(neigh, lladdr, NUD_STALE,
                     NEIGH_UPDATE_F_WEAK_OVERRIDE|NEIGH_UPDATE_F_OVERRIDE);
    if (neigh || !dev->header_ops) {
```

```
            ndisc_send_na(dev, neigh, saddr, &msg->target,!!is_router,true, (ifp != NULL && inc), inc);
                                                                    /*构建并发送邻居通告（NA）报文*/
            if (neigh)
                neigh_release(neigh);
        }
    out:
        if (ifp)
            in6_ifa_put(ifp);
        else
            in6_dev_put(idev);
    }
```

邻居主机收到邻居请求（NS）报文后，最终调用 **ndisc_send_na()**函数构建并应答邻居通告（NA）报文，函数源代码请读者自行阅读。

●接收邻居通告

在 ndisc_rcv()函数中，发送主机将调用 **ndisc_recv_na()**函数接收邻居应答的邻居通告（NA）报文，其中包含邻居物理地址等信息，发送主机用于依此更新邻居实例。

ndisc_recv_na()函数定义如下：

```
static void ndisc_recv_na(struct sk_buff *skb)
{
    struct nd_msg *msg = (struct nd_msg *)skb_transport_header(skb);
    struct in6_addr *saddr = &ipv6_hdr(skb)->saddr;
    const struct in6_addr *daddr = &ipv6_hdr(skb)->daddr;
    u8 *lladdr = NULL;
    u32 ndoptlen = skb_tail_pointer(skb) - (skb_transport_header(skb) +offsetof(struct nd_msg, opt));
    struct ndisc_options ndopts;
    struct net_device *dev = skb->dev;
    struct inet6_ifaddr *ifp;
    struct neighbour *neigh;

    ...    /*数据包、地址检查*/
    if (ndopts.nd_opts_tgt_lladdr) {
        lladdr = ndisc_opt_addr_data(ndopts.nd_opts_tgt_lladdr, dev);      /*链路层地址*/
        ...
    }
    ifp = ipv6_get_ifaddr(dev_net(dev), &msg->target, dev, 1);     /*inet6_ifaddr 实例*/
    if (ifp) {
        ...
    }
    neigh = neigh_lookup(&nd_tbl, &msg->target, dev);        /*查找邻居实例*/

    if (neigh) {
        u8 old_flags = neigh->flags;
        struct net *net = dev_net(dev);

        if (neigh->nud_state & NUD_FAILED)
```

```
            goto out;

        if (lladdr && !memcmp(lladdr, dev->dev_addr, dev->addr_len) &&
            net->ipv6.devconf_all->forwarding && net->ipv6.devconf_all->proxy_ndp &&
            pneigh_lookup(&nd_tbl, net, &msg->target, dev, 0)) {
                goto out;
        }
        /*更新邻居实例*/
        neigh_update(neigh, lladdr,
                     msg->icmph.icmp6_solicited ? NUD_REACHABLE : NUD_STALE,
                     NEIGH_UPDATE_F_WEAK_OVERRIDE|
                     (msg->icmph.icmp6_override ? NEIGH_UPDATE_F_OVERRIDE : 0)|
                     NEIGH_UPDATE_F_OVERRIDE_ISROUTER|
                     (msg->icmph.icmp6_router ? NEIGH_UPDATE_F_ISROUTER : 0));

        if ((old_flags & ~neigh->flags) & NTF_ROUTER) {
            rt6_clean_tohost(dev_net(dev),   saddr);
        }

out:
        neigh_release(neigh);
    }
}
```

ndisc_recv_na()函数所做工作比较简单，就是根据 NA 报文中的链路层地址，调用 neigh_update()函数更新邻居实例。

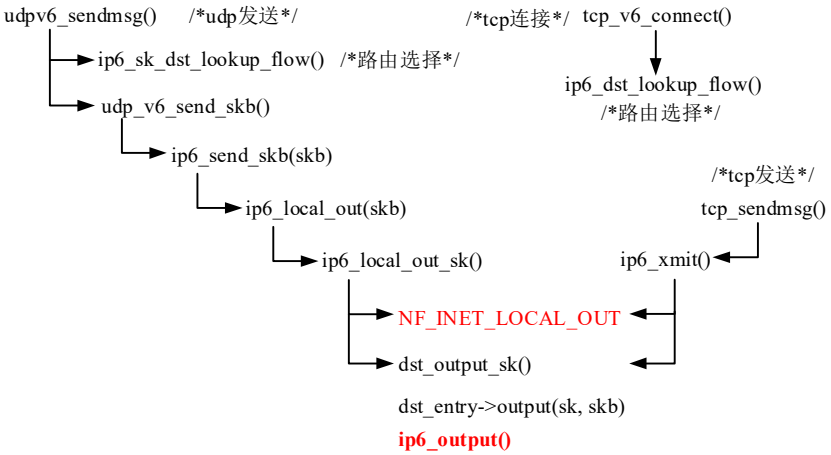
上面主要介绍了 IPv6 邻居子系统中与 IPv4 邻居子系统不同的部分，其余部分与 IPv4 邻居子系统相同，请读者参考 13.1.3 小节内容。

5 发送数据包流程

IPv6 发送数据流程与 IPv4 发送数据包流程相似，下面简要介绍一下 IPv6 发送数据包流程过程中的函数调用。

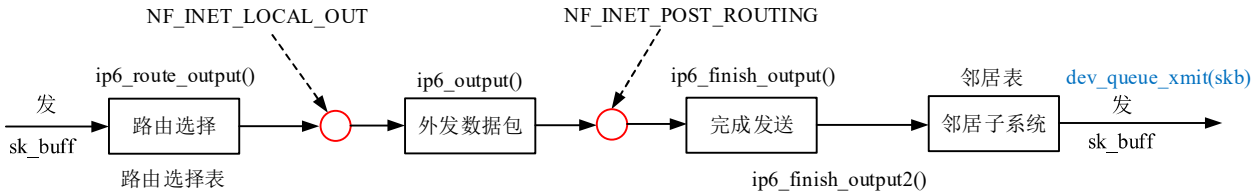
■概述

前面介绍了 UDP、TCP 发送数据包的流程，下图简要列出了函数调用关系：



UDP 如果在发送数据包前没有执行连接操作，则在发送操作中将执行路由选择，然后执行 Netfilter 钩子 NF_INET_LOCAL_OUT 注册的回调函数，最后调用 dst_entry->output()函数发送数据包。如果是本机外发的数据包则输出函数为 **ip6_output()**。TCP 在连接操作中将执行路由选择，发送数据包操作与 UDP 中发送操作相同。

下图简要示意了 IPv6 发送数据包的流程：



ip6_output()函数定义在/net/ipv6/ip6_output.c 文件内，函数代码与 IPv4 中 ip_output()函数很相似。函数内先执行 NF_INET_POST_ROUTING 钩子处注册的回调函数，然后调用 ip6_finish_output()函数。

ip6_finish_output()函数内需要处理数据包分片（见下文），然后调用 ip6_finish_output2()函数在邻居表中查找邻居实例（不存在则创建），如果邻居物理地址未解析则通过邻居发现协议解析物理地址，然后再发送数据包，随后的发送操作与 IPv4 中的发送操作相同。

■分片与重组

在 IPv6 中只允许发送主机对 IPv6 数据包进行分片，路由器转发数据包时，不能进行分片操作。在转发函数 ip6_forward()中，如果数据包过大，将直接被丢弃。

ip6_output()函数中调用的 ip6_finish_output()函数将处理数据包的分片，函数代码如下：

```
static int ip6_finish_output(struct sock *sk, struct sk_buff *skb)
{
    if ((skb->len > ip6_skb_dst_mtu(skb) && !skb_is_gso(skb)) ||
    dst_allfrag(skb_dst(skb)) ||(IP6CB(skb)->frag_max_size && skb->len > IP6CB(skb)->frag_max_size))
        return ip6_fragment(sk, skb, ip6_finish_output2);    /*分片函数，/net/ipv6/ip6_output.c*/
    else
        return ip6_finish_output2(sk, skb);
}
```

ip6_fragment()函数实现分片操作，IPv6 中的分片需要在每个分片数据包中添加分片扩展报头，对每个分片数据包调用 ip6_finish_output2()函数将其发送出去，以上函数源代码请读者自行阅读。

在接收数据包流程中，对投递到本机的分片数据包将进行重组，重组代码位于/net/ipv6/reassembly.c 文件内。

分片重组初始化函数为 ipv6_frag_init()，由 inet6_init()函数调用。初始化函数主要是初始化重组数据结构、参数，注册 inet6_protocol 结构体实例 **frag_protocol** 等。

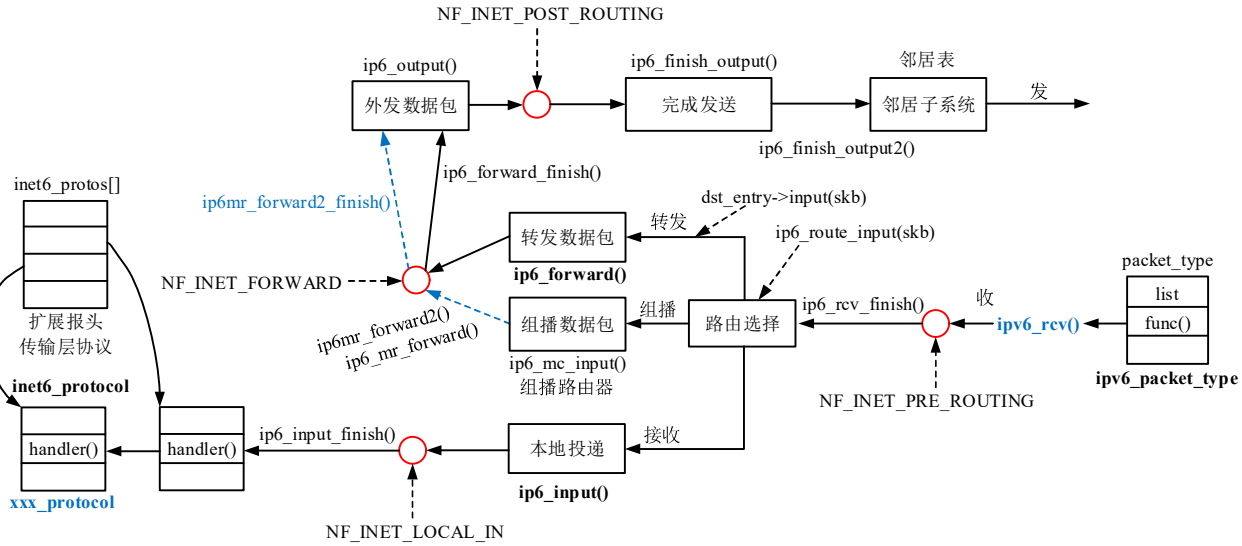
frag_protocol 实例用于处理分片扩展报头，即实现分片数据包的重组，实例定义如下：

```
static const struct inet6_protocol frag_protocol = {
    .handler   =    ipv6_frag_rcv,
    .flags         =    INET6_PROTO_NOPOLICY,
};
```

ipv6_frag_rcv()函数用于实现分片数据包的重组，重组完成后，数据包再传递到传输层，函数源代码请读者自行阅读。

6 接收数据包流程

IPv6 接收数据包流程如下图所示：



IPv6 定义并注册了 packet_type 结构体实例 **ipv6_packet_type**，其中的 func()函数成员为 **ipv6_rcv()**。这个函数是数据包进入 IPv6 网络层的入口函数。

接收通道首先对数据包调用 NF_INET_PRE_ROUTING 钩子注册的回调函数，然后将数据包交给函数 ip6_rcv_finish()处理，此函数内先进行路由选择操作，然后对数据包调用 dst_entry->input()函数。

由前面介绍的添加路由操作可知，对于投递到本机的数据包 input()函数为 **ip6_input()**，转发数据包此函数为 **ip6_forward()**，组播数据包转发函数为 ip6_mc_input()（本机配置为组播路由器）。对于转发数据包，最终都由 ip6_output()函数发出。在调用 ip6_output()函数处理前需要调用 NF_INET_FORWARD 钩子注册的回调函数。

对于投递到本机的数据包，ip6_input()函数将先调用 NF_INET_LOCAL_IN 钩子注册的回调函数，然后将数据包交给 ip6_input_finish()函数处理。

ip6_input_finish()函数需要逐层取出 IPv6 扩展报头、传输层报头，对每一层报头都要调用相应的处理函数。IPv6 中扩展报头、传输层协议都要定义并注册 inet6_protocol 结构体实例。例如，分片扩展报头的 frag_protocol 实例在/net/ipv6/reassembly.c 文件内定义并注册，其它扩展报头的 inet6_protocol 实例，在文件/net/ipv6/exthdrs.c 内定义并注册，传输层协议对应的 inet6_protocol 实例在传输层协议的实现代码中定义并注册。

在最终接收数据包的 ip6_input_finish()函数中将逐级扫描数据包中扩展报头、传输层报头，分别调用

各报头对应 inet6_protocol 实例中的处理函数，处理数据包。最后一级报头将是传输层报头，即最终数据包将传递给传输层协议处理。

7 Netfilter 子系统

Netfilter 子系统同样适用于 IPv6，如前面的发送接收流程图所示，在 IPv6 发送接收数据包路径中同样插入了以下 5 个挂载点：

●**NF_INET_PRE_ROUTING**：在接收路径中，在执行路由选择前调用此挂载点注册的回调函数。

●**NF_INET_LOCAL_IN**：在接收路径中，在将数据包投递到本机前调用此挂载点注册的回调函数。

●**NF_INET_FORWARD**：转发数据包时，在转发前调用此挂载点注册的回调函数。

●**NF_INET_LOCAL_OUT**：发送本地产生数据包时，在执行路由选择后调用此挂载点注册的回调函数。

●**NF_INET_POST_ROUTING**：发送本地数据包和转发数据包时，在完成最后发送前（到达邻居子系统前）调用此挂载点注册的回调函数。

在 IPv6 中 Netfilter 子系统同样可用于实现连接跟踪、数据包过滤、网络地址转换等功能。要实现各功能，需要各挂载点注册回调函数。

Netfilter 子系统在 IPv6 中的实现与 IPv4 相似，实现代码位于/net/ipv6/netfilter/目录下，请读者自行阅读相关源代码。

13.4 小结

第 12 章与本章介绍了 Linux 内核中网络代码的实现。网络可将其视为计算机访问外部设备（主机或进程）的一个总线或接口，只不过通过网络访问的协议比较复杂。

网络又与其它外部设备或总线不同，内核不是通过设备文件访问网络，而是通过套接字访问网络。套接字可视为进程间通信的机制，它不仅适用于访问网络，还适用于本机进程与内核之间的通信等。 每种类型的网络需要定义数据传输协议，套接字关联到网络协议，通过协议定义的操作函数访问网络。第 12 章介绍了套接字的定义、创建、操作函数接口等。

因特网（Internet）是最常见、使用最广泛的计算机网络，但并不是唯一的计算机网络。第 12 章简要介绍了因特网的物理结构，分层协议等。因特网协议簇被称为 TCP/IP，目前主要的版本主要有两个 IPv4 和 IPv6。因特网协议分为应用层、传输层、网络层、数据链路层和物理层，其中传输层、网络层、数据链路层由内核实现。

第 12 章介绍了传输层协议及其在 Linux 内核中的实现。传输层协议主要有 UDP、TCP、ICMP 等。UDP 是无连接的、不可靠的数据报传输协议。UDP 不保证数据包的可靠、正确、完整传输，数据包在传输过程中可能丢失、损坏、乱序到达等。UDP 中通信双方通过端口号来标识（主机由 IP 地址标识），数据包到达主机后，通过端口号寻找到目的套接字，并交给其接收缓存队列，供用户进程读取。

TCP 是一种面向连接，提供可靠数据传输的传输层协议。在通信前，双方进程需要建立连接，在通信过程中需要维持连接状态及连接参数，通信结束后需要关闭连接。

TCP 协议除了提供如 UDP 的交付功能外，还提供了几种附加服务。一是提供可靠数据传输服务：TCP 构建在不可靠的 IP 层上，通过使用应答、确认号、序列号、流量控制、重传等机制，TCP 确保正确、完整、按序地将数据包从发送进程交付给接收进程。二是提供拥塞控制服务：TCP 感知到通信链路和交换设备拥塞时（分组缓存溢出），调节自身发送进网络的流量速率，以防止通信链路和交换设备因被过多流量淹没而产生分组丢失。拥塞控制更像是一种提供给整个因特网的服务。TCP 要提供可靠数据传输和拥塞控制，其协议比 UDP 复杂的多。

ICMP 表示 Internet 控制消息协议（Internet Control Message Protocol）与 IP 结合使用，以便提供与 IP 协议层配置和 IP 数据包处置相关的诊断和控制信息。

第 13 章介绍了网络层（IPv6）、数据链路层协议的实现，以及简要介绍了 IPv6 的实现。网络层协议

主要用于实现主机之间数据包的传递（转发），协议维护了路由选择表和邻居表。发送接收数据包时，都要以目的 IP 地址查找路由选择表，查找匹配的表项，确定数据包下一步的走向，是投递到本机、转发、又或者是将数据包丢弃等。邻居表维护了下一跳网络设备的物理信息，主要是网络设备物理地址（L2 层报头缓存），用于发送数据包时构建数据包数据链路层报头。对于外发（转发）的数据包，网络层协议通过路由选择确定输出网络设备，从邻居实例中获取 L2 层报头，最后将数据包发送给数据链路层。接收数据包时，若是需要转发的数据包，则执行发送数据包相同的操作，如果是投递给本机的数据包，则交由传输层协议处理。

数据链路层协议主要实现网络相邻节点之间的数据传输。网络设备驱动程序及网络设备硬件实现了数据链路层协议，主要的数据链路层协议有以太网、WiFi 等。

第 13 章最后简要介绍了 IPv6 协议及其实现。IPv6 是 IPv4 的升级版，主要用于解决 IPv4 地址不足的问题，IPv4 中用 32 位表示 IP 地址，而 IPv6 中使用 128 位表示 IP 地址。IPv6 的功能及实现与 IPv4 相似。

由于作者水平有限，对网络协议、网络层代码理解不深不透，不足之处还请见谅，后面将继续学习！