

46 | 发送网络包（下）：如何表达我们想让合作伙伴做什么？

2019-07-12 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超


时长 24:01 大小 21.99M



上一节我们讲网络包的发送，讲了上半部分，也即从 VFS 层一直到 IP 层，这一节我们接着看下去，看 IP 层和 MAC 层是如何发送数据的。

解析 ip_queue_xmit 函数

从 ip_queue_xmit 函数开始，我们就要进入 IP 层的发送逻辑了。

 复制代码

```
1 int ip_queue_xmit(struct sock *sk, struct sk_buff *skb, struct flowi *fl)
2 {
3     struct inet_sock *inet = inet_sk(sk);
4     struct net *net = sock_net(sk);
5     struct ip_options_rcu *inet_opt;
6     struct flowi4 *fl4;
```

```

7     struct rtable *rt;
8     struct iphdr *iph;
9     int res;
10
11     inet_opt = rcu_dereference(inet->inet_opt);
12     fl4 = &fl->u.ip4;
13     rt = skb_rtable(skb);
14     /* Make sure we can route this packet. */
15     rt = (struct rtable *)__sk_dst_check(sk, 0);
16     if (!rt) {
17         __be32 daddr;
18         /* Use correct destination address if we have options. */
19         daddr = inet->inet_daddr;
20         .....
21         rt = ip_route_output_ports(net, fl4, sk,
22                                     daddr, inet->inet_saddr,
23                                     inet->inet_dport,
24                                     inet->inet_sport,
25                                     sk->sk_protocol,
26                                     RT_CONN_FLAGS(sk),
27                                     sk->sk_bound_dev_if);
28         if (IS_ERR(rt))
29             goto no_route;
30         sk_setup_caps(sk, &rt->dst);
31     }
32     skb_dst_set_noref(skb, &rt->dst);
33
34 packet_routed:
35     /* OK, we know where to send it, allocate and build IP header. */
36     skb_push(skb, sizeof(struct iphdr) + (inet_opt ? inet_opt->opt.optlen : 0));
37     skb_reset_network_header(skb);
38     iph = ip_hdr(skb);
39     *((__be16 *)iph) = htons((4 << 12) | (5 << 8) | (inet->tos & 0xff));
40     if (ip_dont_fragment(sk, &rt->dst) && !skb->ignore_df)
41         iph->frag_off = htons(IP_DF);
42     else
43         iph->frag_off = 0;
44     iph->ttl = ip_select_ttl(inet, &rt->dst);
45     iph->protocol = sk->sk_protocol;
46     ip_copy_addrs(iph, fl4);
47
48     /* Transport layer set skb->h.foo itself. */
49
50     if (inet_opt && inet_opt->opt.optlen) {
51         iph->ihl += inet_opt->opt.optlen >> 2;
52         ip_options_build(skb, &inet_opt->opt, inet->inet_daddr, rt, 0);
53     }
54
55     ip_select_ident_segs(net, skb, sk,
56                         skb_shinfo(skb)->gso_segs ?: 1);
57
58     /* TODO : should we use skb->sk here instead of sk ? */

```

```

59     skb->priority = sk->sk_priority;
60     skb->mark = sk->sk_mark;
61
62     res = ip_local_out(net, sk, skb);
63     .....
64 }

```

在 `ip_queue_xmit` 中，也即 IP 层的发送函数里面，有三部分逻辑。

第一部分，选取路由，也即我要发送这个包应该从哪个网卡出去。

这件事情主要由 `ip_route_output_ports` 函数完成。接下来的调用链为：
`ip_route_output_ports->ip_route_output_flow->__ip_route_output_key->ip_route_output_key_hash->ip_route_output_key_hash_rcu`。

📄 复制代码

```

1 struct rtable *ip_route_output_key_hash_rcu(struct net *net, struct flowi4 *fl4, struct
2 {
3     struct net_device *dev_out = NULL;
4     int orig_oif = fl4->flowi4_oif;
5     unsigned int flags = 0;
6     struct rtable *rth;
7     .....
8     err = fib_lookup(net, fl4, res, 0);
9     .....
10 make_route:
11     rth = __mkroute_output(res, fl4, orig_oif, dev_out, flags);
12     .....
13 }

```

`ip_route_output_key_hash_rcu` 先会调用 `fib_lookup`。

FIB 全称是 Forwarding Information Base，**转发信息表**。其实就是咱们常说的路由表。

📄 复制代码

```

1 static inline int fib_lookup(struct net *net, const struct flowi4 *flp, struct fib_resu:
2 {
3     struct fib_table *tb;
4     .....
5     tb = fib_get_table(net, RT_TABLE_MAIN);

```

```

5         if (tb)
6             err = fib_table_lookup(tb, flp, res, flags | FIB_LOOKUP_NOREF);
7         .....
8     }
9

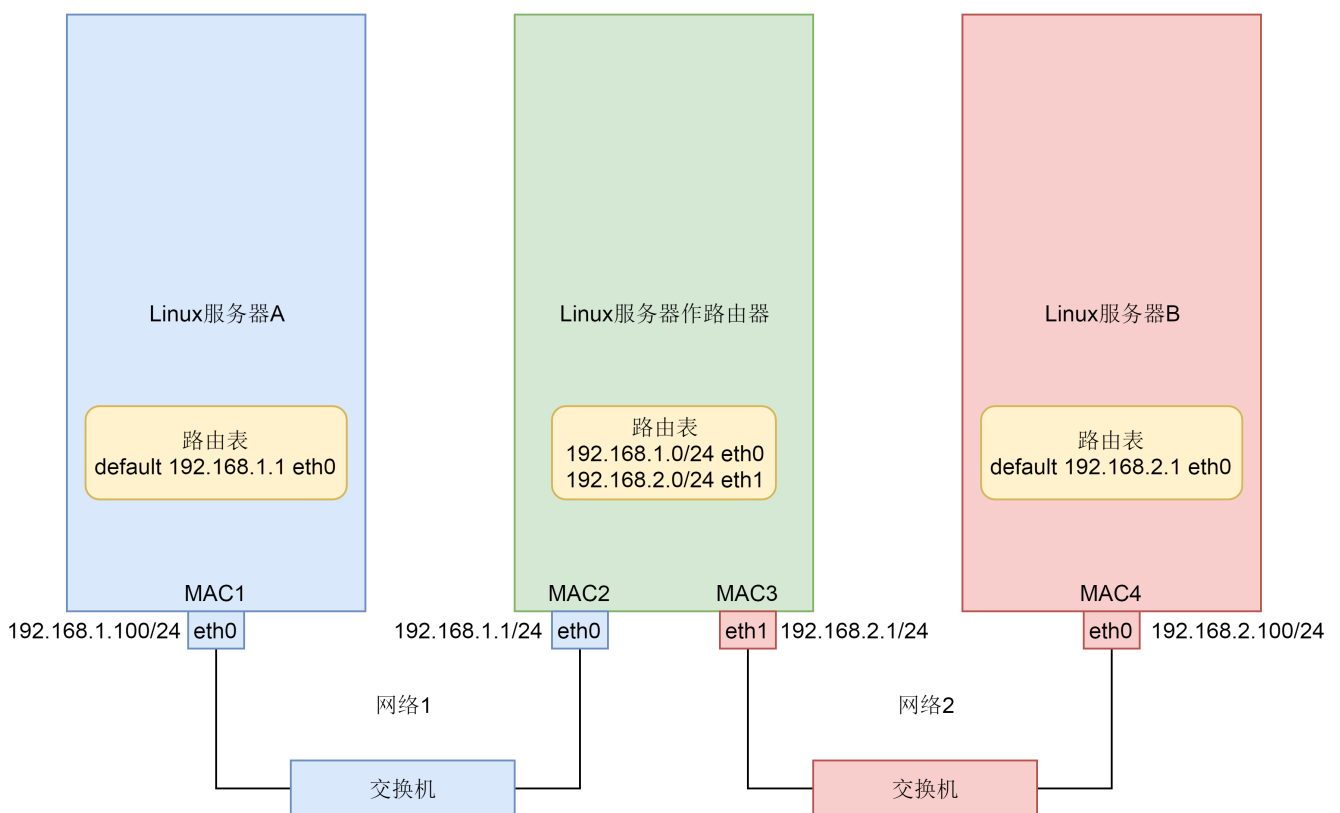
```

路由表可以有多个，一般会有一个主表，RT_TABLE_MAIN。然后 fib_table_lookup 函数在这个表里面进行查找。

路由表是一个什么样的结构呢？

路由就是在 Linux 服务器上的路由表里面配置的一条一条规则。这些规则大概是这样的：想访问某个网段，从某个网卡出去，下一跳是某个 IP。

之前我们讲过一个简单的拓扑图，里面的三台 Linux 机器的路由表都可以通过 ip route 命令查看。



复制代码

```

1 # Linux 服务器 A
2 default via 192.168.1.1 dev eth0
3 192.168.1.0/24 dev eth0 proto kernel scope link src 192.168.1.100 metric 100

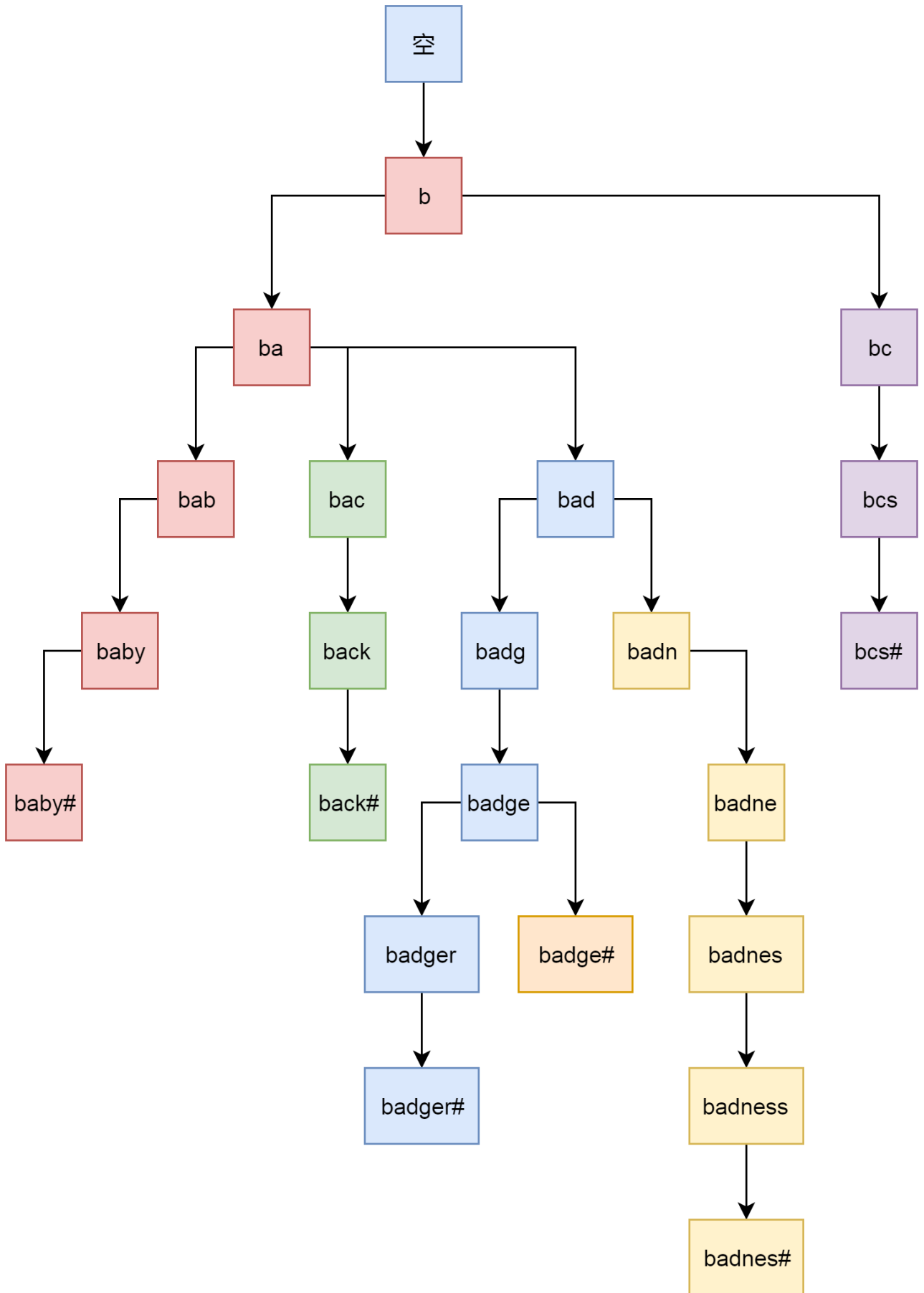
```

```
4
5 # Linux 服务器 B
6 default via 192.168.2.1 dev eth0
7 192.168.2.0/24 dev eth0 proto kernel scope link src 192.168.2.100 metric 100
8
9 # Linux 服务器做路由器
10 192.168.1.0/24 dev eth0 proto kernel scope link src 192.168.1.1
11 192.168.2.0/24 dev eth1 proto kernel scope link src 192.168.2.1
```

其实，对于两端的服务器来讲，我们没有太多路由可以选，但是对于中间的 Linux 服务器做路由器来讲，这里有两条路可以选，一个是往左面转发，一个是往右面转发，就需要路由表的查找。

fib_table_lookup 的代码逻辑比较复杂，好在注释比较清楚。因为路由表要按照前缀进行查询，希望找到最长匹配的那一个，例如 192.168.2.0/24 和 192.168.0.0/16 都能匹配 192.168.2.100/24。但是，我们应该使用 192.168.2.0/24 的这一条。


为了更方面的做这个事情，我们使用了 Trie 树这种结构。比如我们有一系列的字符串：{bcs#, badge#, baby#, back#, badger#, badness#}。之所以每个字符串都加上 #，是希望不要一个字符串成为另外一个字符串的前缀。然后我们把它们放在 Trie 树中，如下图所示：



对于将 IP 地址转成二进制放入 trie 树，也是同样的道理，可以很快进行路由的查询。

找到了路由，就知道了应该从哪个网卡发出去。

然后，ip_route_output_key_hash_rcu 会调用 __mkroute_output，创建一个 struct rtable，表示找到的路由表项。这个结构是由 rt_dst_alloc 函数分配的。

 复制代码

```
1 struct rtable *rt_dst_alloc(struct net_device *dev,
2                             unsigned int flags, u16 type,
3                             bool nopolicy, bool noxfrm, bool will_cache)
4 {
5     struct rtable *rt;
6
7     rt = dst_alloc(&ipv4_dst_ops, dev, 1, DST_OBSOLETE_FORCE_CHK,
8                  (will_cache ? 0 : DST_HOST) |
9                  (nopolicy ? DST_NOPOLICY : 0) |
10                 (noxfrm ? DST_NOXFRM : 0));
11
12     if (rt) {
13         rt->rt_genid = rt_genid_ipv4(dev_net(dev));
14         rt->rt_flags = flags;
15         rt->rt_type = type;
16         rt->rt_is_input = 0;
17         rt->rt_iif = 0;
18         rt->rt_pmtu = 0;
19         rt->rt_gateway = 0;
20         rt->rt_uses_gateway = 0;
21         rt->rt_table_id = 0;
22         INIT_LIST_HEAD(&rt->rt_uncached);
23
24         rt->dst.output = ip_output;
25         if (flags & RTCF_LOCAL)
26             rt->dst.input = ip_local_deliver;
27     }
28
29     return rt;
30 }
```


最终返回 struct rtable 实例，第一部分也就完成了。

第二部分，就是准备 IP 层的头，往里面填充内容。这就要对着 IP 层的头的格式进行理解。

版本	首部长度	服务类型TOS
总长度		
标识		
标志位	片偏移	
TTL		协议
首部校验和		
源IP地址		
目标IP地址		
选项		
数据		

在这里面，服务类型设置为 `tos`，标识位里面设置是否允许分片 `frag_off`。如果不允许，而遇到 MTU 太小过不去的情况，就发送 ICMP 报错。TTL 是这个包的存活时间，为了防止一个 IP 包迷路以后一直存活下去，每经过一个路由器 TTL 都减一，减为零则“死去”。设置 `protocol`，指的是更上层的协议，这里是 TCP。源地址和目标地址由 `ip_copy_addrs` 设置。最后，设置 `options`。

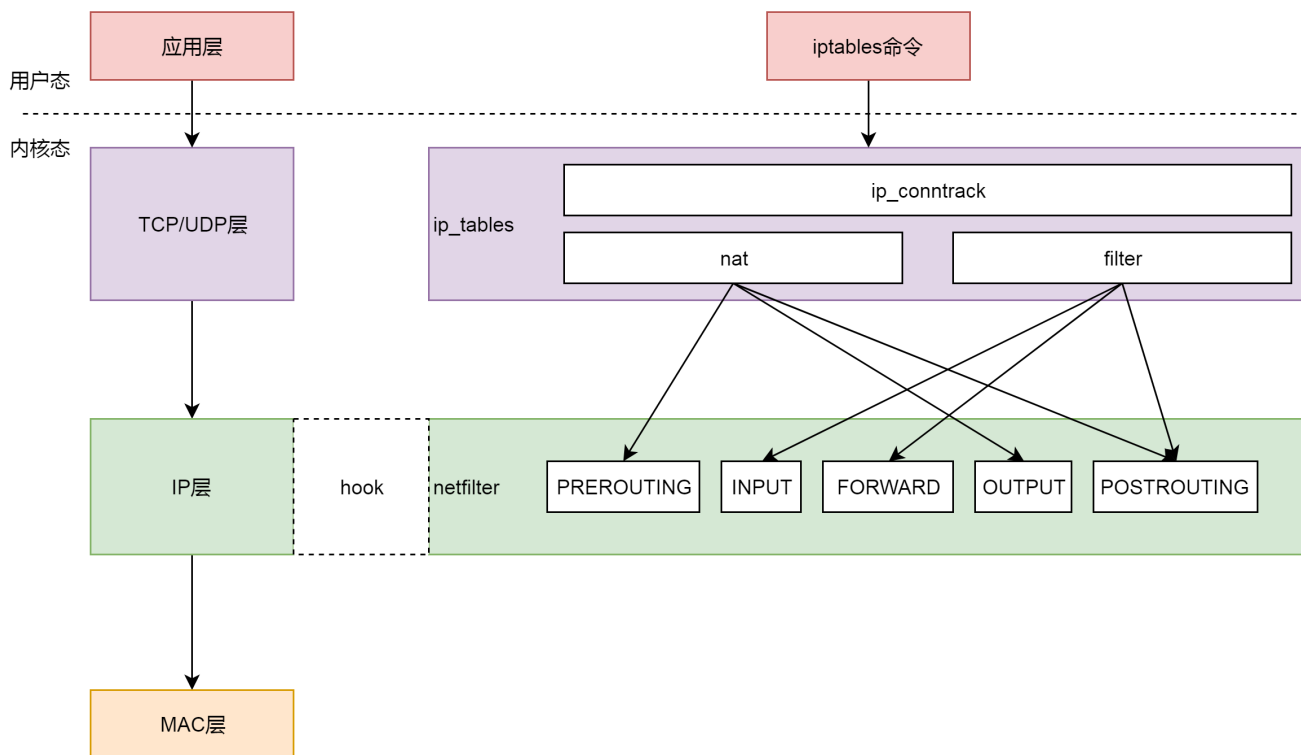
第三部分，就是调用 `ip_local_out` 发送 IP 包。

 复制代码

```
1 int ip_local_out(struct net *net, struct sock *sk, struct sk_buff *skb)
2 {
3     int err;
4
5     err = __ip_local_out(net, sk, skb);
6     if (likely(err == 1))
7         err = dst_output(net, sk, skb);
8
9     return err;
10 }
11
12 int __ip_local_out(struct net *net, struct sock *sk, struct sk_buff *skb)
13 {
14     struct iphdr *iph = ip_hdr(skb);
15     iph->tot_len = htons(skb->len);
16     skb->protocol = htons(ETH_P_IP);
17
18     return nf_hook(NFPROTO_IPV4, NF_INET_LOCAL_OUT,
19                  net, sk, skb, NULL, skb_dst(skb)->dev,
20                  dst_output);
21 }
```

`ip_local_out` 先是调用 `__ip_local_out`，然后里面调用了 `nf_hook`。这是什么呢？`nf` 的意思是 Netfilter，这是 Linux 内核的一个机制，用于在网络发送和转发的关键节点上加上 `hook` 函数，这些函数可以截获数据包，对数据包进行干预。

一个著名的实现，就是内核模块 `ip_tables`。在用户态，还有一个客户端程序 `iptables`，用命令行来干预内核的规则。



iptables 有表和链的概念，最终要的是两个表。

filter 表处理过滤功能，主要包含以下三个链。

INPUT 链：过滤所有目标地址是本机的数据包

FORWARD 链：过滤所有路过本机的数据包

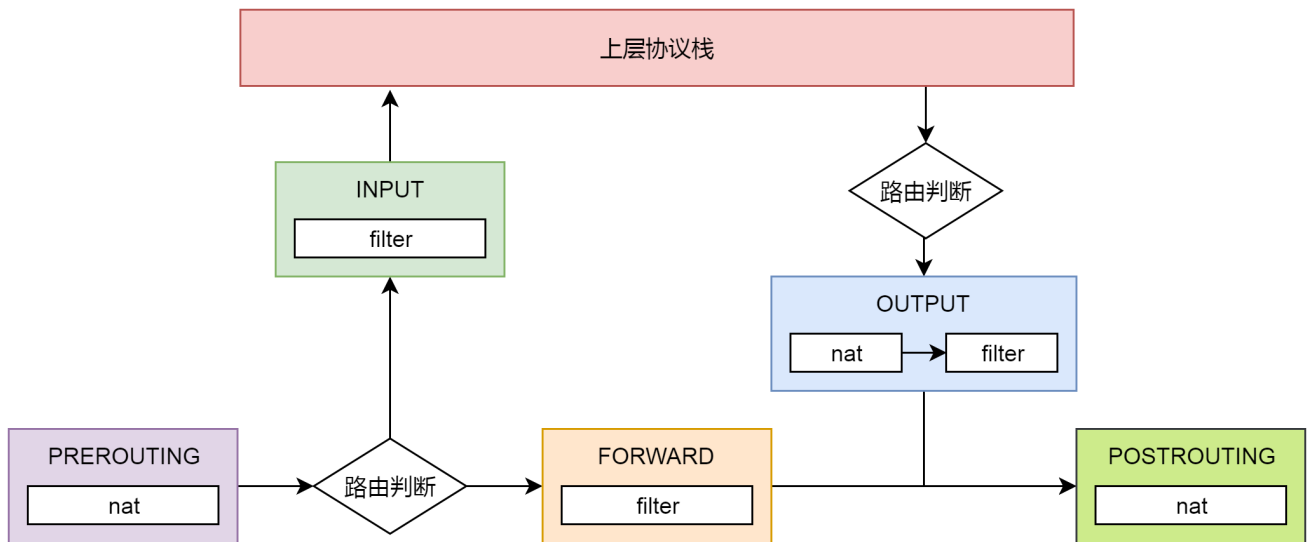
OUTPUT 链：过滤所有由本机产生的数据包

nat 表主要处理网络地址转换，可以进行 SNAT（改变源地址）、DNAT（改变目标地址），包含以下三个链。

PREROUTING 链：可以在数据包到达时改变目标地址

OUTPUT 链：可以改变本地产生的数据包的目标地址

POSTROUTING 链：在数据包离开时改变数据包的源地址



在这里，网络包马上就要发出去了，因而是 `NF_INET_LOCAL_OUT`，也即 `ouput` 链，如果用户曾经在 `iptables` 里面写过某些规则，就会在 `nf_hook` 这个函数里面起作用。

`ip_local_out` 再调用 `dst_output`，就是真正的发送数据。

[复制代码](#)

```

1 /* Output packet to network from transport. */
2 static inline int dst_output(struct net *net, struct sock *sk, struct sk_buff *skb)
3 {
4     return skb_dst(skb)->output(net, sk, skb);
5 }

```

这里调用的就是 `struct rtable` 成员 `dst` 的 `ouput` 函数。在 `rt_dst_alloc` 中，我们可以看到，`output` 函数指向的是 `ip_output`。

[复制代码](#)

```


1 int ip_output(struct net *net, struct sock *sk, struct sk_buff *skb)
2 {
3     struct net_device *dev = skb_dst(skb)->dev;
4     skb->dev = dev;
5     skb->protocol = htons(ETH_P_IP);
6
7     return NF_HOOK_COND(NFPROTO_IPV4, NF_INET_POST_ROUTING,
8                         net, sk, skb, NULL, dev,
9                         ip_finish_output,
10                        !(IPCB(skb)->flags & IPSKB_REROUTED));
11 }

```

在 `ip_output` 里面，我们又看到了熟悉的 `NF_HOOK`。这一次是 `NF_INET_POST_ROUTING`，也即 `POSTROUTING` 链，处理完之后，调用 `ip_finish_output`。

解析 `ip_finish_output` 函数

从 `ip_finish_output` 函数开始，发送网络包的逻辑由第三层到达第二层。`ip_finish_output` 最终调用 `ip_finish_output2`。

 复制代码

```
1 static int ip_finish_output2(struct net *net, struct sock *sk, struct sk_buff *skb)
2 {
3     struct dst_entry *dst = skb_dst(skb);
4     struct rtable *rt = (struct rtable *)dst;
5     struct net_device *dev = dst->dev;
6     unsigned int hh_len = LL_RESERVED_SPACE(dev);
7     struct neighbour *neigh;
8     u32 nexthop;
9     .....
10    nexthop = (__force u32) rt_nexthop(rt, ip_hdr(skb)->daddr);
11    neigh = __ipv4_neigh_lookup_noref(dev, nexthop);
12    if (unlikely(!neigh))
13        neigh = __neigh_create(&arp_tbl, &nexthop, dev, false);
14    if (!IS_ERR(neigh)) {
15        int res;
16        sock_confirm_neigh(skb, neigh);
17        res = neigh_output(neigh, skb);
18        return res;
19    }
20    .....
21 }
```

在 `ip_finish_output2` 中，先找到 `struct rtable` 路由表里面的下一跳，下一跳一定和本机在同一个局域网中，可以通过二层进行通信，因而通过 `__ipv4_neigh_lookup_noref`，查找如何通过二层访问下一跳。

 复制代码


```
1 static inline struct neighbour *__ipv4_neigh_lookup_noref(struct net_device *dev, u32 k
2 {
```

```

3         return __neigh_lookup_noref(&arp_tbl, neigh_key_eq32, arp_hashfn, &key, dev);
4     }

```

`__ipv4_neigh_lookup_noref` 是从本地的 ARP 表中查找下一跳的 MAC 地址。ARP 表的定义如下：


 复制代码

```

1 struct neigh_table arp_tbl = {
2     .family      = AF_INET,
3     .key_len      = 4,
4     .protocol     = cpu_to_be16(ETH_P_IP),
5     .hash         = arp_hash,
6     .key_eq       = arp_key_eq,
7     .constructor  = arp_constructor,
8     .proxy_redo   = parp_redo,
9     .id           = "arp_cache",
10    .....
11    .gc_interval   = 30 * HZ,
12    .gc_thresh1    = 128,
13    .gc_thresh2    = 512,
14    .gc_thresh3    = 1024,
15 };

```

如果在 ARP 表中没有找到相应的项，则调用 `__neigh_create` 进行创建。

 复制代码

```

1 struct neighbour *__neigh_create(struct neigh_table *tbl, const void *pkey, struct net_device *dev)
2 {
3     u32 hash_val;
4     int key_len = tbl->key_len;
5     int error;
6     struct neighbour *n1, *rc, *n = neigh_alloc(tbl, dev);
7     struct neigh_hash_table *nht;
8
9     memcpy(n->primary_key, pkey, key_len);
10    n->dev = dev;
11    dev_hold(dev);
12
13    /* Protocol specific setup. */
14    if (tbl->constructor && (error = tbl->constructor(n)) < 0) {
15        .....
16    }


```

```

17 .....
18     if (atomic_read(&tbl->entries) > (1 << nht->hash_shift))
19         nht = neigh_hash_grow(tbl, nht->hash_shift + 1);
20
21     hash_val = tbl->hash(pkey, dev, nht->hash_rnd) >> (32 - nht->hash_shift);
22
23     for (n1 = rcu_dereference_protected(nht->hash_buckets[hash_val],
24                                         lockdep_is_held(&tbl->lock));
25          n1 != NULL;
26          n1 = rcu_dereference_protected(n1->next,
27                                         lockdep_is_held(&tbl->lock))) {
28         if (dev == n1->dev && !memcmp(n1->primary_key, pkey, key_len)) {
29             if (want_ref)
30                 neigh_hold(n1);
31             rc = n1;
32             goto out_tbl_unlock;
33         }
34     }
35 .....
36     rcu_assign_pointer(n->next,
37                         rcu_dereference_protected(nht->hash_buckets[hash_val],
38                                                   lockdep_is_held(&tbl->lock)));
39     rcu_assign_pointer(nht->hash_buckets[hash_val], n);
40 .....
41 }

```

`__neigh_create` 先调用 `neigh_alloc`，创建一个 `struct neighbour` 结构，用于维护 MAC 地址和 ARP 相关的信息。这个名字也很好理解，大家都是在一个局域网里面，可以通过 MAC 地址访问到，当然是邻居了。

 复制代码

```

1 static struct neighbour *neigh_alloc(struct neigh_table *tbl, struct net_device *dev)
2 {
3     struct neighbour *n = NULL;
4     unsigned long now = jiffies;
5     int entries;
6     .....
7     n = kzalloc(tbl->entry_size + dev->neigh_priv_len, GFP_ATOMIC);
8     if (!n)
9         goto out_entries;
10
11     __skb_queue_head_init(&n->arp_queue);
12     rwlock_init(&n->lock);
13     seqlock_init(&n->ha_lock);
14     n->updated      = n->used = now;
15     n->nud_state     = NUD_NONE;

```


```

16         n->output          = neigh_blackhole;
17         seqlock_init(&n->hh.hh_lock);
18         n->parms            = neigh_parms_clone(&tbl->parms);
19         setup_timer(&n->timer, neigh_timer_handler, (unsigned long)n);
20
21         NEIGH_CACHE_STAT_INC(tbl, allocs);
22         n->tbl              = tbl;
23         refcount_set(&n->refcnt, 1);
24         n->dead              = 1;
25         .....
26     }

```

在 `neigh_alloc` 中，我们先分配一个 `struct neighbour` 结构并且初始化。这里面比较重要的有两个成员，一个是 `arp_queue`，所以上层想通过 ARP 获取 MAC 地址的任务，都放在这个队列里面。另一个是 `timer` 定时器，我们设置成，过一段时间就调用 `neigh_timer_handler`，来处理这些 ARP 任务。

`__neigh_create` 然后调用了 `arp_tbl` 的 constructor 函数，也即调用了 `arp_constructor`，在这里面定义了 ARP 的操作 `arp_hh_ops`。

 复制代码

```


1 static int arp_constructor(struct neighbour *neigh)
2 {
3     __be32 addr = *(__be32 *)neigh->primary_key;
4     struct net_device *dev = neigh->dev;
5     struct in_device *in_dev;
6     struct neigh_parms *parms;
7     .....
8     neigh->type = inet_addr_type_dev_table(dev_net(dev), dev, addr);
9
10    parms = in_dev->arp_parms;
11    __neigh_parms_put(neigh->parms);
12    neigh->parms = neigh_parms_clone(parms);
13    .....
14    neigh->ops = &arp_hh_ops;
15    .....
16    neigh->output = neigh->ops->output;
17    .....
18 }
19
20 static const struct neigh_ops arp_hh_ops = {
21     .family =          AF_INET,
22     .solicit =         arp_solicit,
23     .error_report =    arp_error_report,
24     .output =         neigh_resolve_output,

```

```
25         .connected_output = neigh_resolve_output,  
26     };
```


`__neigh_create` 最后是将创建的 `struct neighbour` 结构放入一个哈希表，从里面的代码逻辑比较容易看出，这是一个数组加链表的链式哈希表，先计算出哈希值 `hash_val`，得到相应的链表，然后循环这个链表找到对应的项，如果找不到就在最后插入一项。

我们回到 `ip_finish_output2`，在 `__neigh_create` 之后，会调用 `neigh_output` 发送网络包。

 复制代码


```
1 static inline int neigh_output(struct neighbour *n, struct sk_buff *skb)  
2 {  
3     .....  
4     return n->output(n, skb);  
5 }  
6
```

按照上面对于 `struct neighbour` 的操作函数 `arp_hh_ops` 的定义，`output` 调用的是 `neigh_resolve_output`。

 复制代码

```
1 int neigh_resolve_output(struct neighbour *neigh, struct sk_buff *skb)  
2 {  
3     if (!neigh_event_send(neigh, skb)) {  
4     .....  
5         rc = dev_queue_xmit(skb);  
6     }  
7     .....  
8 }
```

在 `neigh_resolve_output` 里面，首先 `neigh_event_send` 触发一个事件，看能否激活 ARP。

 复制代码



```

1 int __neigh_event_send(struct neighbour *neigh, struct sk_buff *skb)
2 {
3     int rc;
4     bool immediate_probe = false;
5
6     if (!(neigh->nud_state & (NUD_STALE | NUD_INCOMPLETE))) {
7         if (NEIGH_VAR(neigh->parms, MCAST_PROBES) +
8             NEIGH_VAR(neigh->parms, APP_PROBES)) {
9             unsigned long next, now = jiffies;
10
11             atomic_set(&neigh->probes,
12                        NEIGH_VAR(neigh->parms, UCAST_PROBES));
13             neigh->nud_state = NUD_INCOMPLETE;
14             neigh->updated = now;
15             next = now + max(NEIGH_VAR(neigh->parms, RETRANS_TIME),
16                             HZ/2);
17             neigh_add_timer(neigh, next);
18             immediate_probe = true;
19         }
20         .....
21     } else if (neigh->nud_state & NUD_STALE) {
22         neigh_dbg(2, "neigh %p is delayed\n", neigh);
23         neigh->nud_state = NUD_DELAY;
24         neigh->updated = jiffies;
25         neigh_add_timer(neigh, jiffies +
26                         NEIGH_VAR(neigh->parms, DELAY_PROBE_TIME));
27     }
28
29     if (neigh->nud_state == NUD_INCOMPLETE) {
30         if (skb) {
31             .....
32             __skb_queue_tail(&neigh->arp_queue, skb);
33             neigh->arp_queue_len_Bytes += skb->truesize;
34         }
35         rc = 1;
36     }
37 out_unlock_bh:
38     if (immediate_probe)
39         neigh_probe(neigh);
40     .....
41 }

```

在 `__neigh_event_send` 中，激活 ARP 分两种情况，第一种情况是马上激活，也即 `immediate_probe`。另一种情况是延迟激活则仅仅设置一个 timer。然后将 ARP 包放在 `arp_queue` 上。如果马上激活，就直接调用 `neigh_probe`；如果延迟激活，则定时器到了就会触发 `neigh_timer_handler`，在这里面还是会调用 `neigh_probe`。

我们就来看 neigh_probe 的实现，在这里面会从 arp_queue 中拿出 ARP 包来，然后调用 struct neighbour 的 solicit 操作。

 复制代码

```
1 static void neigh_probe(struct neighbour *neigh)
2     __releases(neigh->lock)
3 {
4     struct sk_buff *skb = skb_peek_tail(&neigh->arp_queue);
5     .....
6     if (neigh->ops->solicit)
7         neigh->ops->solicit(neigh, skb);
8     .....
9 }
```

按照上面对于 struct neighbour 的操作函数 arp_hh_ops 的定义，solicit 调用的是 arp_solicit，在这里我们可以找到对于 arp_send_dst 的调用，创建并发送一个 arp 包，得到结果放在 struct dst_entry 里面。

 复制代码

```
1 static void arp_send_dst(int type, int ptype, __be32 dest_ip,
2                          struct net_device *dev, __be32 src_ip,
3                          const unsigned char *dest_hw,
4                          const unsigned char *src_hw,
5                          const unsigned char *target_hw,
6                          struct dst_entry *dst)
7 {
8     struct sk_buff *skb;
9     .....
10    skb = arp_create(type, ptype, dest_ip, dev, src_ip,
11                    dest_hw, src_hw, target_hw);
12    .....
13    skb_dst_set(skb, dst_clone(dst));
14    arp_xmit(skb);
15 }
16
```

我们回到 neigh_resolve_output 中，当 ARP 发送完毕，就可以调用 dev_queue_xmit 发送二层网络包了。

 复制代码

```


1 /**
2  *      __dev_queue_xmit - transmit a buffer
3  *      @skb: buffer to transmit
4  *      @accel_priv: private data used for L2 forwarding offload
5  *
6  *      Queue a buffer for transmission to a network device.
7  */
8 static int __dev_queue_xmit(struct sk_buff *skb, void *accel_priv)
9 {
10     struct net_device *dev = skb->dev;
11     struct netdev_queue *txq;
12     struct Qdisc *q;
13     .....
14     txq = netdev_pick_tx(dev, skb, accel_priv);
15     q = rcu_dereference_bh(txq->qdisc);
16
17     if (q->enqueue) {
18         rc = __dev_xmit_skb(skb, q, dev, txq);
19         goto out;
20     }
21     .....
22 }

```

就像咱们在讲述硬盘块设备的时候讲过，每个块设备都有队列，用于将内核的数据放到队列里面，然后设备驱动从队列里面取出后，将数据根据具体设备的特性发送给设备。

网络设备也是类似的，对于发送来说，有一个发送队列 `struct netdev_queue *txq`。

这里还有另一个变量叫做 `struct Qdisc`，这个是什么呢？如果我们在一台 Linux 机器上运行 `ip addr`，我们能看到对于一个网卡，都有下面的输出。

 复制代码

```

1 # ip addr
2 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen :
3     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
4     inet 127.0.0.1/8 scope host lo
5         valid_lft forever preferred_lft forever
6     inet6 ::1/128 scope host
7         valid_lft forever preferred_lft forever
8 2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc pfifo_fast state UP group def:
9     link/ether fa:16:3e:75:99:08 brd ff:ff:ff:ff:ff:ff
10    inet 10.173.32.47/21 brd 10.173.39.255 scope global noprefixroute dynamic eth0
11        valid_lft 67104sec preferred_lft 67104sec
12    inet6 fe80::f816:3eff:fe75:9908/64 scope link

```

这里面有个关键字 `qdisc pfifo_fast` 是什么意思呢？`qdisc` 全称是 `queueing discipline`，中文叫排队规则。内核如果需要通过某个网络接口发送数据包，都需要按照为这个接口配置的 `qdisc`（排队规则）把数据包加入队列。

最简单的 `qdisc` 是 `pfifo`，它不对进入的数据包做任何的处理，数据包采用先入先出的方式通过队列。`pfifo_fast` 稍微复杂一些，它的队列包括三个波段（`band`）。在每个波段里面，使用先进先出规则。

三个波段的优先级也不相同。`band 0` 的优先级最高，`band 2` 的最低。如果 `band 0` 里面有数据包，系统就不会处理 `band 1` 里面的数据包，`band 1` 和 `band 2` 之间也是一样。

数据包是按照服务类型（`Type of Service`，`TOS`）被分配到三个波段里面的。`TOS` 是 IP 头里面的一个字段，代表了当前的包是高优先级的，还是低优先级的。

`pfifo_fast` 分为三个先入先出的队列，我们能称为三个 `Band`。根据网络包里面的 `TOS`，看这个包到底应该进入哪个队列。`TOS` 总共四位，每一位表示的意思不同，总共十六种类型。

		4位 TOS					
pfifo_fast	Band 0（最高）	0	0	0	0	正常	
		0	0	0	1	最小代价	
	queuing	Band 1（其次）	0	0	1	0	最大高可靠
			0	1	0	0	最大吞吐量
discipline	Band 2（最次）	1	0	0	0	最小时延	

		tc qdisc show dev eth0															
TOS		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
priomap (band)		1	2	2	2	1	2	0	0	1	1	1	1	1	1	1	1

通过命令行 `tc qdisc show dev eth0`，我们可以输出结果 `priomap`，也是十六个数字。在 0 到 2 之间，和 `TOS` 的十六种类型对应起来。不同的 `TOS` 对应不同的队列。其中 `Band 0` 优先级最高，发送完毕后才轮到 `Band 1` 发送，最后才是 `Band 2`。

```

1 # tc qdisc show dev eth0
2 qdisc pfifo_fast 0: root refcnt 2 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1

```

接下来，__dev_xmit_skb 开始进行网络包发送。

```

1 static inline int __dev_xmit_skb(struct sk_buff *skb, struct Qdisc *q,
2                                 struct net_device *dev,
3                                 struct netdev_queue *txq)
4 {
5     .....
6     rc = q->enqueue(skb, q, &to_free) & NET_XMIT_MASK;
7     if (qdisc_run_begin(q)) {
8         .....
9         __qdisc_run(q);
10    }
11    .....
12 }
13
14 void __qdisc_run(struct Qdisc *q)
15 {
16     int quota = dev_tx_weight;
17     int packets;
18     while (qdisc_restart(q, &packets)) {
19         /*
20          * Ordered by possible occurrence: Postpone processing if
21          * 1. we've exceeded packet quota
22          * 2. another process needs the CPU;
23          */
24         quota -= packets;
25         if (quota <= 0 || need_resched()) {
26             __netif_schedule(q);
27             break;
28         }
29     }
30     qdisc_run_end(q);
31 }

```

__dev_xmit_skb 会将请求放入队列，然后调用 __qdisc_run 处理队列中的数据。

qdisc_restart 用于数据的发送。根据注释中的说法，qdisc 的另一个功能是用于控制网络包的发送速度，因而如果超过速度，就需要重新调度，则会调用 __netif_schedule。

```

1 static void __netif_reschedule(struct Qdisc *q)
2 {
3     struct softnet_data *sd;
4     unsigned long flags;
5     local_irq_save(flags);
6     sd = this_cpu_ptr(&softnet_data);
7     q->next_sched = NULL;
8     *sd->output_queue_tailp = q;
9     sd->output_queue_tailp = &q->next_sched;
10    raise_softirq_irqoff(NET_TX_SOFTIRQ);
11    local_irq_restore(flags);
12 }

```

`__netif_schedule` 会调用 `__netif_reschedule`，发起一个软中断 `NET_TX_SOFTIRQ`。咱们讲设备驱动程序的时候讲过，设备驱动程序处理中断，分两个过程，一个是屏蔽中断的关键处理逻辑，一个是延迟处理逻辑。当时说工作队列是延迟处理逻辑的处理方案，软中断也是一种方案。

在系统初始化的时候，我们会定义软中断的处理函数。例如，`NET_TX_SOFTIRQ` 的处理函数是 `net_tx_action`，用于发送网络包。还有一个 `NET_RX_SOFTIRQ` 的处理函数是 `net_rx_action`，用于接收网络包。接收网络包的过程咱们下一节解析。

```

1 open_softirq(NET_TX_SOFTIRQ, net_tx_action);
2 open_softirq(NET_RX_SOFTIRQ, net_rx_action);
3

```

这里我们来解析一下 `net_tx_action`。

```

1 static __latent_entropy void net_tx_action(struct softirq_action *h)
2 {
3     struct softnet_data *sd = this_cpu_ptr(&softnet_data);
4     .....
5     if (sd->output_queue) {
6         struct Qdisc *head;
7
8         local_irq_disable();

```

```

9      head = sd->output_queue;
10     sd->output_queue = NULL;
11     sd->output_queue_tailp = &sd->output_queue;
12     local_irq_enable();
13
14     while (head) {
15         struct Qdisc *q = head;
16         spinlock_t *root_lock;
17
18         head = head->next_sched;
19         .....
20         qdisc_run(q);
21     }
22 }
23 }

```

我们会发现，`net_tx_action` 还是调用了 `qdisc_run`，还是会调用 `__qdisc_run`，然后调用 `qdisc_restart` 发送网络包。

我们来看一下 `qdisc_restart` 的实现。

 复制代码

```

1 static inline int qdisc_restart(struct Qdisc *q, int *packets)
2 {
3     struct netdev_queue *txq;
4     struct net_device *dev;
5     spinlock_t *root_lock;
6     struct sk_buff *skb;
7     bool validate;
8
9     /* Dequeue packet */
10    skb = dequeue_skb(q, &validate, packets);
11    if (unlikely(!skb))
12        return 0;
13
14    root_lock = qdisc_lock(q);
15    dev = qdisc_dev(q);
16    txq = skb_get_tx_queue(dev, skb);
17
18    return sch_direct_xmit(skb, q, dev, txq, root_lock, validate);
19 }

```

`qdisc_restart` 将网络包从 `Qdisc` 的队列中拿下来，然后调用 `sch_direct_xmit` 进行发送。

```

1 int sch_direct_xmit(struct sk_buff *skb, struct Qdisc *q,
2                     struct net_device *dev, struct netdev_queue *txq,
3                     spinlock_t *root_lock, bool validate)
4 {
5     int ret = NETDEV_TX_BUSY;
6
7     if (likely(skb)) {
8         if (!netif_xmit_frozen_or_stopped(txq))
9             skb = dev_hard_start_xmit(skb, dev, txq, &ret);
10    }
11    .....
12    if (dev_xmit_complete(ret)) {
13        /* Driver sent out skb successfully or skb was consumed */
14        ret = qdisc_qlen(q);
15    } else {
16        /* Driver returned NETDEV_TX_BUSY - requeue skb */
17        ret = dev_requeue_skb(skb, q);
18    }
19    .....
20 }

```

在 `sch_direct_xmit` 中，调用 `dev_hard_start_xmit` 进行发送，如果发送不成功，会返回 `NETDEV_TX_BUSY`。这说明网络卡很忙，于是就调用 `dev_requeue_skb`，重新放入队列。


```

1 struct sk_buff *dev_hard_start_xmit(struct sk_buff *first, struct net_device *dev, struct
2 {
3     struct sk_buff *skb = first;
4     int rc = NETDEV_TX_OK;
5
6     while (skb) {
7         struct sk_buff *next = skb->next;
8         rc = xmit_one(skb, dev, txq, next != NULL);
9         skb = next;
10        if (netif_xmit_stopped(txq) && skb) {
11            rc = NETDEV_TX_BUSY;
12            break;
13        }
14    }
15    .....
16 }

```



在 dev_hard_start_xmit 中，是一个 while 循环。每次在队列中取出一个 sk_buff，调用 xmit_one 发送。

接下来的调用链为：xmit_one->netdev_start_xmit->__netdev_start_xmit。

 复制代码


```
1 static inline netdev_tx_t __netdev_start_xmit(const struct net_device_ops *ops, struct :
2 {
3     skb->xmit_more = more ? 1 : 0;
4     return ops->ndo_start_xmit(skb, dev);
5 }
```

这个时候，已经到了设备驱动层了。我们能看到，
drivers/net/ethernet/intel/ixgb/ixgb_main.c 里面有对于这个网卡的操作的定义。

 复制代码

```
1 static const struct net_device_ops ixgb_netdev_ops = {
2     .ndo_open          = ixgb_open,
3     .ndo_stop          = ixgb_close,
4     .ndo_start_xmit    = ixgb_xmit_frame,
5     .ndo_set_rx_mode   = ixgb_set_multi,
6     .ndo_validate_addr = eth_validate_addr,
7     .ndo_set_mac_address = ixgb_set_mac,
8     .ndo_change_mtu    = ixgb_change_mtu,
9     .ndo_tx_timeout    = ixgb_tx_timeout,
10    .ndo_vlan_rx_add_vid = ixgb_vlan_rx_add_vid,
11    .ndo_vlan_rx_kill_vid = ixgb_vlan_rx_kill_vid,
12    .ndo_fix_features    = ixgb_fix_features,
13    .ndo_set_features    = ixgb_set_features,
14 };
```

在这里面，我们可以找到对于 ndo_start_xmit 的定义，调用 ixgb_xmit_frame。

 复制代码

```
1 static netdev_tx_t
2 ixgb_xmit_frame(struct sk_buff *skb, struct net_device *netdev)
3 {
4     struct ixgb_adapter *adapter = netdev_priv(netdev);
5     .....
```

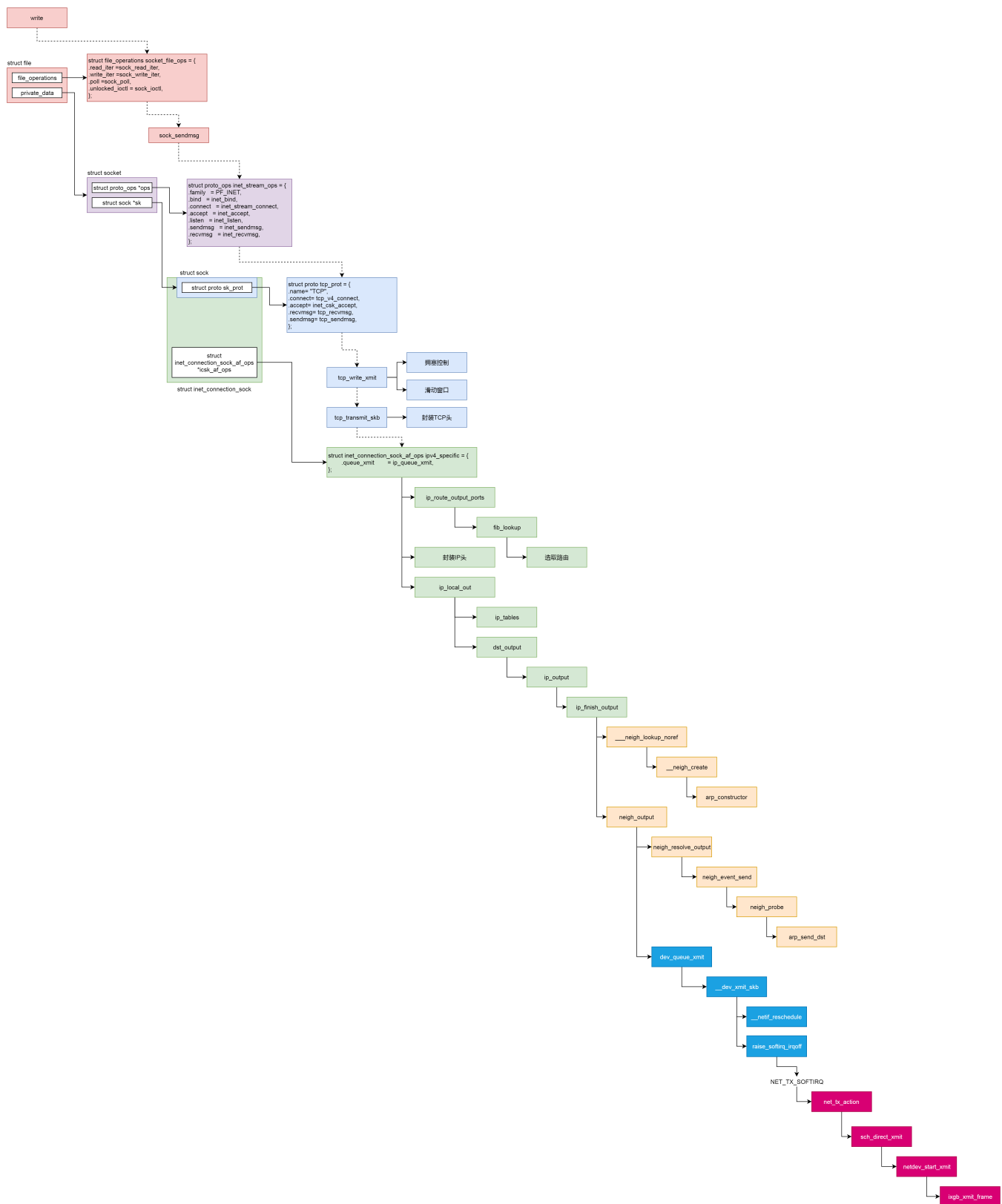
```
6     if (count) {
7         ixgb_tx_queue(adapter, count, vlan_id, tx_flags);
8         /* Make sure there is space in the ring for the next send. */
9         ixgb_maybe_stop_tx(netdev, &adapter->tx_ring, DESC_NEEDED);
10
11     }
12     .....
13     return NETDEV_TX_OK;
14 }
```

在 `ixgb_xmit_frame` 中，我们会得到这个网卡对应的适配器，然后将其放入硬件网卡的队列中。

至此，整个发送才算结束。

总结时刻

这一节，我们继续解析了发送一个网络包的过程，我们整个过程的图画在了下面。



这个过程分成几个层次。

VFS 层：write 系统调用找到 struct file，根据里面的 file_operations 的定义，调用 sock_write_iter 函数。sock_write_iter 函数调用 sock_sendmsg 函数。

Socket 层：从 struct file 里面的 private_data 得到 struct socket，根据里面 ops 的定义，调用 inet_sendmsg 函数。

Socket 层：从 struct socket 里面的 sk 得到 struct sock，根据里面 sk_prot 的定义，调用 tcp_sendmsg 函数。

TCP 层：tcp_sendmsg 函数会调用 tcp_write_xmit 函数，tcp_write_xmit 函数会调用 tcp_transmit_skb，在这里实现了 TCP 层面向连接的逻辑。

IP 层：扩展 struct sock，得到 struct inet_connection_sock，根据里面 icsk_af_ops 的定义，调用 ip_queue_xmit 函数。

IP 层：ip_route_output_ports 函数里面会调用 fib_lookup 查找路由表。FIB 全称是 Forwarding Information Base，转发信息表，也就是路由表。

在 IP 层里面要做的另一个事情是填写 IP 层的头。

在 IP 层还要做的一件事情就是通过 iptables 规则。

MAC 层：IP 层调用 ip_finish_output 进行 MAC 层。

MAC 层需要 ARP 获得 MAC 地址，因而要调用 __neigh_lookup_noref 查找属于同一个网段的邻居，他会调用 neigh_probe 发送 ARP。

有了 MAC 地址，就可以调用 dev_queue_xmit 发送二层网络包了，它会调用 __dev_xmit_skb 会将请求放入队列。

设备层：网络包的发送回触发一个软中断 NET_TX_SOFTIRQ 来处理队列中的数据。这个软中断的处理函数是 net_tx_action。

在软中断处理函数中，会将网络包从队列上拿下来，调用网络设备的传输函数 ixgb_xmit_frame，将网络包发到设备的队列上去。

课堂练习

上一节你应该通过 tcpdump 看到了 TCP 包头的格式，这一节，请你查看一下 IP 包的格式以及 ARP 的过程。

欢迎留言和我分享你的疑惑和见解，也欢迎可以收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院

云计算技术部首席架构师



新版升级：点击「👤 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 45 | 发送网络包（上）：如何表达我们想让合作伙伴做什么？

下一篇 47 | 接收网络包（上）：如何搞明白合作伙伴让我们做什么？

精选留言 (5)

写留言



安排

2019-07-14

老师。应用层调用socket 接口发送数据是到哪个阶段就返回了？是数据写到qdisc中应用就可以返回了吗？还是要等到写到硬件网卡中？

展开



Leon

2019-07-14

最近用go实现了rtp的协议，协议头填充和字节大小计算等等很类似，这节内容有种似曾相识的感觉，借鉴下可以实现的更牛逼，哈

展开 ▾



Linuxer

2019-07-13

设备层：网络包的发送回(这里应该是会吧？)触发一个软中断 NET_TX_SOFTIRQ 来处理队列中的数据。这个软中断的处理函数是 net_tx_action。

在软中断处理函数中，会将网络包从队列上拿下来，调用网络设备的传输函数 ixgb_xmit_frame，将网络包发的(这里应该是到吧？)设备的队列上去

展开 ▾



安排

2019-07-12

发送数据包时，源Mac地址是由协议栈软件加上的吗，还是等数据包到网卡后由网卡硬件自动加上的？

源Mac地址现在一般是写死在网卡里的吗？还是维护在软件协议栈里的一个变量？

展开 ▾



安排

2019-07-12

例如 192.168.2.0/24 和 192.168.0.0/16 都能匹配 192.168.2.100/24。

192.168.0.0/16为什么能匹配192.168.2.100/24 呢？其实对于目的IP我们是不知道子网掩码的，所以192.168.2.100/24这里的24感觉有点迷惑，如果确定它的掩码是24位，那和16位掩码的那个规则就不匹配了吧。

展开 ▾

