

## 第19章 选路请求和选路消息

### 19.1 引言

内核的各种协议并不直接使用前一章提供的函数来访问选路树，而是调用本章提供的几个函数：`rtalloc`和`rtalloc1`是完成路由表查询的两个函数；`rtrequest`函数用于添加和删除路由表项；另外大多数接口在接口连接或断开时都会调用函数 `rtinit`。

选路消息在两个方向上传递信息。进程(如`route`命令)或守护进程(`routed`或`gated`)把选路消息写入选路插口，以使内核添加路由、删除路由或修改现有的路由。当有事件发生时，如接口断开、收到重定向等，内核也会发送选路消息。进程通过选路插口来读取它们感兴趣的内容。在本章中，我们将讨论这些选路消息的格式及其含义，关于选路插口的讨论将在下一章进行。

内核还提供了另一种访问路由表的接口，即系统的 `sysctl`调用，我们将在本章的结尾部分阐述。该系统调用允许进程读取整个路由表或所有已配置的接口及接口地址。

### 19.2 `rtalloc`和`rtalloc1`函数

通常，路由表的查找是通过调用 `rtalloc`和`rtalloc1`函数来实现的。图 19-1给出了 `rtalloc`。

```

58 void
59 rtalloc(ro)
60 struct route *ro;
61 {
62     if (ro->ro_rt && ro->ro_rt->rt_ifp && (ro->ro_rt->rt_flags & RTF_UP))
63         return; /* XXX */
64     ro->ro_rt = rtalloc1(&ro->ro_dst, 1);
65 }

```

route.c

图19-1 `rtalloc` 函数

58-65 参数`ro`是一个指针，它指向TCP或UDP所使用的Internet PCB(第22章)中的`route`结构。如果`ro`已经指向了某个`rtentry`结构(即`ro_rt`非空)，而该结构指向一个接口结构且路由有效，则函数立即返回。否则，`rtalloc1`将被调用，调用的第二个参数为1。我们很快会看到该参数的用途。

如图19-2所示，`rtalloc1`调用了`rnmatch`函数，对于Internet地址来说，该函数就是`rnmatch`数(图18-17)。

66-76 第一个参数是一个指针，它指向一个含有待查找地址的插口地址结构。`sa_family`成员用于选择所查找的路由表。

1. 调用`rnmatch`

77-78 如果符合下列三个条件，则查找成功。

- 1) 存在该协议族的路由表；
- 2) `rnmatch`返回一个非空指针；并且

3) 匹配的radix\_node结构没有设置RNF\_ROOT标志。

注意，树中标有end的两个叶子都设有RNF\_ROOT标志。

route.c

```

66 struct rtentry *
67 rtalloc1(dst, report)
68 struct sockaddr *dst;
69 int report;
70 {
71     struct radix_node_head *rn timer = rt_tables[dst->sa_family];
72     struct rtentry *rt;
73     struct radix_node *rn;
74     struct rtentry *newrt = 0;
75     struct rt_addrinfo info;
76     int s = splnet(), err = 0, msgtype = RTM_MISS;

77     if (rn timer && (rn = rn timer->rn timer_matchaddr((caddr_t) dst, rn timer)) &&
78         ((rn->rn_flags & RNF_ROOT) == 0)) {
79         newrt = rt = (struct rtentry *) rn;
80         if (report && (rt->rt_flags & RTF_CLONING)) {
81             err = rtrequest(RTM_RESOLVE, dst, SA(0),
82                           SA(0), 0, &newrt);
83             if (err) {
84                 newrt = rt;
85                 rt->rt_refcnt++;
86                 goto miss;
87             }
88             if ((rt = newrt) && (rt->rt_flags & RTF_XRESOLVE)) {
89                 msgtype = RTM_RESOLVE;
90                 goto miss;
91             }
92         } else
93             rt->rt_refcnt++;
94     } else {
95         rtstat.rts_unreach++;
96         miss: if (report) {
97             bzero((caddr_t) &info, sizeof(info));
98             info.rti_info[RTAX_DST] = dst;
99             rt_missmsg(msgtype, &info, 0, err);
100         }
101     }
102     splx(s);
103     return (newrt);
104 }

```

route.c

图19-2 rtalloc1 函数

## 2. 查找失败

94-101 在这三个条件中只要有一个条件没有得到满足，查找就会失败，并且统计值 rts\_unreach也要递增。这时，如果调用 rtalloc1 的第二个参数(report)为1，就会产生一个选路消息。任何感兴趣的进程都可以通过选路插口读取该消息。选路消息的类型为 RTM\_MISS，并且函数返回一个空指针。

79 如果三个条件都满足，则查找成功。指向匹配的 radix\_node结构的指针保存在 rt和 newrt中。注意，在 rtentry结构的定义中(图18-24)，两个 radix\_node结构在开头的位置处，如图18-8所示，其中第一个代表一个叶结点。因此， rn\_match返回的 radix\_node结构的指针事实上是一个指向 rtentry结构的指针，该 rtentry结构是一个匹配的叶结点。

### 3. 创建克隆表项

80-82 如果调用的第二个参数非零，而且匹配的路由表项设有 `RTF_CLONING` 标志，则调用 `rtrequest` 函数发送 `RTM_RESOLVE` 命令来创建一个新的 `rtentry` 结构，该结构是查询结果的克隆。ARP 将针对多播地址利用这一机制。

### 4. 克隆失败

83-87 如果 `rtrequest` 返回一个差错，`newrt` 就被重新设置成 `rn_match` 所返回的表项，并增加它的引用计数。然后程序跳转到 `miss` 处，产生一条 `RTM_MISS` 消息。

### 5. 检查是否需要外部转换

88-91 如果 `rtrequest` 成功，并且新克隆的表项设有 `RTF_XRESOLVE` 标志，则程序跳至 `miss` 处，但这次产生的是 `RTM_RESOLVE` 消息。该消息的目的是为了把路由创建的时间通知给用户进程，在 IP 地址到 X.121 地址的转换过程中会用到它。

### 6. 为正常的成功查找递增引用计数

92-93 当查找成功但没有设置 `RTF_CLONING` 标志时，该语句将递增路由表项的引用计数。这是本函数正常情况下的处理流程，之后程序返回一个非空的指针。

虽然是这样小的一段程序，但是在 `rtalloc1` 的处理过程中有很多选择。该函数有 7 个不同的流程，如图 19-3 所示。

	report 参数	RTF_ CLONING 标志	RTM_ RESOLVE 返回	RTF_ XRESOLVE 标志	产生的 路由消息	rt_refcnt	返回值
查找失败	0						空
	1				RTM_MISS		空
查找成功		0				++	ptr
	0					++	ptr
	1	1	OK	0		++	ptr
	1	1	OK	1	RTM_RESOLVE	++	ptr
	1	1	差错		RTM_MISS	++	ptr

图19-3 `rtalloc1` 处理过程小结

需要解释的是，如果存在默认路由，前两行（找不到路由表项的流程）是不可能出现的。还有，在第5、第6两行中的 `rt_refcnt` 也做了递增，因为这两行在调用 `rtrequest` 时使用了 `RTM_RESOLVE` 参数，递增在 `rtrequest` 中完成。

## 19.3 宏 `RTFREE` 和 `rtfree` 函数

宏 `RTFREE`，如图 19-4 所示，仅在引用计数小于等于 1 时才调用 `rtfree` 函数；否则，它仅完成引用计数的递减。

209-213 `rtfree` 函数如图 19-5 所示。当不存在对 `rtentry` 结构的引用时，函数就释放该结构。例如，在图 22-7 中，当释放一个协议控制块时，如果它指向一个路由表项，则需要调用 `rtfree`。

105-115 首先递减路由表项的引用计数，如果它小于等于 0 并且该路由不可用，则该表项可以被释放。如果该表项设有 `RNF_ACTIVE` 或 `RNF_ROOT` 标志，那么这是一个内部差错。因为，如果设有 `RNF_ACTIVE`，那么该结构仍是路由表的一部分；如果设有 `RNF_ROOT`，那么它是一个由 `rn_inithead` 创建的标有 `end` 的结构。

```

209 #define RTFREE(rt) \
210     if ((rt)->rt_refcnt <= 1) \
211         rtfree(rt); \
212     else \
213         (rt)->rt_refcnt--; /* no need for function call */

```

route.h

图19-4 宏RTFREE

```

105 void
106 rtfree(rt)
107 struct rtable *rt;
108 {
109     struct ifaddr *ifa;

110     if (rt == 0)
111         panic("rtfree");
112     rt->rt_refcnt--;
113     if (rt->rt_refcnt <= 0 && (rt->rt_flags & RTF_UP) == 0) {
114         if (rt->rt_nodes->rn_flags & (RNF_ACTIVE | RNF_ROOT))
115             panic("rtfree 2");
116         rttrash--;
117         if (rt->rt_refcnt < 0) {
118             printf("rtfree: %x not freed (neg refs)\n", rt);
119             return;
120         }
121         ifa = rt->rt_ifa;
122         IFAFREE(ifa);
123         Free(rt_key(rt));
124         Free(rt);
125     }
126 }

```

route.c

图19-5 rtfree 函数：释放一个rtable 结构

116 rttrash是一个用于调试的计数器，记录那些不在选路树中但仍未释放的路由表项的数目。当rtrequest开始删除路由时，它被递增，然后在这儿递减。正常情况下，它的值应该是0。

#### 1. 释放接口引用

117-122 先查看引用计数。确认引用计数非负后，IFAFREE将递减ifaddr结构的引用计数。当计数值递减为零时，调用ifafree释放它。

#### 2. 释放选路存储器

123-124 释放由路由表项关键字及其网关所占的存储器。我们会看到 rt\_setgate把它们分配在存储器的同一个连着的块中。因此，只调用一个 Free就可以同时把它们释放。最后还要释放rtable结构。

### 路由表引用计数

路由表引用计数(rt\_refcnt)的处理与其他许多引用计数的处理不同。我们看到，在图18-2中，大多数路由的引用计数为0，而这些没有引用的路由表项并没有被删除。原因就在rtfree中：只有当RTF\_UP标志被删除时，引用计数为0的表项才会被删除。而仅当从选路树中删除路由时，该标志才会被rtrequest删除。

大多数路由是按如下方式使用的。

- 如果到某接口的路由是在配置该接口时自动创建的（典型的，例如以太网接口的配置），则rtinit用命令参数RTM\_ADD来调用rtrequest，以创建新的路由表项，并设置它的引用计数为1。然后，rtinit在退出前把该引用计数递减成0。

对于点到点接口的处理过程也是类似的，所以路由的引用计数也是从0开始。

如果路由是由route命令手工创建的，或者是由选路守护进程创建的，处理过程同样是类似的。route\_output用命令参数RTM\_ADD来调用rtrequest，并设置新路由的引用计数为1。在退出前，route\_output把该引用计数递减到0。

因此，所有新创建的路由都是从引用计数0开始的。

- 当TCP或UDP在插口上发送IP数据包时，ip\_output调用rtalloc，rtalloc再调用rtallocl。如图19-3所示，如果找到了路由，rtallocl就会递增其引用计数。所查找到的路由称为被持路由(held route)，因为协议持有指向路由表项的指针，该指针通常被包含在协议控制块中的route结构里。一个被其他协议持有的rtentry结构是不能被删除的。所以，在rtfree中，当引用计数为0时，rtentry结构才能被删除。
- 协议通过调用RTFREE或rtfree来释放被持路由。在图8-24中，当ip\_output检测到目的地址改变时，我们已经使用了这种处理。在第22章中，释放持有路由的协议控制块时，我们还会遇到这种处理。

在后面的代码中可能会引起混淆的是，rtallocl经常被调用以判断路由是否存在，而调用者并非试图持有该路由。因为rtallocl递增了该路由的引用计数器，所以调用者就立即递减该计数器。

考虑一个被rtrequest删除的路由。它的RTF\_UP标志被清除，并且，如果没有被持有（它的引用计数为0），就要调用rtfree。但rtfree认为引用计数小于0是错的，所以rtrequest查看它的引用计数，如果小于等于0，就递增该计数值并调用rtfree。通常，这将使引用计数变成1，之后，rtfree把引用计数递减到0，并删除该路由。

## 19.4 rtrequest函数

rtrequest函数是添加和删除路由表项的关键点。图19-6给出了调用它的一些其他函数。

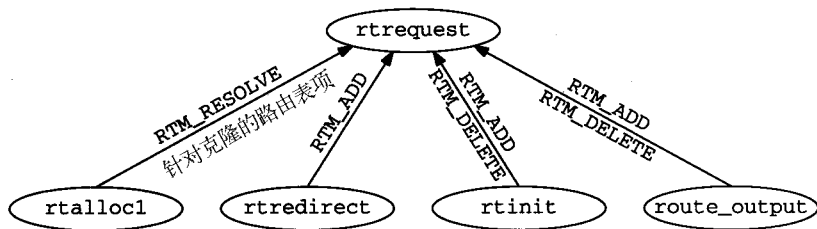


图19-6 调用rtrequest 的函数

rtrequest是一个switch语句，每个case对应一个命令：RTM\_ADD、RTM\_DELETE和RTM\_RESOLVE。图19-7给出了该函数的开头一段以及RTM\_DELETE命令的处理。

290-307 第二个参数，dst，是一个插口地址结构，它指定在路由表中添加或删除的表项。

表项中的 `sa_family` 用于选择路由表。如果 `flags` 参数指出该路由是主机路由（而不是到某个网络的路由），则设置 `netmask` 指针为空，忽略调用者设置的任何值。

```

290 int
291 rtrequest(req, dst, gateway, netmask, flags, ret_nrt)
292 int req, flags;
293 struct sockaddr *dst, *gateway, *netmask;
294 struct rtable **ret_nrt;
295 {
296     int s = splnet();
297     int error = 0;
298     struct rtable *rt;
299     struct radix_node *rn;
300     struct radix_node_head *rnhead;
301     struct ifaddr *ifa;
302     struct sockaddr *ndst;
303 #define senderr(x) { error = x; goto bad; }

304     if ((rnhead = rt_tables[dst->sa_family]) == 0)
305         senderr(ESRCH);
306     if (flags & RTF_HOST)
307         netmask = 0;
308     switch (req) {
309     case RTM_DELETE:
310         if ((rn = rnhead->rnhead_deladdr(dst, netmask, rnhead)) == 0)
311             senderr(ESRCH);
312         if (rn->rn_flags & (RNF_ACTIVE | RNF_ROOT))
313             panic("rtrequest delete");
314         rt = (struct rtable *) rn;
315         rt->rt_flags &= ~RTF_UP;
316         if (rt->rt_gwroute) {
317             rt = rt->rt_gwroute;
318             RTFREE(rt);
319             (rt = (struct rtable *) rn)->rt_gwroute = 0;
320         }
321         if ((ifa = rt->rt_ifa) && ifa->ifa_rtrequest)
322             ifa->ifa_rtrequest(RTM_DELETE, rt, SA(0));
323         rttrash++;
324         if (ret_nrt)
325             *ret_nrt = rt;
326         else if (rt->rt_refcnt <= 0) {
327             rt->rt_refcnt++;
328             rtfree(rt);
329         }
330         break;

```

图19-7 rtrequest 函数：RTM\_DELETE 命令

### 1. 从选路树中删除路由

309-315 `rnhead_deladdr` 函数(图18-17中的 `rn_delete`)从选路树中删除表项，返回相应 `rtable` 结构的指针，并清除 `RTF_UP` 标志。

### 2. 删除对网关路由表项的引用

316-320 如果该表项是一个经过某网关的非直接路由，则 `RTFREE` 递减该网关路由表项的引用计数。如它的引用计数被减为 0，则删除它。设置 `rt_gwroute` 指针为空，并将 `rt` 设置成原来要删除的表项。

### 3. 调用接口请求函数

321-322 如果该表项定义了 `ifa_rtrequest` 函数，就调用该函数。ARP 会使用该函数，例如，在第21章中用它来删除对应的ARP表项。

### 4. 返回指针或删除引用

323-330 因为该表项在接着的代码里不一定被删除，所以递增全局变量 `rttrash`。如果调用者需要选路树中被删除的 `rtentry` 结构的指针（即如果 `ret_nrt` 非空），则返回该指针，但此时不能释放该表项：调用者必须在使用完该表项后调用 `rtfree` 来删除它。如果 `ret_nrt` 为空，则该表项被释放：如果它的引用计数小于等于 0，则递增该计数值，并调用 `rtfree`。`break` 语句将使函数退出。

图19-8给出了函数的下一部分，用于处理 `RTM_RESOLVE` 命令。只有 `rtalloc1` 能够携带此命令参数调用本函数。也只有在从一个设有 `RTF_CLONING` 标志的表项中克隆一个新的表项时，`rtalloc1` 才这样用。

```
331     case RTM_RESOLVE:                                     route.c
332         if (ret_nrt == 0 || (rt = *ret_nrt) == 0)
333             senderr(EINVAL);
334         ifa = rt->rt_ifa;
335         flags = rt->rt_flags & ~RTF_CLONING;
336         gateway = rt->rt_gateway;
337         if ((netmask = rt->rt_genmask) == 0)
338             flags |= RTF_HOST;
339         goto makeroute;                                     route.c
```

图19-8 `rtrequest` 函数：`RTM_RESOLVE` 命令

331-339 最后一个参数，`ret_nrt`，在这个命令里的用途不同：它是一个设有 `RTF_CLONING` 标志的路由表项的指针（图19-2）。新的表项具有相同的 `rt_ifa` 指针、相同的 `rt_gateway` 和相同的标志（`RTF_CLONING` 标志被清除）。如果被克隆表项的 `rt_genmask` 指针为空，则新表项是一个主机路由，因此要设置它的 `RTF_HOST` 标志；否则新表项为网络路由，其网络掩码通过复制 `rt_genmask` 得到。在本节的结尾部分，我们给出了克隆带网络掩码的路由的一个例子。这个 `case` 将跳转至下个图中的 `makeroute` 标记处继续进行。

图19-9给出了 `RTM_ADD` 命令的代码。

### 5. 定位相应的接口

340-342 函数 `ifa_ifwithroute` 为目的 (`dst`) 查找适当的本地接口，并返回指向该接口的 `ifaddr` 结构的指针。

### 6. 为路由表项分配存储器

343-348 分配了一个 `rtentry` 结构。在前一章中我们知道，该结构包含了两个选路树的 `radix_node` 结构及其他路由信息。该结构被清零，之后，其标志 `rt_flags` 被设置成调用本函数的 `flags` 参数，同时再设置 `RTF_UP` 标志。

### 7. 分配并复制网关地址

349-352 `rt_gateway` 函数（图19-11）为路由表 (`dst`) 及其 `gateway` 分配了存储器，然后将 `gateway` 复制到新分配的存储器中，并设置指针 `rt_key`、`rt_gateway` 和 `rt_gwroute`。

### 8. 复制目的地址

route.c

```

340     case RTM_ADD:
341         if ((ifa = ifa_ifwithroute(flags, dst, gateway)) == 0)
342             senderr(ENETUNREACH);
343
344     makeroute:
345         R_Malloc(rt, struct rtentry *, sizeof(*rt));
346         if (rt == 0)
347             senderr(ENOBUFS);
348         Bzero(rt, sizeof(*rt));
349         rt->rt_flags = RTF_UP | flags;
350         if (rt_setgate(rt, dst, gateway)) {
351             Free(rt);
352             senderr(ENOBUFS);
353         }
354         ndst = rt_key(rt);
355         if (netmask) {
356             rt_maskedcopy(dst, ndst, netmask);
357         } else
358             Bcopy(dst, ndst, dst->sa_len);
359
360         rn = rnh->rnh_addaddr((caddr_t) ndst, (caddr_t) netmask,
361                             rnh, rt->rt_nodes);
362         if (rn == 0) {
363             if (rt->rt_gwroute)
364                 rtfree(rt->rt_gwroute);
365             Free(rt_key(rt));
366             Free(rt);
367             senderr(EEXIST);
368         }
369         ifa->ifa_refcnt++;
370         rt->rt_ifa = ifa;
371         rt->rt_ifp = ifa->ifa_ifp;
372         if (req == RTM_RESOLVE)
373             rt->rt_rmx = (*ret_nrt)->rt_rmx; /* copy metrics */
374         if (ifa->ifa_rtrequest)
375             ifa->ifa_rtrequest(req, rt, SA(ret_nrt ? *ret_nrt : 0));
376         if (ret_nrt) {
377             *ret_nrt = rt;
378             rt->rt_refcnt++;
379         }
380         break;
381     }
382     bad:
383     splx(s);
384     return (error);
385 }

```

route.c

图19-9 rtrequest 函数：RTM\_ADD 命令

353-357 把目的地址(路由表表项dst)复制到rn\_key所指向的存储器中。如果提供了网络掩码,则rt\_maskedcopy对dst和netmask进行逻辑与运算,得到新的表项。否则, dst就会被复制成新的表项。对dst和netmask进行逻辑与运算是为了确保表中的表项已经和它的掩码进行了与运算。这样,查找表项与表中的表项进行比较时,只需要另外对查找表项和掩码进行逻辑与运算就可以了。例如,下面的这个命令向以太网接口 le0添加了另一个 IP地址(一个别名),其子网为12而不是13。

```
bsdi $ ifconfig le0 inet 140.252.12.63 netmask 0xffffffe0 alias
```

该例子中存在的一个问题是，我们所指定的全 1 的主机号是错误的。不过，该表项存入路由表后，我们用 `netstat` 验证可知该地址已经和掩码进行过逻辑与运算了。

Destination	Gateway	Flags	Refs	Use	Interface
140.252.12.32	link#1	U C	0	0	le0

#### 9. 往选路树中添加表项

358-366 `rn_h_addaddr` 函数(图18-17中的 `rn_addroute`)向选路树中添加这个 `rtentry` 结构，其中附带了它的目的地址和掩码。如果有差错产生，则释放该结构，并返回 `EEXIST`(即，该表项已经存在于路由表中了)。

#### 10. 保存接口指针

367-369 递增 `ifaddr` 结构的引用计数，并保存 `ifaddr` 和 `ifnet` 结构的指针。

#### 11. 为新克隆的路由复制度量

370-371 如果命令是 `RTM_RESOLVE`(不是 `RTM_ADD`)，则把被克隆的表项中的整个度量结构复制到新的表项里。如果命令是 `RTM_ADD`，则调用者可在函数返回后设置该度量值。

#### 12. 调用接口请求函数

372-373 如果为该表项定义了 `ifa_rtrequest` 函数，则调用该函数。对于 `RTM_ADD` 和 `RTM_RESOLVE` 命令，ARP都要用该函数来完成一些额外的处理。

#### 13. 返回指针并递增引用计数

374-378 如果调用者需要该新结构的指针，则通过 `ret_nrt` 返回该指针，并将引用计数值从0递增到1。

例：克隆的带网络掩码的路由

仅当 `rtrequest` 的 `RTM_RESOLVE` 命令创建克隆路由时，才使用 `rt_genmask` 的值。如果 `rt_genmask` 指针非空，则它指向的插口地址结构就成了新创建路由的网络掩码。在我们的路由表中，即图 18-2，克隆的路由是针对本地以太网和多播地址的。下面的例子引自 [Sklower 1991]，它提供了克隆路由的不同用法。另外一个例子见习题 19.2。

考虑一个B类网络，如 128.1，它在点到点链路之外。子网掩码是 `0xfffff000`，其中含8比特的子网号和8比特的主机号。我们要为所有可能的 254个子网提供路由表项，这些表项的网关是与本机直接相连的路由器，该路由器知道如何到达与 128.1网络相连的链路。

假设该网关不是我们的默认路由器，则最简单的方法就是创建单个表项，该表项以 128.1.0.0为目的、以 `0xfffff0000` 为掩码。可是，假设 128.1网络的拓扑使所有可能的 254个子网中的每一个都有不同的运营特性：RTTs、MTUs和时延等。那么如果每个子网都有单独的路由表项，我们就能够看到，无论何时连接断开后，TCP都会刷新该路由表项的统计值，如路由的RTT、RTT变量等(图27-3)。尽管我们可以用 `route` 命令手工地为 254个子网中的每一个子网都添加路由表项，但更好的方法是采用克隆机制。

由系统管理员先创建一个以 128.1.0.0为目的，以 `0xfffff0000` 为网络掩码的表项。再设置其 `RTF_CLONING` 标志，并设置 `genmask` 为 `0xfffff000`(与网络掩码不同)。这时，如果在路由表中查找 128.1.2.3，而路由表中没有子网 128.1.2的表项，那么具有掩码 `0xfffff0000` 的网络 128.1的表项为最佳匹配。因为该表项设有 `RTF_CLONING` 标志，所以要创建一个新的表项，新表项以 128.1.2为目的，以 `0xfffff000`(`genmask` 的值)为网络掩码。

这样，下一次引用该子网内的主机时，如 128.1.2.88，最佳匹配就是新创建的表项。

### 19.5 rt\_setgate函数

选路树中的每个叶子都有一个表项（rt\_key，也就是在 rtenry 结构开头的 radix\_node 结构的 rn\_key 成员）和一个相关联的网关（rt\_gateway）。在创建路由表项时，它们都被指定为插口地址结构。rt\_setgate 为这两个结构分配存储器，如图 19-10 所示。

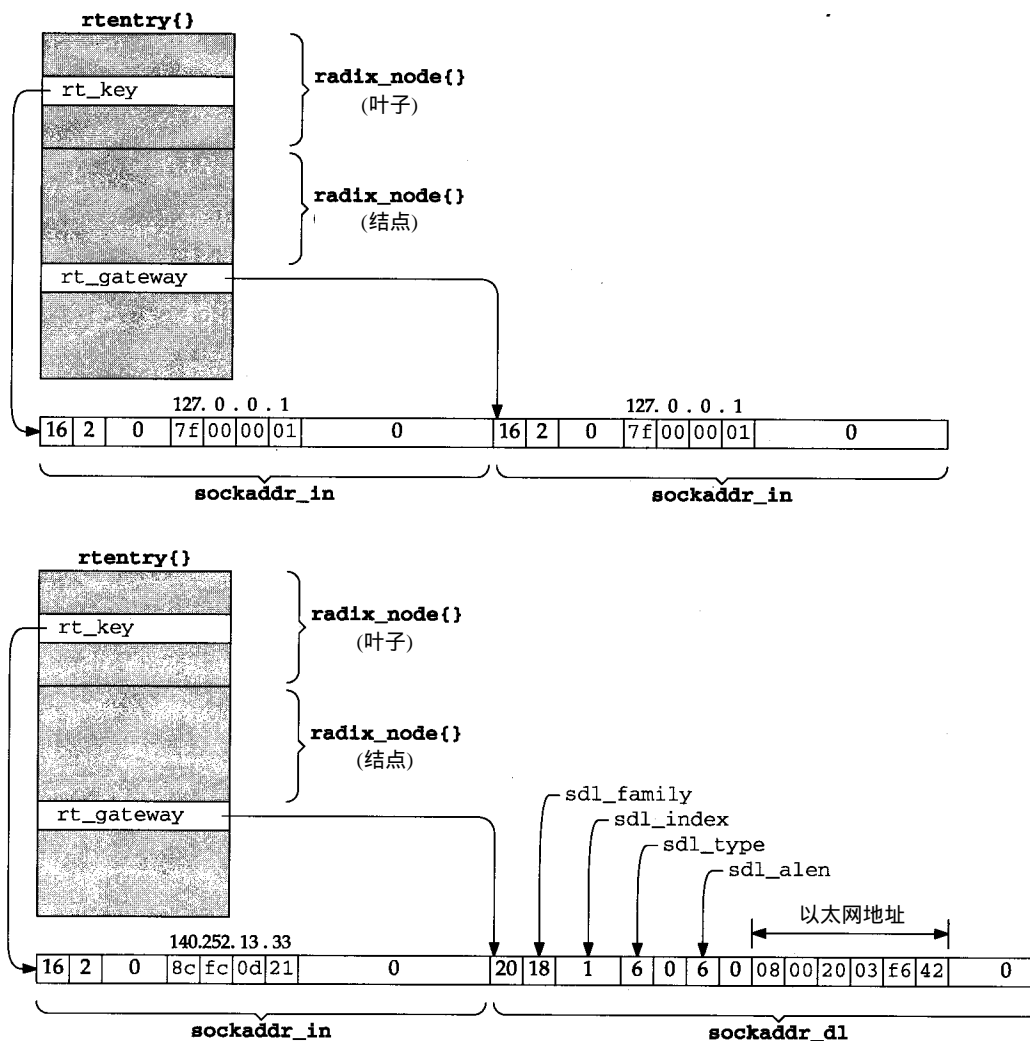


图19-10 路由表表项和相关网关示例

这个例子给出了图 18-2 中的两个表项，它们的表项分别是 127.0.0.1 和 140.252.13.33。前一个的网关成员指向一个 Internet 插口地址结构，后一个的网关成员指向一个含以太网地址的数据链路插口地址。前一个是在系统初始化时，由 route 系统将其添加到路由表中的；后一个是由 ARP 创建的。

在图 19-11 中，我们有意识地把两个由 `rt_key` 指向的结构紧挨着画在一起，因为它们是由 `rt_setgate` 一起分配的。

```
384 int
385 rt_setgate(rt0, dst, gate)
386 struct rtable *rt0;
387 struct sockaddr *dst, *gate;
388 {
389     caddr_t new, old;
390     int     dlen = ROUNDUP(dst->sa_len), glen = ROUNDUP(gate->sa_len);
391     struct rtable *rt = rt0;
392
393     if (rt->rt_gateway == 0 || glen > ROUNDUP(rt->rt_gateway->sa_len)) {
394         old = (caddr_t) rt_key(rt);
395         R_Malloc(new, caddr_t, dlen + glen);
396         if (new == 0)
397             return 1;
398         rt->rt_nodes->rn_key = new;
399     } else {
400         new = rt->rt_nodes->rn_key;
401         old = 0;
402     }
403     Bcopy(gate, (rt->rt_gateway = (struct sockaddr *) (new + dlen)), glen);
404     if (old) {
405         Bcopy(dst, new, dlen);
406         Free(old);
407     }
408     if (rt->rt_gwroute) {
409         rt = rt->rt_gwroute;
410         RTFREE(rt);
411         rt = rt0;
412         rt->rt_gwroute = 0;
413     }
414     if (rt->rt_flags & RTF_GATEWAY) {
415         rt->rt_gwroute = rtalloc1(gate, 1);
416     }
417     return 0;
418 }
```

route.c

图19-11 rt\_setgate 函数

#### 1. 依据插口地址结构设置长度

384-391 dlen是目的插口地址结构的长度，glen是网关插口地址结构的长度。ROUNDUP宏把数值上舍入成4的倍数个字节，但大多数插口地址结构的长度本身就是4的倍数。

#### 2. 分配存储器

392-401 如果还没有给该路由表项和网关分配存储器，或glen大于当前rt\_gateway所指向的结构长度，则分配一片新的存储器，并使rn\_key指向新分配的存储器。

#### 3. 使用分配给表项和网关的存储器

398-401 由于已经给表项和网关分配了一片足够大小的存储器，因此，直接将new指向这个已经存在的存储器。

#### 4. 复制新网关

402 复制新的网关结构，并且设置rt\_gateway，使其指向插口地址结构。

#### 5. 从原有的存储器中将表项复制到新存储器中

403-406 如果分配了新的存储器，则在网关字段被复制前，先复制路由表表项dst，并释放原有的存储器片。

#### 6. 释放网关路由指针

407-412 如果该路由表项含有非空的 `rt_gwroute` 指针, 则用 `RTFREE` 释放该指针所指的结构, 并设置 `rt_gwroute` 为空。

#### 7. 查找并保存新的网关路由指针

413-415 如果路由表项是一个非直接路由, 则 `rtalloc1` 查找新网关的路由表项, 并将它保存在 `rt_gwroute` 中。如果非直接路由指定的网关无效, 则 `rt_setgate` 并不返回任何差错, 但 `rt_gwroute` 会是一个空指针。

## 19.6 rtinit函数

Internet 协议添加或删除相关接口的路由时, 对 `rtinit` 的调用有四个。

- 在设置点到点接口的目的地址时, `in_control` 调用 `rtinit` 两次。第一次调用指定 `RTM_DELETE` 命令, 以删除所有现存的到该目的地址的路由 (图6-21); 第二次调用指定 `RTM_ADD` 命令, 以添加新路由。
- `in_ifinit` 调用 `rtinit` 为广播网络添加一条网络路由或为点到点链路 (图6-19) 添加一条主机路由。如果是给以太网接口添加的路由, 则 `in_ifinit` 自动设置其 `RTF_CLONING` 标志。
- `in_ifscrub` 调用 `rtinit`, 以删除一个接口现存的路由。

图19-12给出了 `rtinit` 函数的第一部分。 `cmd` 参数只能是 `RTM_ADD` 或 `RTM_DELETE`。

route.c

```

441 int
442 rtinit(ifa, cmd, flags)
443 struct ifaddr *ifa;
444 int      cmd, flags;
445 {
446     struct rtable *rt;
447     struct sockaddr *dst;
448     struct sockaddr *deldst;
449     struct mbuf *m = 0;
450     struct rtable *nrt = 0;
451     int      error;

452     dst = flags & RTF_HOST ? ifa->ifa_dstaddr : ifa->ifa_addr;
453     if (cmd == RTM_DELETE) {
454         if ((flags & RTF_HOST) == 0 && ifa->ifa_netmask) {
455             m = m_get(M_WAIT, MT_SONAME);
456             deldst = mtod(m, struct sockaddr *);
457             rt_maskedcopy(dst, deldst, ifa->ifa_netmask);
458             dst = deldst;
459         }
460         if (rt = rtalloc1(dst, 0)) {
461             rt->rt_refcnt--;
462             if (rt->rt_ifa != ifa) {
463                 if (m)
464                     (void) m_free(m);
465                 return (flags & RTF_HOST ? EHOSTUNREACH
466                     : ENETUNREACH);
467             }
468         }
469     }
470     error = rtrequest(cmd, dst, ifa->ifa_addr, ifa->ifa_netmask,
471         flags | ifa->ifa_flags, &nrt);
472     if (m)
473         (void) m_free(m);

```

route.c

图19-12 `rt_init` 函数: 调用 `rtrequest` 处理命令

### 1. 为路由获取目的地址

452 如果是一个到达某主机的路由，则目的地址是点到点链路的另一端。否则，我们处理的就是一个网络路由，其目的地址是接口的单播地址（经ifa\_netmask掩码过的）。

### 2. 用网络掩码给网络地址掩码

453-459 如果要删除路由，则必须在路由表中查找该目的地址，并得到它的路由表项。如果要删除的是一个网络路由且接口拥有相关联的网络掩码，则分配一个 mbuf，用 rt\_maskedcopy 对目的地址和调用参数中的掩码地址进行逻辑与运算，并将结果复制到 mbuf 中。令 dst 指向 mbuf 中掩码过的复制值，它就是下一步要查找的目的地址。

### 3. 查找路由表项

460-469 rtalloc1 在路由表中查找目的地址，如果能找到，则先递减该表项的引用计数（因为 rtalloc1 递增了该引用计数）。如果路由表中该接口的 ifaddr 指针不等于调用者的参数，则返回一个差错。

### 4. 处理请求

470-473 rt\_request 执行 RTM\_ADD 或 RTM\_DELETE 命令。当 rt\_request 返回时，如果之前分配了 mbuf，则释放它。

图19-13给出了 rtinit 的后半部分。

```

474     if (cmd == RTM_DELETE && error == 0 && (rt = nrt)) {
475         rt_newaddrmsg(cmd, ifa, error, nrt);
476         if (rt->rt_refcnt <= 0) {
477             rt->rt_refcnt++;
478             rtfree(rt);
479         }
480     }
481     if (cmd == RTM_ADD && error == 0 && (rt = nrt)) {
482         rt->rt_refcnt--;
483         if (rt->rt_ifa != ifa) {
484             printf("rtinit: wrong ifa (%x) was (%x)\n", ifa,
485                 rt->rt_ifa);
486             if (rt->rt_ifa->ifa_rtrequest)
487                 rt->rt_ifa->ifa_rtrequest(RTM_DELETE, rt, SA(0));
488             IFAFREE(rt->rt_ifa);
489             rt->rt_ifa = ifa;
490             rt->rt_ifp = ifa->ifa_ifp;
491             ifa->ifa_refcnt++;
492             if (ifa->ifa_rtrequest)
493                 ifa->ifa_rtrequest(RTM_ADD, rt, SA(0));
494         }
495         rt_newaddrmsg(cmd, ifa, error, nrt);
496     }
497     return (error);
498 }

```

route.c

route.c

图19-13 rtinit 函数：后半部分

### 5. 删除成功时产生一个选路消息

474-480 如果删除了一个路由，并且 rtrequest 返回 0 和被删除的 rtentry 结构的指针（nrt 中），就用 rt\_newaddrmsg 产生一个选路插口消息。如果引用计数小于等于 0，则递增该引用计数，并用 rtfree 释放该路由。

## 6. 成功添加

481-482 如果添加了一个路由, 并且 `rtrequest` 返回了0和被添加的 `rtentry` 结构的指针 (`nrt`), 就递减其引用计数 (因为 `rtrequest` 递增了该引用计数)。

## 7. 不正确的接口

483-494 如果新路由表项中接口的 `ifaddr` 指针不等于调用参数, 则表明有差错产生。利用 `rtrequest`, 通过调用 `ifa_ifwithroute` 来测定新路由中的 `ifa` 指针 (`rtrequest` 函数如图19-9所示)。产生这个差错后, 做如下步骤: 向控制台输出一条出错消息; 如果定义了 `ifa_rtrequest` 函数, 就以 `RTM_DELETE` 为参数调用它; 释放 `ifaddr` 结构; 设置 `rt_ifa` 指针为调用者指定的值; 递增接口的引用计数; 如果定义了 `ifa_rtrequest` 函数, 就以 `RTM_ADD` 为参数调用它。

## 8. 产生选路消息

495 用 `rt_newaddrmsg` 为 `RTM_ADD` 命令产生一个选路插口消息。

## 19.7 `rtredirect` 函数

当收到一个ICMP重定向后, `icmp_input` 调用 `rtredirect` 及 `pfctlinput` (图11-27)。后一个函数又调用 `udp_ctlinput` 和 `tcp_ctlinput`, 这两个函数遍历所有的UDP和TCP协议控制块(PCB)。如果PCB连接到一个外部地址, 而到该外部地址的方向已经被改变, 并且该PCB持有到那个外部地址的路由, 则调用 `rtfree` 释放该路由。下一次使用这些控制块发送到该外部地址的IP数据报时, 就会调用 `rtalloc`, 并在路由表中查找该目的地址, 很可能会找到一条新(改变过方向的)路由。

`rtredirect` 函数的作用是验证重定向中的信息, 并立即更新路由表, 产生选路插口消息。图19-14给出了 `rtredirect` 函数的前半部分。

147-157 函数的参数包括: `dst`, 导致重定向的数据报的目的IP地址(图8-18中的HD); `gateway`, 路由器的IP地址, 用作该目的的新网关字段(图8-18中的R2); `netmask`, 空指针; `flags`, 设置了 `RTF_GATEWAY` 标志和 `RTF_HOST` 标志; `src`, 发送重定向的路由器的IP地址(图8-18中的R1); `rtp`, 空指针。需要指出的是, `icmp_input` 调用本函数时, 参数 `netmask` 和 `rtp` 是空指针, 但是其他协议调用本函数时, 这两个参数未必为空指针。

### 1. 新路由必须直接相连

158-162 新的网关必须是直接相连的, 否则该重定向无效。

### 2. 查找目的地址的路由表项并验证重定向

163-177 调用 `rtalloc1` 在路由表中查找到目的地址的路由。验证重定向时, 下列条件必须为真, 否则该重定向无效, 并且函数返回一个差错。要注意的一点是, `icmp_input` 会忽略从 `rtredirect` 返回的任何差错。ICMP也会忽略它, 即不会为一个无效的重定向而产生一个差错信息。

- 必须未设置 `RTF_DONE` 标志;
- `rtalloc` 必须已找到一个到 `dst` 的路由表项;
- 发送重定向的路由器的地址 (`src`) 必须等于当前为目的地址设置的 `rt_gateway`;
- 新网关的接口 (由 `ifa_ifwithnet` 返回的 `ifa` 结构) 必须等于当前为目的地址设置的接口 (`rt_ifa`), 也就是说, 新网关必须和当前网关在同一个网络上; 并且

- 新网关不能把到这个主机的路由改变为到它自己，也就是说，不能存在与 gateway 相等的带有单播地址或广播地址的连接着的接口。

route.c

```

147 int
148 rtredirect(dst, gateway, netmask, flags, src, rtp)
149 struct sockaddr *dst, *gateway, *netmask, *src;
150 int flags;
151 struct rtentry **rtp;
152 {
153     struct rtentry *rt;
154     int error = 0;
155     short *stat = 0;
156     struct rt_addrinfo info;
157     struct ifaddr *ifa;

158     /* verify the gateway is directly reachable */
159     if ((ifa = ifa_ifwithnet(gateway)) == 0) {
160         error = ENETUNREACH;
161         goto out;
162     }
163     rt = rtalloc1(dst, 0);
164     /*
165      * If the redirect isn't from our current router for this dst,
166      * it's either old or wrong. If it redirects us to ourselves,
167      * we have a routing loop, perhaps as a result of an interface
168      * going down recently.
169      */
170 #define equal(a1, a2) (bcmp((caddr_t)(a1), (caddr_t)(a2), (a1)->sa_len) == 0)
171     if (!(flags & RTF_DONE) && rt &&
172         (!equal(src, rt->rt_gateway) || rt->rt_ifa != ifa))
173         error = EINVAL;
174     else if (ifa_ifwithaddr(gateway))
175         error = EHOSTUNREACH;
176     if (error)
177         goto done;
178     /*
179      * Create a new entry if we just got back a wildcard entry
180      * or if the lookup failed. This is necessary for hosts
181      * which use routing redirects generated by smart gateways
182      * to dynamically build the routing tables.
183      */
184     if ((rt == 0) || (rt_mask(rt) && rt_mask(rt)->sa_len < 2))
185         goto create;

```

route.c

图19-14 rtredirect 函数：验证收到的重定向

### 3. 必须创建一个新路由

178-185 如果到达目的地址的路由没有找到，或找到的路由表项是默认路由，则为该目的地址创建一个新的路由。如程序注释所述，对于可访问多个路由器的主机来说，当默认路由器出错时，它可以利用这种机制来获悉正确的路由器。判断是否为默认路由的测试方法是查看该路由表项是否具有相关的掩码以及掩码的长度字段是否小于 2，因为默认路由的掩码是 rn\_zeros(图18-35)。

图19-15给出了rtredirect函数的后半部分。

### 4. 创建新的主机路由

186-195 如果到达目的地址的当前路由是一个网络路由，并且重定向是主机重定向而不是

网络重定向，那么就为该目的地址建立一个主机路由，而不必去管现存的网络路由。我们要提示的是，flags参数总是指明RTF\_HOST标志，因为Net/3 ICMP把所有收到的重定向都看成主机重定向。

```

186      /*
187      * Don't listen to the redirect if it's
188      * for a route to an interface.
189      */
190      if (rt->rt_flags & RTF_GATEWAY) {
191          if (((rt->rt_flags & RTF_HOST) == 0) && (flags & RTF_HOST)) {
192              /*
193              * Changing from route to net => route to host.
194              * Create new route, rather than smashing route to net.
195              */
196              create:
197              flags |= RTF_GATEWAY | RTF_DYNAMIC;
198              error = rtrequest((int) RTM_ADD, dst, gateway,
199                              netmask, flags,
200                              (struct rtentry **) 0);
201              stat = &rtstat.rts_dynamic;
202          } else {
203              /*
204              * Smash the current notion of the gateway to
205              * this destination. Should check about netmask!!!
206              */
207              rt->rt_flags |= RTF_MODIFIED;
208              flags |= RTF_MODIFIED;
209              stat = &rtstat.rts_newgateway;
210              rt_setgate(rt, rt_key(rt), gateway);
211          }
212      } else
213          error = EHOSTUNREACH;
214  done:
215      if (rt) {
216          if (rtp && !error)
217              *rtp = rt;
218          else
219              rtfree(rt);
220      }
221  out:
222      if (error)
223          rtstat.rts_badredirect++;
224      else if (stat != NULL)
225          (*stat)++;
226      bzero((caddr_t) &info, sizeof(info));
227      info.rti_info[RTAX_DST] = dst;
228      info.rti_info[RTAX_GATEWAY] = gateway;
229      info.rti_info[RTAX_NETMASK] = netmask;
230      info.rti_info[RTAX_AUTHOR] = src;
231      rt_missmsg(RTM_REDIRECT, &info, flags, error);
232  }

```

route.c

图19-15 rtrequest 函数：后半部分

### 5. 创建路由

196-201 rtrequest创建一个新路由，并将标志RTF\_GATEWAY和RTF\_DYNAMIC置位。参数netmask是一个空指针，这是因为新路由是一个主机路由，它的掩码是隐含的全1比特。

stat指向一个计数器，它在后面的程序里递增。

#### 6. 改变现存的主机路由

202-211 当到达目的地址的当前路由已经是一个主机路由时，才执行这段代码。此时，不需要创建新的表项，但需要修改现存的表项：设置RTF\_MODIFIED标志，并调用rt\_setgate来修改路由表项的rt\_gateway字段，使其指向新的网关地址。

#### 7. 如果目的地址直接相连，则忽略

212-213 如果到达目的地址的当前路由是一个直接路由（没有设置RTF\_GATEWAY标志），那么该重定向针对的是一个已直接连接的目的地址。此时，函数返回 EHOSTUNREACH。

#### 8. 返回指针并递增统计值

214-225 如果找到了路由表项，那么该表项被返回（如果rtp非空且没有出错）或者用rtfree释放它。相关的统计值被递增。

#### 9. 产生选路消息

226-232 rt\_addrinfo结构清零，并由 rt\_missmsg产生一个选路插口消息。raw\_input把该消息发送到所有对重定向感兴趣的进程。

## 19.8 选路消息的结构

选路消息由一个定长的首部和至多 8 个插口地址结构组成。该定长首部是下列三种结构中的一个：

- rt\_msghdr
- if\_msghdr
- ifa\_msghdr

图18-11给出了产生不同消息的函数的概观图，图 18-9给出了每种消息类型所使用的结构。选路消息三种首部结构的前三个成员的数据类型及其含义是相同的，分别为：消息的长度、版本和类型。这样，消息接受者就可以对消息进行解码了。而且，每种结构都各有一个成员来编码首部之后 8 个可能的插口地址结构：rtm\_addrs、ifm\_addrs和ifam\_addrs成员，它们都是一个比特掩码。

图19-16给出了最常用的结构，即 rt\_msghdr。RTM\_IFINFO消息使用了图 19-17中的 if\_msghdr结构。RTM\_NEWADDR和RTM\_DELADDR消息使用图19-18中的 ifa\_msghdr结构。

```

139 struct rt_msghdr {
140     u_short rtm_msglen;           /* to skip over non-understood messages */
141     u_char  rtm_version;         /* future binary compatibility */
142     u_char  rtm_type;            /* message type */
143     u_short rtm_index;           /* index for associated ifp */
144     int     rtm_flags;           /* flags, incl. kern & message, e.g. DONE */
145     int     rtm_addrs;           /* bitmask identifying sockaddrs in msg */
146     pid_t   rtm_pid;            /* identify sender */
147     int     rtm_seq;            /* for sender to identify action */
148     int     rtm_errno;          /* why failed */
149     int     rtm_use;            /* from rtenry */
150     u_long  rtm_inits;          /* which metrics we are initializing */
151     struct rt_metrics rtm_rmx;   /* metrics themselves */
152 };

```

route.h

route.h

图19-16 rt\_msghdr 结构

```

235 struct if_msghdr {
236     u_short ifm_msglen;          /* to skip over non-understood messages */
237     u_char  ifm_version;        /* future binary compatability */
238     u_char  ifm_type;           /* message type */
239
239     int      ifm_addrs;          /* like rtm_addrs */
240     int      ifm_flags;          /* value of if_flags */
241     u_short  ifm_index;         /* index for associated ifp */
242     struct if_data ifm_data;    /* statistics and other data about if */
243 };

```

if.h

图19-17 if\_msghdr 结构

```

248 struct ifa_msghdr {
249     u_short ifam_msglen;        /* to skip over non-understood messages */
250     u_char  ifam_version;      /* future binary compatability */
251     u_char  ifam_type;         /* message type */
252
252     int      ifam_addrs;        /* like rtm_addrs */
253     int      ifam_flags;        /* value of ifa_flags */
254     u_short  ifam_index;       /* index for associated ifp */
255     int      ifam_metric;      /* value of ifa_metric */
256 };

```

if.h

图19-18 ifa\_msghdr 结构

注意，这三种不同结构的前三个成员具有相同的数据结构和含义。

三个变量rtm\_addrs、ifm\_addrs和ifam\_addrs都是比特掩码，它们定义了首部之后的插口地址结构。图19-19给出了比特掩码用到的一些常量。

比特掩码		数组索引		rtsock.c 中的名字	描 述
常 量	值	常 量	值		
RTA_DST	0x01	RTAX_DST	0	dst	目的插口地址结构
RTA_GATEWAY	0x02	RTAX_GATEWAY	1	gate	网关插口地址结构
RTA_NETMASK	0x04	RTAX_NETMASK	2	netmask	网络掩码插口地址结构
RTA_GENMASK	0x08	RTAX_GENMASK	3	genmask	克隆掩码插口地址结构
RTA_IFP	0x10	RTAX_IFP	4	ifpaddr	接口名称插口地址结构
RTA_IFA	0x20	RTAX_IFA	5	ifaaddr	接口地址插口地址结构
RTA_AUTHOR	0x40	RTAX_AUTHOR	6		重定向产生者的插口地址结构
RTA_BRD	0x80	RTAX_BRD	7	brdaddr	广播或点到点的目的地址
		RTAX_MAX	8		rti-into[]数组的元素个数

图19-19 用来引用rti\_info 数组成员的常量

比特掩码的值可以用常数1左移数组下标指定的位数而得到。例如，0x20(RTA\_IFA)是1左移五位(RTAX\_IFA)。我们会在代码中看到这个过程。

插口地址结构总是按照数组下标递增的次序，一个接一个地出现的。例如，如果掩码是0x87，则第一个插口地址结构的内容为目的地址，接着是网关地址，网络掩码，最后是广播地址。

内核用图19-19中的数组下标来引用rt\_addrinfo结构，如图19-20所示。该结构具有与我们所述相同的比特掩码，以表示哪些地址存在。它的另一个成员指向那些插口地址结构。

```

199 struct rt_addrinfo {                                route.h
200     int      rti_addrs;                               /* bitmask, same as rtm_addrs */
201     struct sockaddr *rti_info[RTAX_MAX];
202 };

```

route.h

图19-20 `rt_addrinfo` 结构：表示哪些地址存在的掩码和指向这些地址的指针

例如，如果 `rti_addrs` 成员中设置了 `RTA_GATEWAY` 比特，则 `rti_info[RTA_GATEWAY]` 成员就是含网关地址的插口地址结构的指针。对于 Internet 协议，该插口地址结构就是含网关的 IP 地址的 `sockaddr_in` 结构。

图19-19中的第五栏给出了文件 `rtsock.c` 中 `rti_info` 数组成员相应的名称。它们的定义形式如下：

```
#define dst_info.rti_info[RTAX_DST]
```

在本章的很多源文件中我们都将遇到这些名称。元素 `RTAX_AUTHOR` 没有命名，因为进程不会向内核传递该元素。

我们已经有两次遇到过 `rt_addrinfo` 结构：在函数 `rtalloc1` (图19-2) 和 `rtredirect` 中 (图19-14)。图19-21给出了 `rtalloc1` 在路由表查找失败后调用 `rt_missmsg` 时创建的该结构的格式。

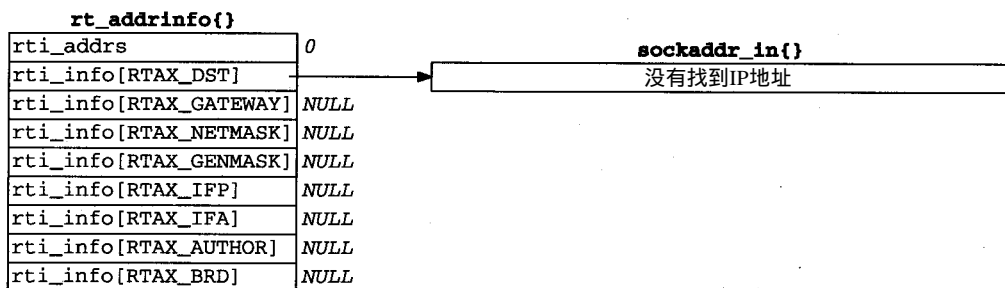


图19-21 `rtalloc1` 传递给 `rt_missmsg` 的 `rt_addrinfo` 结构

所有未使用的指针都是空指针，因为该结构在使用前被设置成 0。还要指出的是，`rti_addrs` 成员没有被初始化成相应的比特掩码，因为在内核使用该结构时，`rti_info` 数组中的指针为空就代表不存在该插口地址结构。仅在进程和内核之间传递的消息里该掩码才是必不可少的。

图19-22给出了 `rtredirect` 调用 `rt_missmsg` 时创建的该结构的格式。

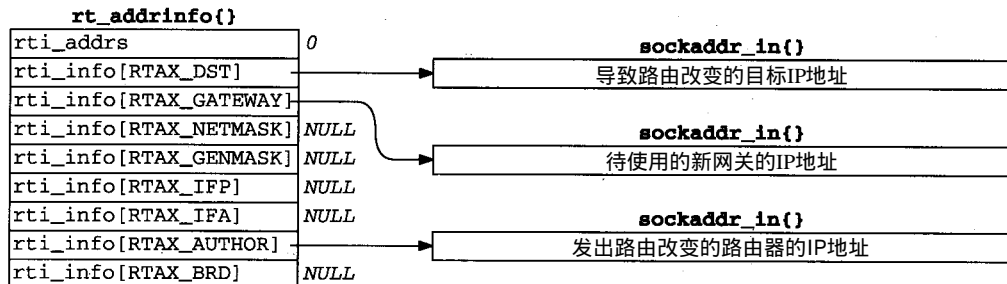


图19-22 `rtredirect` 传递给 `rt_missmsg` 的 `rt_addrinfo` 结构

下一节中将介绍该结构是如何被放置在发送到其他进程的消息里的。

图19-23给出了route\_cb结构，我们会在下一节中遇到。该结构由四个计数器组成，前三个分别针对IP、XNS和OSI协议，最后一个为“任意”计数器。每个计数器分别记录相应的域中当前存在的选路插口的数目。

203-208 内核跟踪选路插口监听器的数目。这样，当不存在等待消息的进程时，内核就可以避免创建选路消息以及调用raw\_input发送该消息。

```

203 struct route_cb {                                     route.h
204     int     ip_count;                                /* IP */
205     int     ns_count;                                /* XNS */
206     int     iso_count;                               /* ISO */
207     int     any_count;                               /* sum of above three counters */
208 };

```

图19-23 route\_cb 结构：选路插口监听器的计数器

## 19.9 rt\_missmsg函数

如图19-24所示，rt\_missmsg函数使用了图19-21和图19-22中的结构，并调用rt\_msg1在mbuf链中为进程创建了相应的变长消息，之后调用raw\_input将该mbuf链传递给所有相关的选路插口。

```

516 void
517 rt_missmsg(type, rtinfo, flags, error)                rtsock.c
518 int     type, flags, error;
519 struct rt_addrinfo *rtinfo;
520 {
521     struct rt_msghdr *rtm;
522     struct mbuf *m;
523     struct sockaddr *sa = rtinfo->rti_info[RTAX_DST];
524     if (route_cb.any_count == 0)
525         return;
526     m = rt_msg1(type, rtinfo);
527     if (m == 0)
528         return;
529     rtm = mtod(m, struct rt_msghdr *);
530     rtm->rtm_flags = RTF_DONE | flags;
531     rtm->rtm_errno = error;
532     rtm->rtm_addrs = rtinfo->rti_addrs;
533     route_proto.sp_protocol = sa ? sa->sa_family : 0;
534     raw_input(m, &route_proto, &route_src, &route_dst);
535 }

```

图19-24 rt\_missmsg 函数

516-525 如果没有任何选路插口监听器，则函数立即退出。

### 1. 在mbuf链中创建消息

526-528 rt\_msg1(19.12节)在mbuf链中创建相应的消息，并返回该链的指针。利用图19-22中的rt\_addrinfo结构，图19-25给出了所得到的mbuf链的一个例子。这些信息之所以要放在一个mbuf链中，是因为raw\_input要调用sbappendaddr把该mbuf链添加到插口接收

缓存的尾部。

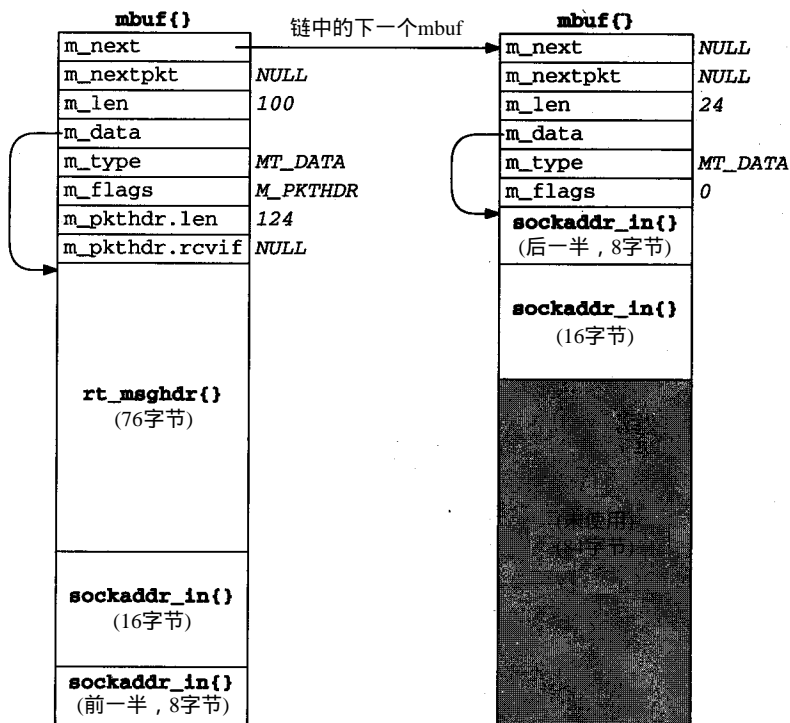


图19-25 由rt\_msg1 创建的对应于图19-22的mbuf链

## 2. 完成消息的创建

529-532 成员rtm\_flags和rtm\_errno被设置成调用者传递的值。成员rtm\_addrs的值是由rti\_addrs复制而得到的。在图19-21和图19-22中，我们给出的rti\_addrs值为0，因此，rt\_msg1依据rti\_info数组中的指针是否为空，来计算并保存相应的比特掩码。

## 3. 设置消息的协议，调用raw\_input

533-534 raw\_input的后三个参数指定了选路消息的协议、源和目的。这三个结构被初始化成：

```
struct sockaddr route_dst = { 2, PF_ROUTE, };
struct sockaddr route_src = { 2, PF_ROUTE, };
struct sockproto route_proto = { PF_ROUTE };
```

内核不会修改前两个结构。第三个结构 sockproto，我们第一次遇到。图19-26给出了它的定义。

```
128 struct sockproto {
129     u_short sp_family;          /* address family */
130     u_short sp_protocol;       /* protocol */
131 };
socket.h
socket.h
```

图19-26 sockproto 结构

该结构的协议族成员一直保持了它的初始值 PF\_ROUTE，但其协议成员的值在每一次调

用raw\_input时都要进行设置。当进程调用 socket 创建选路插口时，第三个参数定义了该进程所感兴趣的协议。raw\_input的调用者再把route\_proto结构的sp\_protocol成员设置成选路消息的协议。在rt\_missmsg这种情况下，该成员被设置成目的插口地址结构的sa\_family(如果调用者指定了该值)，在图19-21和图19-22中，其值为AF\_INET。

## 19.10 rt\_ifmsg函数

在图4-30中我们可以看出，if\_up和if\_down都调用了图19-27中的rt\_ifmsg。在接口连接或断开时，该函数被用来产生一个选路插口消息。

```
540 void  
541 rt_ifmsg(ifp)  
542 struct ifnet *ifp;  
543 {  
544     struct if_msghdr *ifm;  
545     struct mbuf *m;  
546     struct rt_addrinfo info;  
  
547     if (route_cb.any_count == 0)  
548         return;  
  
549     bzero((caddr_t) & info, sizeof(info));  
550     m = rt_msg1(RTM_IFINFO, &info);  
551     if (m == 0)  
552         return;  
  
553     ifm = mtod(m, struct if_msghdr *);  
554     ifm->ifm_index = ifp->if_index;  
555     ifm->ifm_flags = ifp->if_flags;  
556     ifm->ifm_data = ifp->if_data; /* structure assignment */  
557     ifm->ifm_addrs = 0;  
  
558     route_proto.sp_protocol = 0;  
559     raw_input(m, &route_proto, &route_src, &route_dst);  
560 }
```

rtsock.c

图19-27 rt\_ifmsg 函数

547-548 如果没有选路插口监听器，函数立即退出。

### 1. 在mbuf链中创建消息

549-552 rt\_addrinfo结构被设置成0。rt\_msg1在一个mbuf链中创建相应的消息。需要注意的是，rt\_addrinfo结构中的所有指针都为空，选路消息仅由定长的 if\_msghdr结构组成，而不含任何地址。

### 2. 完成消息的创建

553-557 把接口的索引、标志和 if\_data结构复制到mbuf中的报文里。把ifm\_addrs比特掩码设置成0。

### 3. 设置消息的协议，调用raw\_input

558-559 因为该消息可应用于所有的协议，所以其协议被设置成 0。并且该消息是关于某接口的，而不是针对特定的目的地。raw\_input把该消息传递给相应的监听器。

## 19.11 rt\_newaddrmsg函数

从图19-13中可以看到，在接口上添加或从中删除一个地址时，rtinit要以RTM\_ADD或RTM\_DELETE为参数调用rt\_newaddrmsg。图19-28给出了rt\_newaddrmsg函数的前半部分。

```

569 void
570 rt_newaddrmsg(cmd, ifa, error, rt)
571 int      cmd, error;
572 struct ifaddr *ifa;
573 struct rtentry *rt;
574 {
575     struct rt_addrinfo info;
576     struct sockaddr *sa;
577     int      pass;
578     struct mbuf *m;
579     struct ifnet *ifp = ifa->ifa_ifp;

580     if (route_cb.any_count == 0)
581         return;

582     for (pass = 1; pass < 3; pass++) {
583         bzero((caddr_t) & info, sizeof(info));
584         if ((cmd == RTM_ADD && pass == 1) ||
585             (cmd == RTM_DELETE && pass == 2)) {
586             struct ifa_msghdr *ifam;
587             int      ncmd = cmd == RTM_ADD ? RTM_NEWADDR : RTM_DELADDR;

588             ifaaddr = sa = ifa->ifa_addr;
589             ifpaddr = ifp->if_addrlist->ifa_addr;
590             netmask = ifa->ifa_netmask;
591             brdaddr = ifa->ifa_dstaddr;
592             if ((m = rt_msg1(ncmd, &info)) == NULL)
593                 continue;
594             ifam = mtod(m, struct ifa_msghdr *);
595             ifam->ifam_index = ifp->if_index;
596             ifam->ifam_metric = ifa->ifa_metric;
597             ifam->ifam_flags = ifa->ifa_flags;
598             ifam->ifam_addrs = info.rti_addrs;
599         }

```

图19-28 rt\_newaddrmsg 函数的前半部分：创建 ifa\_msghdr

580-581 如果没有选路插口监听器，函数立即退出。

#### 1. 产生两个选路消息

582 本函数要产生两个选路报文，一个用于提供有关接口的信息，另一个提供有关地址的信息。因此，for循环执行两次，每次产生一个消息。如果命令是 RTM\_ADD，则第一个消息的类型是RTM\_NEWADDR，第二个消息的类型是RTM\_ADD；如果命令是RTM\_DELETE，则第一个消息的类型是RTM\_DELETE，第二个消息的类型是RTM\_DELADDR。RTM\_NEWADDR和RTM\_DELADDR消息的首部为ifa\_msghdr结构，而RTM\_ADD和RTM\_DELETE消息的首部为rt\_msghdr结构。

583 rt\_addrinfo结构被设置为0。

#### 2. 产生至多含四个地址的消息

588-591 这四个插口地址结构包含的是有关被添加或删除的接口地址的信息，它们的指针存储于rti\_info数组中。其中ifaaddr、ifpaddr、netmask和brdaddr引用的是名为info的rti\_info数组中的成员，如图19-19所示。rt\_msg1在mbuf链中创建了相应的消息。注意，sa也设置成指向ifa\_addr结构的指针，我们将在函数的尾部看到选路消息的协议被设置成该插口地址结构的族。

把ifa\_msghdr结构的其他成员设置成接口的索引、度量和标志。其比特掩码由rt\_msg1设置。

图19-29给出了rt\_newaddrmsg的后半部分。该部分用于创建rt\_msghdr消息，该消息中包含了有关被添加或删除的路由表项的信息。

### 3. 创建消息

600-609 rt\_mask、rt\_key和rt\_gateway这三个地址结构的指针存放在rti\_info数组中。sa被设置成目的地址的指针，它的族将成为选路消息的协议。rt\_msg1在mbuf链中创建相应的消息。

设置其余的rt\_msghdr结构成员。其中，比特掩码由rt\_msg1设置。

### 4. 设置消息的协议，调用raw\_input

616-619 设置消息的协议，并由raw\_input把消息发送给相应的监听器。函数在完成了两次循环后返回。

```

600          if ((cmd == RTM_ADD && pass == 2) ||
601              (cmd == RTM_DELETE && pass == 1)) {
602              struct rt_msghdr *rtm;

603              if (rt == 0)
604                  continue;
605              netmask = rt_mask(rt);
606              dst = sa = rt_key(rt);
607              gate = rt->rt_gateway;
608              if ((m = rt_msg1(cmd, &info)) == NULL)
609                  continue;
610              rtm = mtod(m, struct rt_msghdr *);
611              rtm->rtm_index = ifp->if_index;
612              rtm->rtm_flags |= rt->rt_flags;
613              rtm->rtm_errno = error;
614              rtm->rtm_addrs = info.rti_addrs;
615          }
616          route_proto.sp_protocol = sa ? sa->sa_family : 0;
617          raw_input(m, &route_proto, &route_src, &route_dst);
618      }
619 }

```

rtsock.c

rtsock.c

图19-29 rt\_newaddrmsg 函数的后半部分：创建rt\_msghdr 消息

## 19.12 rt\_msg1函数

前三节的函数都调用rt\_msg1函数来创建一个相应的选路消息。图19-25还给出了一个mbuf链，该链是由rt\_msg1按照图19-22中的rt\_msghdr和rt\_addrinfo结构创建的。图19-30给出了本函数的代码。

rtsock.c

```
399 static struct mbuf *
400 rt_msg1(type, rtinfo)
401 int     type;
402 struct rt_addrinfo *rtinfo;
403 {
404     struct rt_msghdr *rtm;
405     struct mbuf *m;
406     int     i;
407     struct sockaddr *sa;
408     int     len, dlen;
409
410     m = m_gethdr(M_DONTWAIT, MT_DATA);
411     if (m == 0)
412         return (m);
413     switch (type) {
414     case RTM_DELAADDR:
415     case RTM_NEWADDR:
416         len = sizeof(struct ifa_msghdr);
417         break;
418     case RTM_IFINFO:
419         len = sizeof(struct if_msghdr);
420         break;
421     default:
422         len = sizeof(struct rt_msghdr);
423     }
424     if (len > MHLEN)
425         panic("rt_msg1");
426     m->m_pkthdr.len = m->m_len = len;
427     m->m_pkthdr.rcvif = 0;
428     rtm = mtod(m, struct rt_msghdr *);
429     bzero((caddr_t) rtm, len);
430
431     for (i = 0; i < RTAX_MAX; i++) {
432         if ((sa = rtinfo->rta_info[i]) == NULL)
433             continue;
434         rtm->rta_addrs |= (1 << i);
435         dlen = ROUNDUP(sa->sa_len);
436         m_copyback(m, len, dlen, (caddr_t) sa);
437         len += dlen;
438     }
439     if (m->m_pkthdr.len != len) {
440         m_freem(m);
441         return (NULL);
442     }
443     rtm->rtm_msglen = len;
444     rtm->rtm_version = RTM_VERSION;
445     rtm->rtm_type = type;
446     return (m);
447 }
```

rtsock.c

图19-30 rt\_msg1 函数：获取并初始化mbuf

#### 1. 得到mbuf并确定消息首部的长度

399-422 获得一个含分组首部的mbuf，并将分组消息定长部分的长度存入len中。图18-9中各种类型的消息里，有两个使用ifa\_msghdr结构，有一个使用if\_msghdr结构，其余的九个使用rt\_msghdr结构。

## 2. 验证结构是否适合 mbuf

423-424 定长结构的大小必须完全适合分组首部 mbuf的数据部分，因为该 mbuf指针将被 mtdot 转换成一个结构指针，之后通过指针来引用该结构。三个结构中最大的为 if\_msghdr，其长度为84，小于MHLEN(100)。

## 3. 初始化mbuf分组首部并使结构清零

425-428 初始化分组首部的两个字段，并将 mbuf 中的结构设置成0。

## 4. 将插口地址结构复制到mbuf链中

429-436 调用者传递了一个 rt\_addrinfo 结构的指针。与 rti\_info 中所有非空指针相对应的插口地址结构都被 m\_copyback 复制到 mbuf 里。将数值1左移下标 RTX\_xxx 对应的位数就可以得到相应的 RTA\_xxx 比特掩码 (图19-19)。将每个比特掩码用逻辑或添加到 rti\_addrs 成员中去，调用者在函数返回时可将它保存为相应的报文结构成员。ROUNDUP 宏将每个插口地址结构的大小上舍入成下一个4的倍数个字节。

437-440 在循环结束时，如果 mbuf 分组首部的长度不等于 len，则表明函数 m\_copyback 没能获得所需的 mbuf。

## 5. 保存长度、版本和类型

441-445 把长度、版本和报文类型保存到报文结构的前三个成员中。再次说明一下，因为所有的三种 xxx\_msghdr 结构都以相同的成员开始，所以尽管代码中的指针 rtm 是一个 rt\_msghdr 结构的指针，但它可以处理所有这三种情况。

## 19.13 rt\_msg2函数

rt\_msg1 在 mbuf 链中创建一个选路消息，调用它的三个函数接着又调用 raw\_input，从而把 mbuf 结构附加到一个或多个插口的接收缓存中去。与 rt\_msg1 不同，rt\_msg2 在存储器缓存中创建选路消息，而不是在 mbuf 链中创建。并且 rt\_msg2 有一个 walkarg 结构的参数，在选路域中处理 sysctl 系统调用时，有两个函数使用该参数调用了 rt\_msg2。以下是两种调用 rt\_msg2 的情况：

- 1) route\_output 调用它处理 RTM\_GET 命令
- 2) sysctl\_dumpentry 和 sysctl\_iflist 调用它处理 sysctl 系统调用。

在给出 rt\_msg2 的代码之前，图19-31给出了在第2种情况下使用的 walkarg 结构的定义。我们在遇到它的各成员时再一一介绍。

```

41 struct walkarg {
42     int      w_op;                /* NET_RT_xxx */
43     int      w_arg;              /* RTF_xxx for FLAGS, if_index for IFLIST */
44     int      w_given;            /* size of process' buffer */
45     int      w_needed;           /* #bytes actually needed (at end) */
46     int      w_tmemsiz;          /* size of buffer pointed to by w_tmemb */
47     caddr_t  w_where;            /* ptr to process' buffer (maybe null) */
48     caddr_t  w_tmemb;            /* ptr to our malloc'ed buffer */
49 };

```

rtsock.c

图19-31 walkarg 结构：选路域内 sysctl 系统调用中使用

图19-32给出了 rt\_msg2 函数的前半部分，它与 rt\_msg1 的前半部分类似。

rtsock.c

```

446 static int
447 rt_msg2(type, rtinfo, cp, w)
448 int      type;
449 struct rt_addrinfo *rtinfo;
450 caddr_t cp;
451 struct walkarg *w;
452 {
453     int      i;
454     int      len, dlen, second_time = 0;
455     caddr_t cp0;
456     rtinfo->rta_addrs = 0;
457 again:
458     switch (type) {
459     case RTM_DELADDR:
460     case RTM_NEWADDR:
461         len = sizeof(struct ifa_msghdr);
462         break;
463     case RTM_IFINFO:
464         len = sizeof(struct if_msghdr);
465         break;
466     default:
467         len = sizeof(struct rt_msghdr);
468     }
469     if (cp0 = cp)
470         cp += len;
471     for (i = 0; i < RTAX_MAX; i++) {
472         struct sockaddr *sa;
473         if ((sa = rtinfo->rta_info[i]) == 0)
474             continue;
475         rtinfo->rta_addrs |= (1 << i);
476         dlen = ROUNDUP(sa->sa_len);
477         if (cp) {
478             bcopy((caddr_t) sa, cp, (unsigned) dlen);
479             cp += dlen;
480         }
481         len += dlen;
482     }

```

rtsock.c

图19-32 rt\_msg2 函数：复制插口地址结构

446-455 本函数将选路消息保存在一个存储器缓存中，调用者用 `cp` 参数指定该缓存的起始位置。调用者必须保证缓存足够长，以容纳所产生的选路消息。当 `cp` 参数为空时，`rt_msg2` 不保存任何结果而是处理输入参数，并返回保存结果所需要的字节总数。这样可以帮助调用者确定缓存的大小。我们可以看到 `route_output` 就利用了这一机制，它调用本函数两次：第一次确定缓存的大小；在获得了大小无误的缓存后，再次调用本函数以保存结果。`route_output` 调用本函数时，最后一个参数为空，但如果是作为 `sysctl` 系统调用处理的一部分被调用时，该参数就不是空指针了。

#### 1. 确定结构的大小

458-470 定长消息结构的大小是根据消息类型来确定的。如果 `cp` 指针非空，则把大小等于定长消息结构长度的偏移量添加到 `cp` 指针上去。

#### 2. 复制插口地址结构

471-482 for循环查看rti\_info数组的每个元素。遇到非空指针时,设置rti\_addrs比特掩码中的相应比特,并将该插口地址结构复制到cp中(如果cp指针非空),并修改长度变量。

图19-33给出了rt\_msg2函数的后半部分。其代码用于处理可选参数walkarg结构。

```

483     if (cp == 0 && w != NULL && !second_time) {                                rtsock.c
484         struct walkarg *rw = w;
485         rw->w_needed += len;
486         if (rw->w_needed <= 0 && rw->w_where) {
487             if (rw->w_tmemsiz < len) {
488                 if (rw->w_tmemb
489                     free(rw->w_tmemb, M_RTABLE);
490                 if (rw->w_tmemb = (caddr_t)
491                     malloc(len, M_RTABLE, M_NOWAIT))
492                     rw->w_tmemsiz = len;
493             }
494             if (rw->w_tmemb) {
495                 cp = rw->w_tmemb;
496                 second_time = 1;
497                 goto again;
498             } else
499                 rw->w_where = 0;
500         }
501     }
502     if (cp) {
503         struct rt_msghdr *rtm = (struct rt_msghdr *) cp0;
504         rtm->rtm_version = RTM_VERSION;
505         rtm->rtm_type = type;
506         rtm->rtm_msglen = len;
507     }
508     return (len);
509 }

```

图19-33 rt\_msg2 函数:处理可选参数walkarg

483-484 当调用者传递了一个非空的walkarg结构指针且函数第一次执行到这里时,该if语句的判断条件才为真。变量second\_time被初始化成0,它将在本if语句中被设置成1,然后程序往回跳转到图19-32中的标号again处。cp为空指针的测试是不必要的,因为当w指针非空时,cp指针一定是空,反之亦然。

### 3. 检查是否要保存数据

485-486 w\_needed将增大,其增量为报文长度的值。该变量的初始值为0减去sysctl函数中用户缓存的长度。例如,如果该缓存为500比特,则w\_needed的初始值为-500。只要该变量为负值,则表明缓存内还有剩余空间。在调用进程中,w\_where是指向该缓存的指针。w\_where为空表明调用进程不想要函数的处理结果,而仅想获得sysctl处理结果的大小。因此,当w\_where为空时,rt\_msg2没有必要将数据复制给进程,也就是返回给调用者。同样,rt\_msg2也没有必要为保存结构而申请缓存,也不需要返回到标号again处再次执行,因为再次执行是为了把结果填入到缓存里。本函数的处理实际上只有五种情况,如图19-34所示。

### 4. 第一次调用时或消息长度增加时分配缓存

487-493 w\_tmemsiz是w\_tmemb所指向的缓存的长度。它被sysctl\_rtable初始化成0。

因此，对于给定的 sysctl 请求，在第一次调用 rt\_msg2 时，必须为它分配一个缓存。同样，当产生的结果的长度增加时，必须释放原有的缓存，并重新分配一个更大的缓存。

调用者	cp	w	w.w_where	second_time	描述
route_output	空	空			希望返回长度
	非空	空			希望返回结果
sysctl_rtable	空	非空	空	0	进程希望返回长度
	空	非空	非空	0	第一次执行，计算长度
	非空	非空	非空	1	第二次执行，保存结果

图19-34 rt\_msg2 函数的五种执行情况

### 5. 返回再执行一次并保存结果

494-499 如果 w\_tmemsiz 非空，则表明该缓存已经存在或刚被分配。设置 cp 指向该缓存，把 second\_time 设置成 1，跳转至标号 again 处。因为 second\_time 的值为 1，所以第二次执行到本图的第一个语句时，if 语句的判断不再为真。如果 w\_tmemb 为空，则表明调用 malloc 失败，因此，把进程中的缓存指针设置成空指针以阻止返回任何结果。

### 6. 保存长度、版本和类型

502-509 如果 cp 非空，则保存消息首部的前三个成员。函数返回报文的长度。

## 19.14 sysctl\_rtable 函数

本函数处理选路插口的 sysctl 系统调用。如图 18-11 所示，net\_sysctl 函数调用了该函数。

在解释其源代码之前，图 19-35 给出了该系统调用关于路由表的一种典型的用法。这个例子来自于 arp 程序。

```

int      mib[6];
size_t   needed;
char      *buf, *lim, *next;
struct rt_msghdr *rtm;

mib[0] = CTL_NET;
mib[1] = PF_ROUTE;
mib[2] = 0;
mib[3] = AF_INET;      /* address family; can be 0 */
mib[4] = NET_RT_FLAGS; /* operation */
mib[5] = RTF_LLINFO;   /* flags; can be 0 */

if (sysctl(mib, 6, NULL, &needed, NULL, 0) < 0)
    quit("sysctl error, estimate");

if ( (buf = malloc(needed)) == NULL)
    quit("malloc");

if (sysctl(mib, 6, buf, &needed, NULL, 0) < 0)
    quit("sysctl error, retrieval");

lim = buf + needed;
for (next = buf; next < lim; next += rtm->rtm_msglen) {
    rtm = (struct rt_msghdr *)next;
    ... /* do whatever */
}

```

图 19-35

mib数组的前三个元素引导内核调用 `sysctl_rtable`，以处理其余的元素。

`mib[4]`用于指定操作的类型，共支持3种操作类型。

1) `NET_RT_DUMP`：返回 `mib[3]`指定的地址族所对应的路由表。如果地址族为 0，则返回所有的路由表。

针对每一个路由表项，程序都将返回一个 `RTM_GET`选路消息。每个消息可能包含两个、三个或四个插口地址结构。这些地址结构由指针 `rt_key`、`rt_gateway`、`rt_netmask`和 `rt_genmask`所指向。其中最后两个指针可能为空。

2) `NET_RT_FLAGS`：与前一个命令相同，但 `mib[5]`指定了一个 `RTF_xxx`标志(图18-25)，程序仅返回那些设置了该标志的表项。

3) `NET_RT_IFLIST`：返回所有已配置接口的信息。如果 `mib[5]`的值不是零，则程序仅返回 `if_index`为相应值的接口。否则，返回所有 `ifnet`链表上的接口。

针对每个接口，将返回一个 `RTM_IFINFO`消息，该消息传递了有关接口本身的一些信息。之后用一个 `RTM_NEWADDR`消息传递接口的 `if_addrlist`上的每个 `ifaddr`结构。如果 `mib[3]`的值为非0，则 `RTM_NEWADDR`消息仅返回那些地址族与 `mib[3]`相匹配的地址。否则，`mib[3]`为0，将返回所有地址的信息。

这个操作是为了替代 `SIOCGIFCONF ioctl`(图4-26)

与该系统调用有关的一个问题是，该系统返回信息的数量是可变的，这种变化取决于路由表表项的数目和接口的数目。因此，第一次调用 `sysctl`所指定的第三个参数通常是个空指针，也就是说，不需要返回任何选路信息，只要把该信息所占的比特数目返回即可。从图 19-35中我们可以看出，进程第一次调用 `sysctl`之后调用了 `malloc`，接着再调用 `sysctl`来获取信息。第二次调用时，通过第四个参数，`sysctl`又返回了该比特数目(该数目与上次相比可能会有变化)。通过该数目我们可以得到指针 `lim`，它指向的位置位于返回的最后一个字节之后。进程接着就遍历缓存中的每个消息，利用 `rtm_msglen`可找到下一个消息。

图19-36给出了不同的Net/3程序访问路由表和接口列表时指定的这六个 `mib`变量的值。

mib[]	arp	route	netstat	routed	gated	rwhod
0	CTL_NET	CTL_NET	CTL_NET	CTL_NET	CTL_NET	CTL_NET
1	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE
2	0	0	0	0	0	0
3	AF_INET	0	0	AF_INET	0	AF_INET
4	NET_RT_FLAGS	NET_RT_DUMP	NET_RT_DUMP	NET_RT_IFLIST	NET_RT_IFLIST	NET_RT_IFLIST
5	RTF_LLINFO	0	0	0	0	0

图19-36 调用 `sysctl` 获取路由表和接口列表的程序举例

前三个程序从路由表中提取路由表项，后三个从接口列表中提取数据。`route`程序仅支持Internet选路协议，所以它指定 `mib[3]`的值为 `AF_INET`，而 `gated`还支持其他协议，所以它对应的 `mib[3]`的值为0。

图19-37画出了三个 `sysctl_xxx`函数的结构，在后面的几节中会逐个予以阐述。

图19-38给出了 `sysctl_rtable`函数。

#### 1. 验证参数

705-719 当进程调用 `sysctl`来设置一个路由表中不支持的变量时，使用 `new`参数。因此该参数必须是一个空指针。

720-721 namelen必须是3，因为系统调用处理到这儿时，name数组中有三个元素：name[0]，地址族(进程中它被指定为mib[3])；name[1]，操作(mib[4])；以及name[2]，标志(mib[5])。

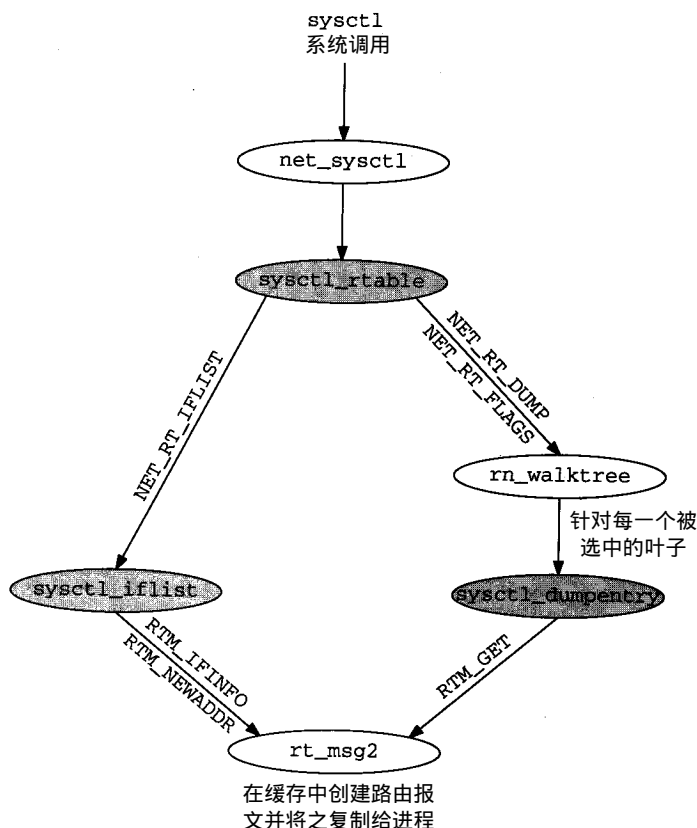


图19-37 支持针对选路插口的 sysctl 系统调用的函数

```

705 int
706 sysctl_rtable(name, namelen, where, given, new, newlen)
707 int *name;
708 int namelen;
709 caddr_t where;
710 size_t *given;
711 caddr_t *new;
712 size_t newlen;
713 {
714     struct radix_node_head *rn timer;
715     int i, s, error = EINVAL;
716     u_char af;
717     struct walkarg w;
718
719     if (new)
720         return (EPERM);
721     if (namelen != 3)
722         return (EINVAL);
723     af = name[0];
724     Bzero(&w, sizeof(w));

```

rtsock.c

图19-38 sysctl\_rtable 函数：处理 sysctl 系统调用请求

```

724     w.w_where = where;
725     w.w_given = *given;
726     w.w_needed = 0 - w.w_given;
727     w.w_op = name[1];
728     w.w_arg = name[2];

729     s = splnet();
730     switch (w.w_op) {

731     case NET_RT_DUMP:
732     case NET_RT_FLAGS:
733         for (i = 1; i <= AF_MAX; i++)
734             if ((rn timer = rt_tables[i]) && (af == 0 || af == i) &&
735                 (error = rn->rn_walktree(rn,
736                                         sysctl_dumpentry, &w)))
737                 break;
738         break;

739     case NET_RT_IFLIST:
740         error = sysctl_iflist(af, &w);
741     }
742     splx(s);
743     if (w.w_tmemo)
744         free(w.w_tmemo, M_RTABLE);
745     w.w_needed += w.w_given;
746     if (where) {
747         *given = w.w_where - where;
748         if (*given < w.w_needed)
749             return (ENOMEM);
750     } else {
751         *given = (11 * w.w_needed) / 10;
752     }
753     return (error);
754 }

```

—rtsock.c

图19-38 (续)

## 2. 初始化walkarg结构

723-728 把walkarg结构(图19-31)设置成0,并初始化下列成员:把w\_where设置成调用进程中为结果准备的缓存地址;w\_given是该缓存的比特数目(当w\_where为空指针时,作为输入参数,该成员没有实际含义,但在输出时它必须被设置成将要返回的结果的长度);w\_needed被设置成缓存的大小的负数;w\_op指明操作类型(值为NET\_RT\_XXX);w\_arg被设置成标志值。

## 3. 导出路由表

731-738 NET\_RT\_DUMP和NET\_RT\_FLAGS操作的处理是相同的:利用一个循环语句遍历所有的路由表(rt\_table数组),如果系统使用了该路由表,并且地址族调用参数为0或地址族调用参数与该路由表的族相匹配,则调用rn\_walktree函数来处理整个路由表。图18-17所给出的rn\_walktree函数是通常使用的rn\_walktree函数。该函数的第二个参数的值是另一个函数的地址,这个函数(sysctl\_dumpentry)将被调用以处理选路树的每一个叶子。rn\_walktree的第三个参数是个任意类型的指针,该指针将传递给sysctl\_dumpentry函数。在这里,它指向一个walkarg结构,该结构包含了有关sysctl调用的所有信息。

#### 4. 返回接口列表

739-740 NET\_RT\_IFLIST操作调用sysctl\_iflist函数来处理所有的ifnet结构。

#### 5. 释放缓存

743-744 如果由rt\_msg2分配的缓存里含有选路消息，则释放该缓存。

#### 6. 更新w\_needed

745 rt\_msg2函数把每个消息的长度都加入到 w\_needed中。而该变量被我们初始化成 w\_given的负数，所以它的值可以表示成：

$$w\_needed = 0 - w\_given + totalbytes$$

式中，totalbytes表示由rt\_msg2函数添加的所有消息的长度总和。通过往 w\_needed中加入w\_given的值，我们就能得到所有消息的字节总数：

$$\begin{aligned} w\_needed &= 0 - w\_given + totalbytes + w\_given \\ &= totalbytes \end{aligned}$$

因为等式中两个w\_given的值最终相互抵消，所以当进程所指定的 w\_where是个空指针时，就没有必要初始化w\_given的值。事实上，图19-35中的变量needed就没有被初始化。

#### 7. 返回报文的实际长度

746-749 如果where指针非空，则通过given指针返回保存在缓存中的字节数。如果返回的数值小于进程指定的缓存的大小，则返回一个差错，因为返回信息被截短了。

#### 8. 返回报文长度的估算值

750-752 当where指针为空时，进程只想获得要返回的字节总数。为了防止在两次sysctl调用之间相应的表被增大，我们将该字节总数扩大了10%。10%这个增量的确定没有特定的理由。

### 19.15 sysctl\_dumpentry函数

在前一节中我们阐述了被sysctl\_rtable调用的rn\_walktree是如何调用本函数的。图19-39给出了本函数的代码。

623-630 每次调用本函数时，第一个参数指向一个 radix\_node结构，同时它也是一个 rtable结构的指针，第二个参数指向一个由 sysctl\_rtable初始化了的walkarg结构。

#### 1. 检测路由表项的标志

631-632 如果进程指定了标志值(mib[5])，则忽略那些rt\_flags成员中没有设置该标志的表项。在图19-36中，我们可以看到arp程序使用这种方法来选择那些设有RTF\_LLINFO标志的表项，因为ARP仅对这些表项感兴趣。

#### 2. 构造选路消息

633-638 rti\_info数组中的下列四个指针是从路由表项中复制而得的：dst、gate、netmask和genmask。前两个总是非空的，但另外两个可以是空指针。调用 rt\_msg2是为了构造一个RTM\_GET消息。

#### 3. 复制消息回传给进程

639-651 如果进程需要返回一个报文，并且rt\_msg2分配了一个缓存，则将选路报文中的其余部分填写到w\_tmem所指向的缓存中去，并调用 copyout复制消息回传给进程。如果复制成功，就增大w\_where，增加的数目等于所复制的字节数目。

```

623 int
624 sysctl_dumpentry(rn, w)
625 struct radix_node *rn;
626 struct walkarg *w;
627 {
628     struct rtentry *rt = (struct rtentry *) rn;
629     int error = 0, size;
630     struct rt_addrinfo info;

631     if (w->w_op == NET_RT_FLAGS && !(rt->rt_flags & w->w_arg))
632         return 0;
633     bzero((caddr_t) & info, sizeof(info));
634     dst = rt_key(rt);
635     gate = rt->rt_gateway;
636     netmask = rt_mask(rt);
637     genmask = rt->rt_genmask;
638     size = rt_msg2(RTM_GET, &info, 0, w);
639     if (w->w_where && w->w_tmem) {
640         struct rt_msghdr *rtm = (struct rt_msghdr *) w->w_tmem;

641         rtm->rtm_flags = rt->rt_flags;
642         rtm->rtm_use = rt->rt_use;
643         rtm->rtm_rmx = rt->rt_rmx;
644         rtm->rtm_index = rt->rt_ifp->if_index;
645         rtm->rtm_errno = rtm->rtm_pid = rtm->rtm_seq = 0;
646         rtm->rtm_addrs = info.rti_addrs;
647         if (error = copyout((caddr_t) rtm, w->w_where, size))
648             w->w_where = NULL;
649         else
650             w->w_where += size;
651     }
652     return (error);
653 }

```

rtsock.c

图19-39 sysctl\_dumpentry 函数：处理一个路由表项

## 19.16 sysctl\_iflist函数

图19-40给出了本函数的代码。本函数由 sysctl\_rtable 直接调用，用来把接口列表返回给进程。

本函数由一个 for 循环组成，该循环从指针 ifnet 开始，针对每个接口重复执行。接着用 while 循环处理每个接口的 ifaddr 结构链表。函数将针对每个接口产生一个 RTM\_IFINFO 选路消息，并且针对每个地址产生一个 RTM\_NEWADDR 消息。

### 1. 检测接口索引

654-666 进程可以指定一个非零的标志参数（图19-36中的 mib[5]）。函数用接口的 if\_index 值与之比较，只有匹配时，才进行处理。

### 2. 创建选路消息

667-670 在 RTM\_IFINFO 消息中只返回了唯一的一个接口地址结构，即 ifpaddr。该 RTM\_IFINFO 消息是由 rt\_msg2 创建的。info 结构中的 ifpaddr 指针被设置成 0，因为该 info 结构还要用来产生后面的 RTM\_NEWADDR 消息。

### 3. 复制消息回传给进程

671-681 如果进程需要返回消息，则填入 if\_msghdr 结构的其余部分，用 copyout 给进

程复制该缓存，并增大w\_where。

rtsock.c

```

654 int
655 sysctl_iflist(af, w)
656 int     af;
657 struct walkarg *w;
658 {
659     struct ifnet *ifp;
660     struct ifaddr *ifa;
661     struct rt_addrinfo info;
662     int     len, error = 0;

663     bzero((caddr_t) & info, sizeof(info));
664     for (ifp = ifnet; ifp; ifp = ifp->if_next) {
665         if (w->w_arg && w->w_arg != ifp->if_index)
666             continue;
667         ifa = ifp->if_addrlist;
668         ifpaddr = ifa->ifa_addr;
669         len = rt_msg2(RTM_IFINFO, &info, (caddr_t) 0, w);
670         ifpaddr = 0;
671         if (w->w_where && w->w_tmem) {
672             struct if_msghdr *ifm;

673             ifm = (struct if_msghdr *) w->w_tmem;
674             ifm->ifm_index = ifp->if_index;
675             ifm->ifm_flags = ifp->if_flags;
676             ifm->ifm_data = ifp->if_data;
677             ifm->ifm_addrs = info.rti_addrs;
678             if (error = copyout((caddr_t) ifm, w->w_where, len))
679                 return (error);
680             w->w_where += len;
681         }
682         while (ifa = ifa->ifa_next) {
683             if (af && af != ifa->ifa_addr->sa_family)
684                 continue;
685             ifaaddr = ifa->ifa_addr;
686             netmask = ifa->ifa_netmask;
687             brdaddr = ifa->ifa_dstaddr;
688             len = rt_msg2(RTM_NEWADDR, &info, 0, w);
689             if (w->w_where && w->w_tmem) {
690                 struct ifa_msghdr *ifam;

691                 ifam = (struct ifa_msghdr *) w->w_tmem;
692                 ifam->ifam_index = ifa->ifa_ifp->if_index;
693                 ifam->ifam_flags = ifa->ifa_flags;
694                 ifam->ifam_metric = ifa->ifa_metric;
695                 ifam->ifam_addrs = info.rti_addrs;
696                 if (error = copyout(w->w_tmem, w->w_where, len))
697                     return (error);
698                 w->w_where += len;
699             }
700         }
701         ifaaddr = netmask = brdaddr = 0;
702     }
703     return (0);
704 }

```

rtsock.c

图19-40 sysctl\_iflist 函数：返回接口列表及其地址

#### 4. 循环处理每一个地址结构，检测其地址族

682-684 处理接口的每一个 ifaddr 结构。进程也可以指定一个非零地址族 (图19-36中的

mib[3])来选择仅处理那些指定族的接口地址。

#### 5. 创建选路消息

685-688 rt\_msg2创建的每个RTM\_NEWADDR消息中最多可以返回三个插口地址结构：ifaddr、netmask和brdaddr。

#### 6. 复制消息回传给进程

689-699 如果进程需要返回消息，则填入ifa\_msghdr结构的其余部分，用copyout给进程复制缓存，并增大w\_where。

701 因为info数组还要在下一个接口消息中使用，所以程序将其中的这三个指针设置成0。

### 19.17 小结

所有选路消息的格式都是相同的——一个定长的结构，后面跟着若干个插口地址结构。共有三种不同类型的消息，各自具有相应的定长结构，每种定长结构的前三个元素都分别标识消息的长度、版本和类型。每种结构中的比特掩码指定哪些插口地址结构跟在定长结构之后。

这些消息以两种方式在内核与进程之间传递。消息可以在任意一个方向上传递，并且都是通过选路插口每次读或写一个消息。这就使得一个超级用户进程对内核路由表的访问具有完全的读写能力。选路守护进程(如routed和gated)就是这样实现其期望的选路策略的。

另外，任何一个进程都可以用sysctl系统调用来读取内核路由表的内容。这种方法不需要涉及选路插口，也不需要特别的权限。最终的结果通常包含许多选路消息，该结果作为系统调用的一部分被返回。由于进程不知道结果的大小，因此，为系统调用提供了一种方法来返回结果的大小而不返回结果本身。

### 习题

19.1 RTF\_DYNAMIC和RTF\_MODIFIED标志之间有什么区别？对于一个给定的路由表项，它们可以同时设置吗？

19.2 当用下列命令添加默认路由时会有什么情况发生？

```
bsdi $ route add default -cloning -genmask 255.255.255.255 sun
```

19.3 某路由表包含了15个ARP表项和20个路由，试估算用sysctl导出该路由表时需要多少空间。