第18章 Radix树路由表

18.1 引言

由IP完成的路由选择是一种选路机制，它通过搜索路由表来确定从哪个接口把分组发送出去。它与选路策略(routing policy)不一样，选路策略是一组规则的集合，这些规则用来确定哪些路由可以编入到路由表中。 Net/3内核实现选路机制，而选路守护进程，典型地如routed或gated，实现选路策略。由于分组转发是频繁发生的 (一个繁忙的系统每秒要转发成百上千个分组)，相对而言，选路策略的变化要少些，因此路由表的结构必须能够适应这种情况。

关于路由选择的详细情况，我们分三章进行讨论：

- 本章将讨论Net/3分组转发代码所使用的Radix树路由表的结构。每次发送或转发分组时，IP都将查看该表(发送时分组需要查看该表，是因为IP必须决定哪个本地接口将接收该分组)。
- 第19章着重讨论内核与Radix树之间的接口函数以及内核与选路进程 (通常指实现选路策略的选路守护进程)之间交换的选路消息。进程可以通过这些消息来修改内核的路由表(添加路由、删除路由等)，并且当发生了一个异步事件，可能影响到路由策略 (如收到重定向，接口断开等)时，内核也通过这些消息来通知守护进程。
- 第20章给出了内核与进程之间交换选路消息时使用的选路插口。

18.2 路由表结构

在讨论Net/3路由表的内部结构之前，我们需要了解一下路由表中包含的信息类型。图 18-1是图1-17(作者以太网中的四个系统)的下半部分。

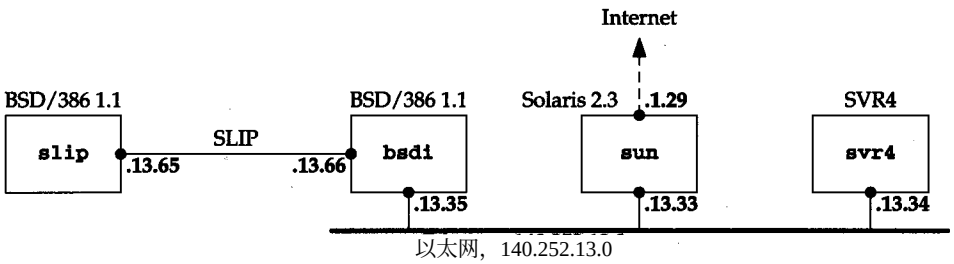

图18-1 路由表例子中使用子网

图18-2给出了图18-1中bsdi上的路由表。

为了能够更容易地看出每个表项中所设置的标志，我们已经对 netstat输出的“ Flags ”列进行了修改。

该表中的路由是按照下列过程添加的。其中，第 1、3、5、8和第9步是在系统的初始化阶段执行/etc/netstart shell脚本时完成的。

```
bsdi $ netstat -rn
Routing tables

Internet:
Destination       Gateway            Flags      Refs      Use   Interface
default           140.252.13.33      UG S          0        3   le0
127               127.0.0.1          UG S   R      0        2   lo0
127.0.0.1         127.0.0.1          U H            1       55   lo0
128.32.33.5       140.252.13.33      UGHS           2       16   le0
140.252.13.32     link#1             U    C         0        0   le0
140.252.13.33     8:0:20:3:f6:42     U H  L        11    55146   le0
140.252.13.34     0:0:c0:c2:9b:26    U H  L         0        3   le0
140.252.13.35     0:0:c0:6f:2d:40    U H  L         1       12   lo0
140.252.13.65     140.252.13.66      U H            0       41   sl0
224               link#1             U    C         0        0   le0
224.0.0.1         link#1             U H  L         0        5   le0
```

图18-2 主机bsdi 上的路由表

1) 默认路由是由route命令添加的。该路由通往主机sun(140 252 13 33)，主机sun拥有一条到Internet的PPP链路。

2) 到网络127的路由表项通常是由选路守护进程 (如gated)创建的，也可以通过/etc/netstart文件中的route命令将其添加到路由表中。该表项使得所有发往该网络的分组都将被环回驱动器 (图5-27)拒绝，但发往主机127.0.0.1的分组除外，因为对于该类分组，在下一步中添加的一条更特殊的路由将屏蔽本路由表项的作用。

3) 到环回接口(127.0.0.1)的表项是由ifconfig命令配置的。

4) 到vangogh.cs.berkeley.edu(128.32.33.5)的表项是用route命令手工创建的。该路由指定的路由器与默认路由所指定的相同 (都是140.252.13.33)。但是在拥有一条替代默认路由的通往特定主机的路由之后，我们就能把路由度量存储在该路由表项中。这些度量能够可以由管理者选择设置。每次 TCP建立一条到达目的主机的连接时都使用该度量，并且在连接关闭时，由TCP对其进行更新。我们将在图27-3中详细描述这些度量。

5) 接口le0的初始化是由 ifconfig命令完成的。该命令会在路由表中增加一条到140.252.13.32网络的表项。

6) 到以太网上另两台主机sun(140.252.13.33)和svr4(140.252.13.34)的路由表项是由ARP创建的，见第21章。它们都是临时路由，经过一段时间后，如果还未被使用，它们就会被自动删除。

7) 到本机(140.252.13.35)的表项是在第一次引用本机IP地址时创建的。该接口是一个环回，也就是说，任何发往本机IP地址的数据报将从内部反送回来。 4.4BSD中包含了自动创建该路由的新功能，见第21.13节。

8) 到主机140.252.13.65的表项是在ifconfig配置SLIP接口时创建的。

9) 通过以太网接口到达网络244的路由是由route命令添加的。

10) 到多播组224.0.0.1(所有主机的组， all-host group)的表项是Ping程序在连接224.0.0.1即“ Ping 224.0.0.1 ”时创建的。它也是一条临时路由，如果在一段时间内未被使用，就会被自动删除。

图18-2中的“ Flags ”列需要简单地说明一下。图18-25列出了所有可能的标志。

U 该路由存在。

G 该路由通向一个网关 (路由器)。这种路由被称为间接路由。如果没有设置本标志，则表明路由的目的地与本机直接相连，称为直接路由。

H 该路由通往一台主机，也就是说，目的地址是一个完整的主机地址。如果没有设置本标志，则路由通往一个网络，目的地址是一个网络地址：一个网络号，或一个网络号与子网号的组合。`netstat`命令并不区分这一点，但每一条网络路由中都包含一个网络掩码，而主机路由中则隐含了一个全1的掩码。

S 该路由是静态的。图18-2中`route`命令创建的三个路由表项是静态的。

C 该路由可被克隆 (clone)以产生新的路由。在本路由表中有两条路由设置了这个标志：一条是到本地以太网 (140.252.13.32)的路由，ARP通过克隆该路由创建到以太网中其他特定主机的路由；另一条是到多播组 224的路由，克隆该路由可以创建到特定多播组(如224.0.0.1)的路由。

L 该路由含有链路层地址。本标志应用于单播地址和多播地址。由 ARP从以太网路由克隆而得到的所有主机路由都设置了本标志。

R 环回驱动器(为设有本标志的路由而设计的普通接口)将拒绝所有使用该路由的数据报。

添加带有拒绝标志的路由的功能由 NET/2提供。它提供了一种简单的方法，来防止主机向外发送以网络127为目的地的数据报。参见习题6.6。

在4.3BSD Reno之前，内核将为IP地址维护两个不同的路由表：一个针对主机路由，另一个针对网络路由。对于给定的路由，将根据它的类型添加到相应的路由表中。默认路由被存储在网络路由表中，其目的地址是 0.0.0.0。查找过程隐含了这样一种层次关系：首先查看主机路由表；如果找不到，则查找网络路由表；如果仍找不到，则查找默认路由。仅当三次查找都失败时，才认为目的地不可达。 [Leffler et al. 1998]的第11.5节描述了一种带链表结构的hash表，该hash表同时用于Net/1中的主机路由表和网络路由表。

4.3BSD Reno [Sklower 1991]的变化主要与路由表的内部表示有关。这些变化允许相同的路由表函数访问不同协议栈的路由表，如 OSI协议，它的地址是变长的，这一点与长度固定为32位的IP地址不同。为了提高查询速度，路由表的内部结构也做了变动。

Net/3路由表采用Patricia树结构[Sedgewick 1990]来表示主机地址和网络地址 (Patricia 支持“从文字数字的编码中提取信息的 Patricia算法”)。待查找的地址和树中的地址都被看成比特序列。这样就可以用相同的函数来查找和维护不同类型的树，如：含有 32 bit定长IP地址的树、含有48 bit定长XNS地址的树以及一棵含有变长OSI地址的树。

使用Patricia树构造路由表的思想应归功于 Van Jacobson 的[Sklower 1991]。

举个例子就可以很容易地描述出这个算法。查找路由表的目标就是为了找到一个最能匹配给定目标的特定地址。我们称这个给定的目标为查找键 (search key)。所谓最能匹配的地址，也就是说，一个能够匹配的主机地址要优于一个能够匹配的网络地址；而一个能够匹配的网络地址要优于默认地址。

每条路由表项都有一个对应的网络掩码，尽管在主机路由中没有存储掩码，但它隐含了一个全1比特的掩码。我们对查找键和路由表项的掩码进行逻辑与运算，如果得到的值与该路由表项的目的地址相同，则称该路由表项是匹配的。对于某个给定的查找键，可能会从路由表中找到多条这样的匹配路由，所以在单个表同时包含网络路由和主机路由的情况下，我们

必须有效地组织该表，使得总能先找到那个更能匹配的路由。

让我们来讨论图18-3给出的例子。图中给出了两个查找键，分别是 127.0.0.1和127.0.0.2。为了更容易地说明逻辑与运算，图中同时给出了它们的十六进制值。图中给出的两个路由表项分别是主机路由127.0.0.1(它隐含了一个全1的掩码0xffffffff)和网络路由127.0.0.0(它的掩码是0xff000000)。

		查找键=127.0.0.1		查找键=127.0.0.2	
		主机路由	**网络路由**	**主机路由**	**网络路由**
1	查找键	7f000001	7f000001	7f000002	7f000002
2	路由表键	7f000001	7f000001	7f000001	7f000000
3	路由表掩码	ffffffff	ff000000	ffffffff	ff000000
4	1和3的逻辑与	7f000001	7f000000	7f000002	7f000000
	2和4相等？	相等	相等	不等	相等

图18-3 分别以127.0.0.1和127.0.0.2为查找键的路由表查找示例

其中两个路由表项都能够匹配查找键 127.0.0.1，这时路由表的结构必须确保能够先找到更能匹配该查找键的表项(127.0.0.1)。

图18-4给出了对应于图18-2的Net/3路由表的内部表示。执行带-A标志的netstat命令可以导出路由表的树型结构，图18-4就是根据导出的结果而建立的。

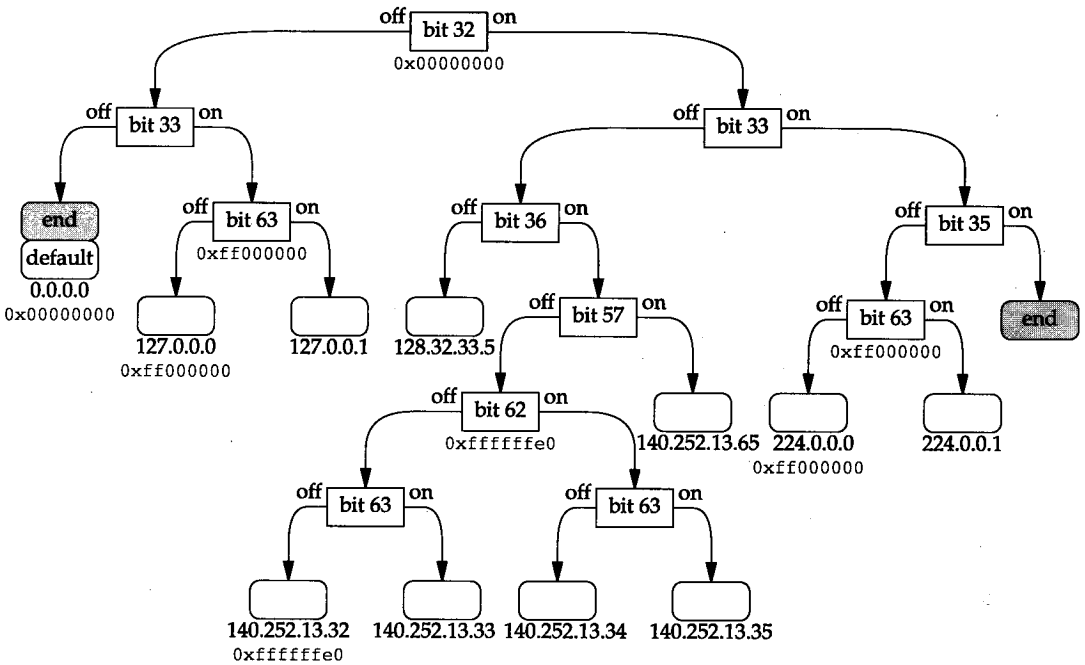

图18-4 对应于图18-2的Net/3路由表

标有“ end ”的两个阴影框是该树结构中带有特殊标志的叶结点，该标志代表树的端点。左边的那个拥有一个全0键，而右边的拥有一个全1键。左边的两个标有“ end ”和“ default ”的框垒在一起，这两个框有特殊的意义，它们与重复键有关，具体内容可参考 18.9节。

方角框被称为内部结点 (internal node)或简称为结点(node)，圆角框被称为叶子。每一个

内部结点对应于测试查找键的一个比特位，其左右各有一个分枝。每一个叶子对应于一个主机地址或者对应于一个网络地址。如果在叶子下面有一个十六进制数，那么这个叶子就对应于一个网络地址，该十六进制数就是叶子的网络掩码。如果在叶子下面没有十六进制的掩码，那么这个叶子就是一个主机地址，其隐含的掩码是0xffffffff。

有一些内部结点也含有网络掩码，在后面的学习中，我们将会了解这些掩码在回溯过程中是如何使用的。图中的每一个结点还包含了一个指向其父结点的指针 (没有在图中表示出来)，它能使树结构的回溯、删除及非递归操作更加方便。

比特比较是运用在插口地址结构上的，因此，在图 18-4中给出的比特位置是从插口地址结构中的起始位置开始算的。图 18-5给出了sockaddr_in结构中的比特位置。

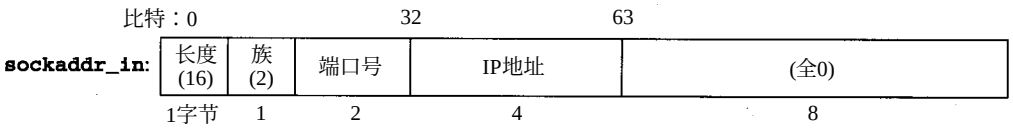

图18-5 Internet插口地址结构的比特位置

IP地址的最高位比特是比特 32，最低位是比特 63。此外还列出了长度是 16，地址族为2(AF_INET)，这两个数值在我们所列举的例子中将会遇到。

为了解释这些例子，还需要给出树中不同IP地址的比特表示形式。它们都被列在图18-6中，该图还给出了下面例子中要用到的一些其他 IP地址的比特表示形式。该图采用了加粗的字体来表示图18-4中分支点所对应的比特位置。

现在我们举一些特定的例子来说明路由表的查找过程是如何完成的。

1. 与主机地址匹配的例子

假定主机地址127.0.0.1是查找键 —— 待查找的目的地址。比特 32为0，因此，沿树顶点向左分支继续查找，到下一个结点。比特 33为1，因此，从该结点右分支继续查找，到下一个结点。比特63为1，因此，从右分支继续查找，到下一个结点。而下一个结点是个叶子，此时查找键(127.0.0.1)与叶子中的地址 (127.0.0.1)相比较。它们完全匹配，这样查找函数就可以返回该路由表项。

	32 bit IP地址								点分割表示
比特位置	**3333 2345**	**3333 6789**	**4444 0123**	**4444 4567**	**4455 8901**	**5555 2345**	**5555 6789**	**6666 0123**	
	0000	1010	0000	0001	0000	0010	0000	0011	**10.1.2.3**
	0111	0000	0000	0000	0000	0000	0000	0001	**112.0.0.1**
	0111	1111	0000	0000	0000	0000	0000	0000	**127.0.0.0**
	0111	1111	0000	0000	0000	0000	0000	0001	**127.0.0.1**
	0111	1111	0000	0000	0000	0000	0000	0011	**127.0.0.3**
	1000	0000	0010	0000	0010	0001	0000	0101	**128.32.33.5**
	1000	0000	0010	0000	0010	0001	0000	0110	**128.32.33.6**
	1000	1100	1111	1100	0000	1101	0010	0000	**140.252.13.32**
	1000	1100	1111	1100	0000	1101	0010	0001	**140.252.13.33**
	1000	1100	1111	1100	0000	1101	0010	0010	**140.252.13.34**
	1000	1100	1111	1100	0000	1101	0010	0011	**140.252.13.35**
	1000	1100	1111	1100	0000	1101	0100	0001	**140.252.13.65**
	1110	0000	0000	0000	0000	0000	0000	0000	**224.0.0.0**
	1110	0000	0000	0000	0000	0000	0000	0001	**224.0.0.1**

图18-6 图18-2和图18-4中IP地址的比特表示形式

2. 与主机地址匹配的例子

再假定查找键是地址140.252.13.35。比特32为1，因此，沿树顶点向右分支继续查找。比特33为0，比特36为1，比特57为0，比特62为1，比特63为1，因此，查找在底部标有140.252.13.35的叶子处终止。查找键与路由表键完全匹配。

3. 与网络地址匹配的例子

假定查找键是127.0.0.2。比特32为0，比特33为1，比特63为0，因此，查找在标有127.0.0.0的叶子处终止。查找键和路由表键并没有完全匹配，因此，需要进一步看它是不是一个能够匹配的网络地址。对查找键和网络掩码(0xff000000)进行逻辑与运算，得到的结果与该路由表键相同，即认为该路由表项能够匹配。

4. 与默认地址匹配的例子

假定查找键是10.1.2.3。比特32为0，比特33为0，因此，查找在标有“end”和“default”并带有重复键的叶子处终止。在这两个叶子中重复的路由表键是0.0.0.0。查找键与路由表键值没有完全匹配，因此，需要进一步看它是不是一个能够匹配的网络地址。这种匹配运算要对每个含网络掩码的重复键都试一遍。第一个键(标有end)没有网络掩码，可以跳过不查。第二个键(默认表项)有一个0x00000000的掩码。查找键和这个掩码进行逻辑与运算，所得结果和路由表键(0)相等，即认为该路由表项能够匹配。这样默认路由就被用做匹配路由。

5. 带回溯过程的与网络地址匹配的例子

假定查找键是127.0.0.3。比特32为0，比特33为1，比特63为1，因此，查找在标有127.0.0.1的叶子处终止。查找键和路由表键没有完全匹配。由于这个叶子没有网络掩码，无法进行网络掩码匹配的尝试。此时就要进行回溯。

回溯算法在树中向上移动，每次移动一层。如果遇到的内部结点含有掩码，则对查找关键字和该掩码进行逻辑与运算，得到一个键值，然后以这个键值作为新的查找键，在含该掩码的内部结点为开始的子树中进行另一次查找，看是否能找到匹配的结点。如果找不到，则回溯过程继续沿树上移，直到到达树的顶点。

在这个例子中，查找上移一层到达比特63对应的结点，该结点含有一个掩码。于是对查找键和掩码(0xff000000)进行逻辑与运算，得到一个新的查找键，其值为127.0.0.0。然后从该结点开始查找127.0.0.0。比特63为0，因此，沿左分支到达标有127.0.0.0的叶子上。用新的查找键与路由表键相比较，它们是相等的，因此认为这个叶子是匹配的。

6. 多层回溯的例子

假定查找键是112.0.0.1。比特32为0，比特33为1，比特63为1，因此，查找在标有127.0.0.1的叶子处终止。该键与查找键不相等，并且路由表项中没有网络掩码，因此需要进行回溯。

查找过程向上移动一层，到达比特63对应的结点，该结点含有一个掩码。对查找关键字和该掩码(0xff000000)进行逻辑与运算，然后再从这个结点开始进一步查找。在新的查找键中比特63为0，因此，沿左分支到达标有127.0.0.0的叶子。比较之后发现逻辑与运算得到的查找键(112.0.0.0)和路由表键并不相等。

因此继续向上回溯一层，从比特63对应的结点上移到比特33对应的结点。但这个结点没有掩码，再继续向上回溯。到达的下一层是树的顶点(比特32)，它有一个掩码。对查找键(112.0.0.1)和该掩码(0x00000000)进行逻辑与运算后，从该点开始一个新的查找。在新的查找

键中，比特32为0，比特33也为0，因此，查找在标有“ end ”和“ default ”的叶子处结束。通过与重复键列表中的每一项进行比较，发现默认键与新的查找键相匹配，因此采用默认路由。

从这个例子中可以知道，如果在路由表中存在默认路由，那么当回溯最终到达树的顶点时，它的掩码为全0比特，这使得查找向树中最左边叶子的方向进行搜索，最终与默认路由相匹配。

7. 带回溯和克隆过程、并与主机地址相匹配的例子

假定查找键是224.0.0.5。比特32为1，比特331，比特35为0，比特63为1，因此，查找在标有224.0.0.1的叶子处结束。路由表的键值和查找关键字并不相等，并且该路由表项不包含网络掩码，因此要进行回溯。

回溯向上移动一层，到达比特 63对应的结点。这个结点含有掩码 `0xff000000`，因此，对查找键和该掩码进行逻辑与运算，产生一个新的查找键，即 224.0.0.0。再从这个结点开始一次新的查找。在新的查找键中比特 63为0，于是沿左分支到达标有224.0.0.0的叶子。这个路由表键和逻辑与运算得到的查找键相匹配，因此这个路由表项是匹配的。

该路由上设置了“ 克隆 ”标志 (见图18-2)，因此，以224.0.0.5为地址创建一个新的叶子。新的路由表项是：

```
Destination    Gateway      Flags      Refs      Use    Interface
224.0.0.5      link#1       UHL         0         0        le0
```

图18-7从比特35对应的结点开始，给出了图 18-4中路由表树右边部分的新的排列。注意，无论何时向树中添加新的叶子，都需要两个结点：一个作为叶子，另一个作为测试某一位比特的内部结点。

新创建的表项就被返回给查找224.0.0.5的调用者。

8. 大图

图18-8是一张比较大的图，它描述了所有涉及到的数据结构。该图的底部来自于图 3-32。

现在我们将解释图中的几个要点，在后面，本章还将给出详细的阐述。

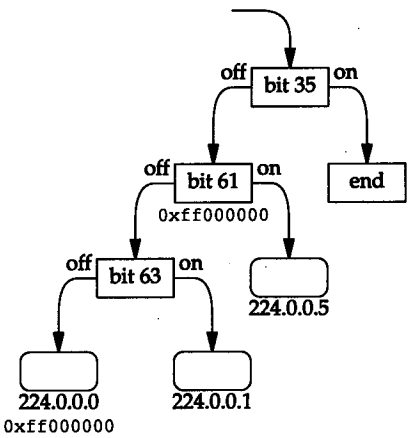

图18-7 插入224.0.0.5路由表项后图18-6的改动

- `rt_tables`是指向`radix_node_head`结构的指针数组。每一个地址族都有一个数组单元与之对应。`rt_tables[AF_INET]`指向Internet路由表树的顶点。
- `radix_node_head`结构包含三个`radix_node`结构。这三个结构是在初始化路由树时创建的，中间的是树的顶点。它对应于图 18-4中最上端标有“ bit 32 ”的结点框。三个`radix_node`结构中的第一个是图18-4中最左边的叶子(与默认路由共享的重复)，第三个结构是最右边的叶子。在一个空的路由表中，就只包含这三个 `radix_node`结构，我们将会看到`rn_inithead`函数是如何构建它们的。
- 全局变量`mask_rnhead`也指向一个`radix_node_head`结构。它是包含了所有掩码的一棵独立树的首部结构。观察图 18-4中给出的八个掩码可知，有一个掩码重复了四次，有两个掩码重复了一次。通过把掩码放在一棵单独的树中，可以做到对每一个掩码只需要维护它的一个备份即可。

• 路由表树是用 rtentry结构创建的，在图 18-8中，有两个 rtentry结构。每一个 rtentry结构包含两个radix_node结构，因为每次向树中插入一个新的路由时，都需要两个结点：一个是内部结点，对应于某一位测试比特；另一个是叶子，对应于一个主机路由或一个网络路由。在每一个 rtentry结构中，给出了内部结点对应的要测试的那位比特以及叶子中所包含的地址。

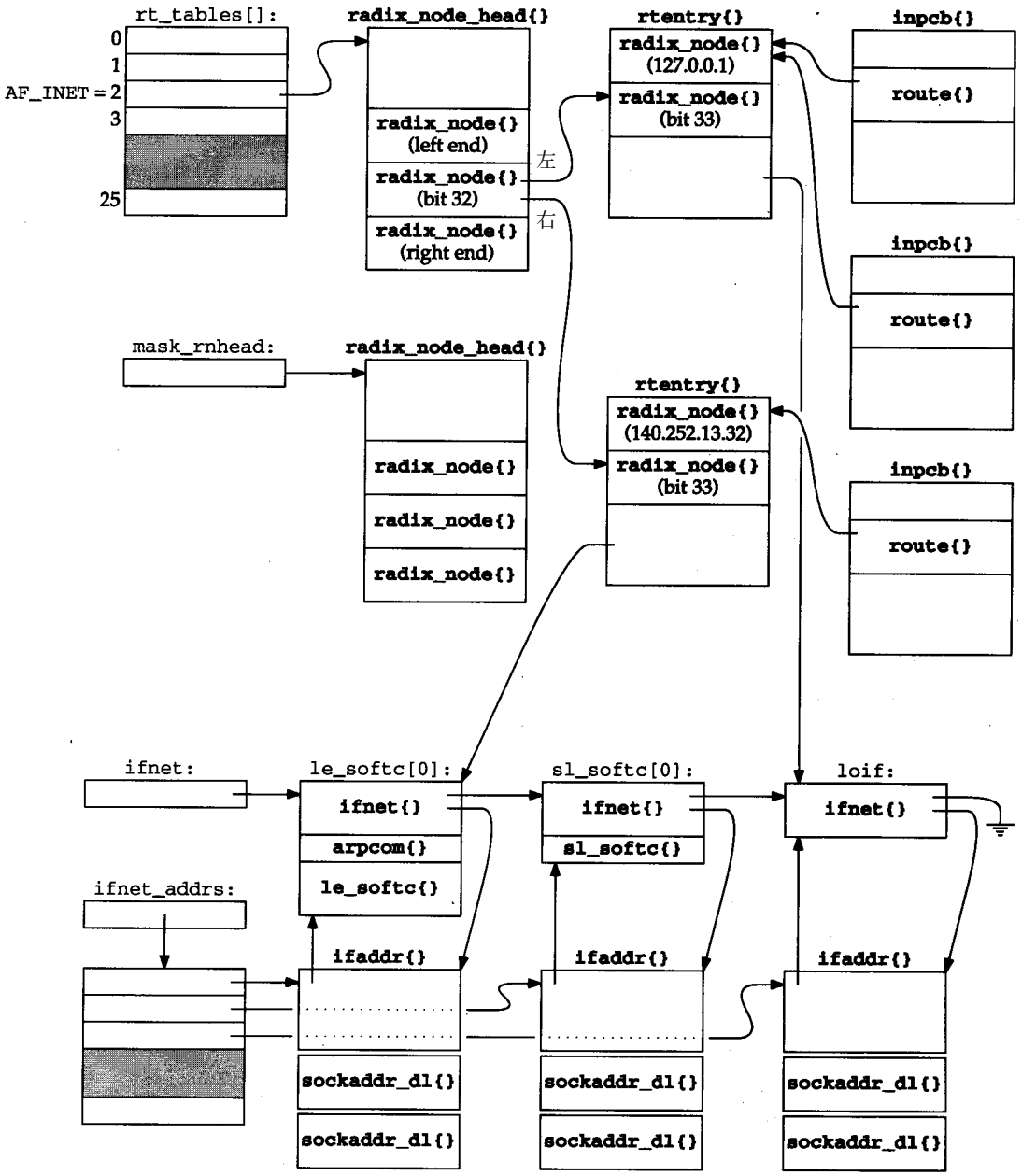

图18-8 路由表中涉及的数据结构

rtentry结构中的其余部分是该路由的一些其他重要信息。虽然我们只给出了该结构中的一个指向ifnet结构的指针，但在这个结构中还包含了指向 ifaddr结构的指针、该路由的

标志、指向另一个rtentry结构的指针(如果该路由是一个非直接路由)和该路由的度量等等。

- 存在于每一个UDP和TCP插口(图22-1)中的协议控制块PCB(见第22章)中包含了一个指向rtentry结构的route结构。每次发送一个IP数据报时，UDP和TCP输出函数都传递一个指向PCB中route结构的指针，作为调用ip_output的第三个参数。使用相同路由的PCB都指向相同的路由表项。

18.3 选路插口

在4.3BSD Reno的路由表做了变动后，路由子系统和进程间的交互过程也要做出变动，这就引出了选路插口(routing socket)的概念。在4.3BSD Reno之前，是由进程(如route命令)通过发出定长的ioctl来修改路由表的。4.3BSD Reno采用新的PF_ROUTE域把它改变成一种更为通用的消息传递模式。进程在PF_ROUTE域中创建一个原始插口(raw socket)，就能够向内核发送选路消息，以及从内核接收选路消息(如重定向或来自于内核的其他的异步通知)。

图18-9给出了12种不同类型的选路消息。消息类型位于rt_msghdr结构(图19-16)中的rtm_type字段。进程只能发送其中的5种消息(写入到选路插口中)，但可以接收全部的12种消息。

我们将在第19章给出这些选路消息的详细讨论。

rtm_type	发往内核？	从内核发出？	描　述	结构类型
RTM_ADD	•	•	添加路由	rt-msghdr
RTM_CHANGE	•	•	改变网关、度量或标志	rt-msghdr
RTM_DELADDR		•	从接口中删除地址	ifa-msghdr
RTM_DELETE	•	•	删除路由	rt-msghdr
RTM_GET	•	•	报告度量及其他路由信息	rt-msghdr
RTM_IFINFO		•	接口打开或关闭等	rt-msghdr
RTM_LOCK	•	•	锁定指明的度量	rt-msghdr
RTM_LOSING		•	内核怀疑某路由无效	rt-msghdr
RTM_MISS		•	查找这个地址失败	rt-msghdr
RTM_NEWADDR		•	接口中添加了地址	ifa-msghdr
RTM_REDIRECT		•	内核得知要使用不同的路由	rt-msghdr
RTM_RESOLVE		•	请求将目的地址解析成链路层地址	rt-msghdr

图18-9　通过选路插口交换的消息类型

18.4 代码介绍

路由选择中使用的各种结构和函数是通过五个C文件和三个头文件来定义的。图18-10列出了这些文件。

通常，前缀rn_表示radix结点函数，这些函数可以对Patricia树进行查找和操作，前缀raw_表示路由控制块函数，rout_、rt_和rt这三个前缀表示常用的选路函数。

尽管有的文件和函数以raw为前缀，但在所有的选路章节中我们仍使用术语选路控制块(routing control block)而不是原始控制块。这是为了防止与第32章中讨论的原始IP控制块及其函数相混淆。虽然原始控制块及相关函数不仅仅用于Net/3中的选路插口(使用这些结构和函数的原始OSI协议之一)，但是本书中我们只用做PF_ROUTE域中的选路插口。

文　件	描　述
net/radix.h	radix结点定义
net/raw_cb.h	选路控制块定义
net/route.h	选路结构
net/radix.c	radix结点(Patricia树)函数
net/raw_cb.c	选路控制块函数
net/raw_usrreq.c	选路控制块函数
net/route.c	选路函数
net/rtsock.c	选路插口函数

图18-10　本章中讨论的文件

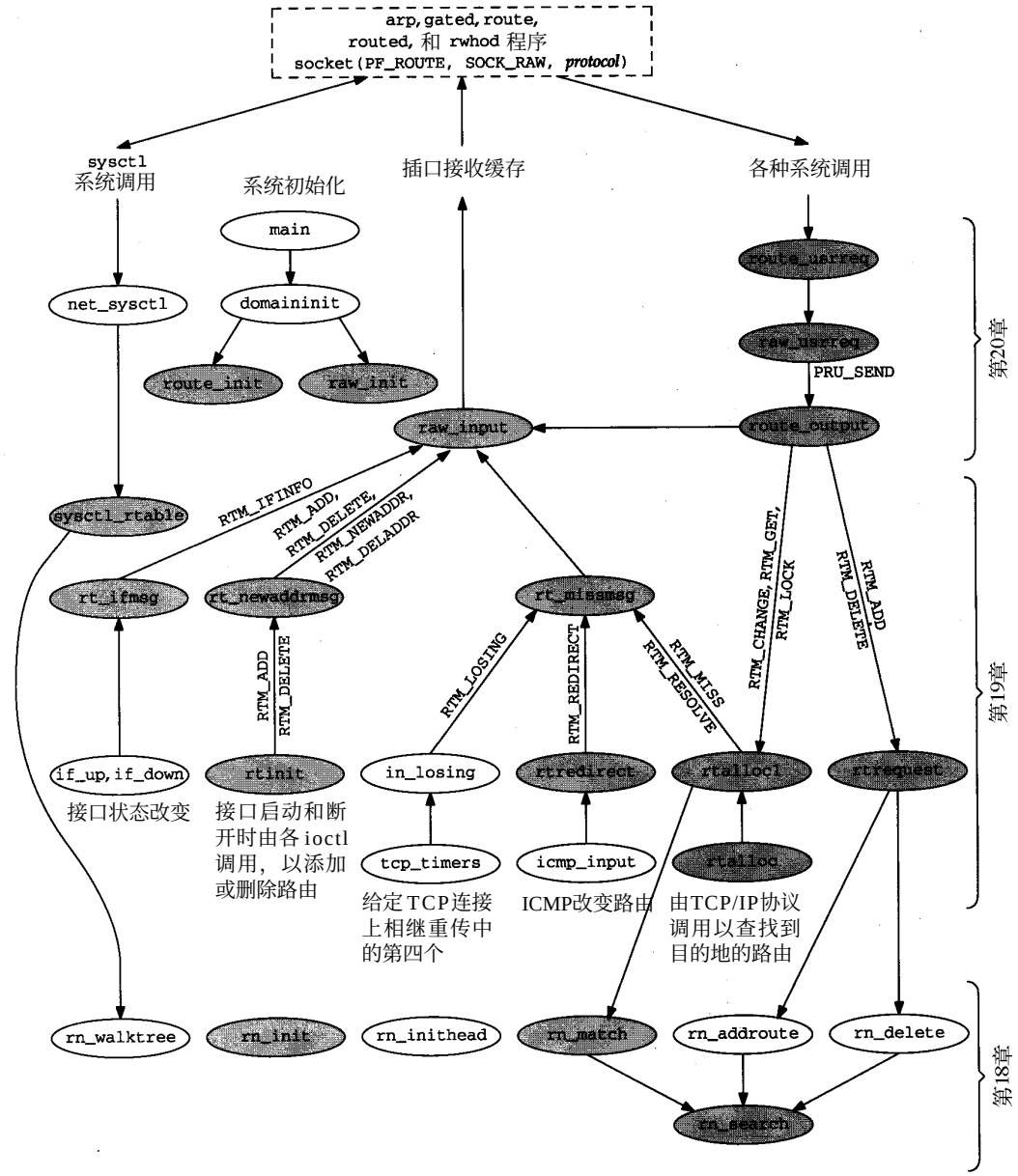

图18-11　各选路函数之间的关系

图18-11给出了一些基本的选路函数，并表示了它们之间的相互关系。其中带阴影的椭圆是在本章和下面两章中要涉及的内容。在图中，我们还给出了每种类型的选路消息 (共12种)的产生之处。

rtalloc函数是由 Internet协议调用的，用于查找到达指定目的地的路由。在ip_rtaddr、ip_forward、ip_output和ip_setmoptions函数中都已出现过rtalloc，在后面介绍的in_pcbconnect和tcp_mss函数中也将会遇到它。

图18-11还给出了在选路域中创建插口的五个典型程序：

- arp处理ARP高速缓存，该ARP高速缓存被存储在Net/3的IP路由表中(见第21章)；
- gated和routed是选路守护进程，它们与其他路由器进行通信。当选路环境发生变化时(指路由器及链路断开或连通)，对内核的路由表进行操作；
- route通常是由启动脚本或系统管理员执行的一个程序，用于添加或删除路由；
- rwhod在启动时会调用一个选路sysctl来测定连接的接口。

当然，任何进程(具有超级用户的权限)都能打开一个选路插口向选路子系统发送或从中接收消息；在图18-11中，我们只给出了一些常用的系统程序。

18.4.1 全局变量

图18-12列出了在三个有关路由选择的章节中介绍的全局变量。

变　量	数据类型	描　述
rt_tables	struct radix_node_head *[]	路由表表头指针的数组
mask_rnhead	struct radix_node_head *	指向掩码表表头的指针
rn_mkfreelist	struct radix_mask *	可用radix_mask结构的链表表头
max_keylen	int	以字节为单位的路由表键值的最大长度
rn_zeros	char *	长为max_keylen、值为全零比特的数组
rn_ones	char *	长为max_keylen、值为全1比特的数组
maskedKey	char *	长为max_keylen、掩码过的查找键数组
rtstat	struct rtstat	路由选择统计(图18-13)
rttrash	int	未释放的非表中路由的数目
rawcb	struct rawcb	选路控制块双向链表表头
raw_recvspace	u_long	选路插口接收缓冲区的默认大小，8192字节
raw_sendspace	u_long	选路插口发送缓冲区的默认大小，8192字节
route_cb	struct route_cb	选路插口监听器的数目，每个协议的数目及总的数目
route_dst	struct sockaddr	保存选路消息中目的地址的临时变量
route_src	struct sockaddr	保存选路消息中源地址的临时变量
route_proto	struct sockproto	保存选路消息中协议的临时变量

图18-12　在三个有关选路的章节中介绍的全局变量

18.4.2 统计量

图18-13列举了一些路由选择统计量，它们是在全局结构 rtstat中维护的。

在代码的处理中，我们可以发现计数器是怎样增加的。这些计数器在 SNMP中并未使用。

图18-14给出了netstat -rs命令输出的一些统计数据的样例，该命令用于显示rtstat结构。

rtstat成员	描　述	在SNMP中的使用
rts_badredirect	无效重定向调用的数目	
rts_dynamic	由重定向创建的路由数目	
rts_newgateway	由重定向修改的路由数目	
rts_unreach	查找失败的次数	
rts_wildcard	由通配符匹配的查找次数(从未使用)	

图18-13　在rtstat 结构中维护的路由选择统计数据

netstat-rs的输出	rtstat 成员
1029 bad routing redirects	rts_badredirect
0 dynamically created routes	rts_dynamic
0 new gateways due to redirects	rts_newgateway
0 destinations found unreachable	rts_unreach
0 uses of a wildcard route	rts-wildcard

图18-14　路由选择统计数据样例

18.4.3　SNMP变量

图18-15给出了名为ipRouteTable的IP路由表以及相应的内核变量。

IP路由表，index = <*ipRouteDest*>		
SNMP变量	变　量	描　述
ipRouteDest	rt_key	IP目的地址。值为0.0.0.0时，代表默认路由
ipRouteIfIndex	rt_ifp, if_index	接口号：ifIndex
ipRouteMetric1	-1	基本的路由度量。其含义取决于选路协议的值 (ipRouteProto)。值为-1，表示没有使用
ipRouteMetric2	-1	可选的路由度量
ipRouteMetric3	-1	可选的路由度量
ipRouteMetric4	-1	可选的路由度量
ipRouteNextHop	rt_gateway	下一跳路由器的IP地址
ipRouteType	(见正文)	路由类型：1=其他，2=无效路由，3=直接的，4=间接的
ipRouteProto	(见正文)	路由协议：1=其他，4=ICMP重定向，8=RIP, 13=OSPF, 14= BGP 等
ipRouteAge	(未实现)	从路由最后一次被修改或被确定为正确时起的秒数
ipRouteMask	rt_mask	在和ipRouteDest比较前，与目的主机IP地址进行逻辑与运算的掩码
ipRouteMetric5	-1	可选的路由度量
ipRouteInfo	NULL	本选路协议特定的MIB定义的引用

图18-15　IP路由表：ipRouteTable

如果在rt_flags中将标志RTF_GATEWAY置位，则该路由就是远程的，ipRouteType等于4；否则该路由就是直达的，ipRouteType值为3。对于ipRouteProto，如果将标志RTF_DYNAMIC或RTF_MODIFIED置位，则该路由就是由ICMP来创建或修改的，值为4，否则为其他情况，其值为1。最后，如果rt_mask指针为空，则返回的掩码就是全1(即主机路由)。

18.5 Radix结点数据结构

在图18-8中可以发现每一个路由表的表头都是一个radix_node_head结构，而选路树中所有的结点(包括内部结点和叶子)都是radix_node结构。radix_node_head结构如图18-16所示。

radix.h

```
91 struct radix_node_head {
92     struct radix_node *rnh_treetop;
93     int     rnh_addrsize;        /* (not currently used) */
94     int     rnh_pktsize;         /* (not currently used) */
95     struct radix_node *(*rnh_addaddr)    /* add based on sockaddr */
96             (void *v, void *mask,
97              struct radix_node_head * head, struct radix_node nodes[]);
98     struct radix_node *(*rnh_addpkt)     /* add based on packet hdr */
99             (void *v, void *mask,
100             struct radix_node_head * head, struct radix_node nodes[]);
101    struct radix_node *(*rnh_deladdr)    /* remove based on sockaddr */
102            (void *v, void *mask, struct radix_node_head * head);
103    struct radix_node *(*rnh_delpkt)     /* remove based on packet hdr */
104            (void *v, void *mask, struct radix_node_head * head);
105    struct radix_node *(*rnh_matchaddr)      /* locate based on sockaddr */
106            (void *v, struct radix_node_head * head);
107    struct radix_node *(*rnh_matchpkt)   /* locate based on packet hdr */
108            (void *v, struct radix_node_head * head);
109    int     (*rnh_walktree)      /* traverse tree */
110            (struct radix_node_head * head, int (*f) (), void *w);

111    struct radix_node rnh_nodes[3];      /* top and end nodes */
112 };
```

radix.h

图18-16 radix_node_head 结构：每棵选路树的顶点

92 rnh_treetop指向路由树顶端的radix_node结构。可以看到radix_ node_head结构的最后一项分配了三个radix_node结构，其中中间的那个被初始化成树的顶点 (图18-8)。

93-94 rnh_addrsize和rnh_pktsize目前未被使用。

rnh_addrsize是为了能够方便地将路由表代码导入到系统中去，因为系统的插口地址结构中没有标识其长度的字节。rnh_pktsize是为了能够利用radix结点机制直接检查分组头结构中的地址，而无需把该地址拷贝到某个插口地址结构中去。

95-110 从rnh_addaddr到rnh_walktree是七个函数指针，它们所指向的函数将被调用以完成对树的操作。如图 18-17所示，rn_inithead仅初始化了其中的四个指针，剩下的三个指针在Net/3中未被使用。

成 员	被rn_inithead初始化为
rnh_addaddr	*rn_addroute*
rnh_addpkt	*NULL*
rnh_deladdr	*rn_delete*
rnh_delpkt	*NULL*
rnh_matchaddr	*rn_match*
rnh_matchpkt	*NULL*
rnh_walktree	*rn_walktree*

图18-17 在radix_node_head 结构中的七个函数指针

111-112 图18-18给出了组成树中结点的radix_node结构。在图18-8中，我们可以发现，在radix_node_head中分配了三个这样的radix_node结构，而在每一个rtentry结构中分配了两个radix_node结构。

41-45 前五个成员是内部结点和叶子都有的成员。后面是一个union：如果结点是叶子，那么它定义了三个成员；如果是内部结

点，那么它定义了另外不同的三个成员。由于在 Net/3代码中经常使用union中的这些成员，因此，用一组#define语句定义它们的简写形式。

41-42 rn_mklist是该结点掩码链表的表头。我们将在18.9节中描述该字段。rn_p指向该结点的父结点。

43 如果rn_b值大于或者等于零，那么该结点为内部结点；否则就是叶子。对于内部结点来说，rn_b就是要测试的比特位置：例如，在图 18-4中，树的顶结点的rn_b值为32。对于叶子来说，rn_b是负的，它的值等于- 1减去网络掩码索引(index of the network mask)。该索引是指掩码中出现的第一个零的比特位置。图 18-19给出了图18-4中掩码的索引。

radix.h

```
40 struct radix_node {
41     struct radix_mask *rn_mklist;   /* list of masks contained in subtree */
42     struct radix_node *rn_p;     /* parent pointer */
43     short    rn_b;               /* bit offset; -1-index(netmask) */
44     char     rn_bmask;           /* node: mask for bit test */
45     u_char   rn_flags;           /* Figure 18.20 */
46     union {
47         struct {                 /* leaf only data: rn_b < 0 */
48             caddr_t rn_Key;      /* object of search */
49             caddr_t rn_Mask;     /* netmask, if present */
50             struct radix_node *rn_Dupedkey;
51         } rn_leaf;
52         struct {                 /* node only data: rn_b >= 0 */
53             int     rn_Off;      /* where to start compare */
54             struct radix_node *rn_L;     /* left pointer */
55             struct radix_node *rn_R;     /* right pointer */
56         } rn_node;
57     } rn_u;
58 };

59 #define rn_dupedkey rn_u.rn_leaf.rn_Dupedkey
60 #define rn_key      rn_u.rn_leaf.rn_Key
61 #define rn_mask     rn_u.rn_leaf.rn_Mask
62 #define rn_off      rn_u.rn_node.rn_Off
63 #define rn_l        rn_u.rn_node.rn_L
64 #define rn_r        rn_u.rn_node.rn_R
```

radix.h

图18-18 radix_node 结构：路由树的结点

	32 bit IP掩码(比特32 - 36)								索引	rn_b
	3333 2345	3333 6789	4444 0123	4444 4567	4455 8901	5555 2345	5555 6789	6666 0123		
00000000:	0000	0000	0000	0000	0000	0000	0000	0000	0	–1
ff000000:	1111	1111	0000	0000	0000	0000	0000	0000	40	–41
fffffffe0:	1111	1111	1111	1111	1111	1111	1110	0000	59	–60

图18-19 掩码索引的例子

我们可以发现，其中的全0掩码的索引是特殊处理的：它的索引是 0，而不是32。

44 内部结点的rn_bmask是个单字节的掩码，用于检测相应的比特位是 0还是1。在叶子中它的值为0。很快我们将会看到如何运用成员 rn_bmask和成员rn_off。

45 图18-20给出了成员rn_flags的三个值。

常 量	描 述
RNF_ACTIVE	该结点是活的(alive)(针对rtfree)
RNF_NORMAL	该叶子含有正常路由(目前未被使用)
RNF_ROOT	该叶子是树的根叶子

图18-20 rn_flags 的值

RNF_ROOT标志只有在radix_node_head结构中的三个radix结点(树的顶结点、左端结点和右端结点)中才能设置。这三个结点不能从路由树中删除。

48-49 对于叶子而言，rn_key指向插口地址结构，rn_mask指向保存掩码的插口地址结构。如果rn_mask为空，则其掩码为隐含的全 1值(即，该路由指向某个主机而不是某个网络)。

图18-21例举了一个与图18-4中的叶子140.252.13.32相对应的radix_node结构的例子。

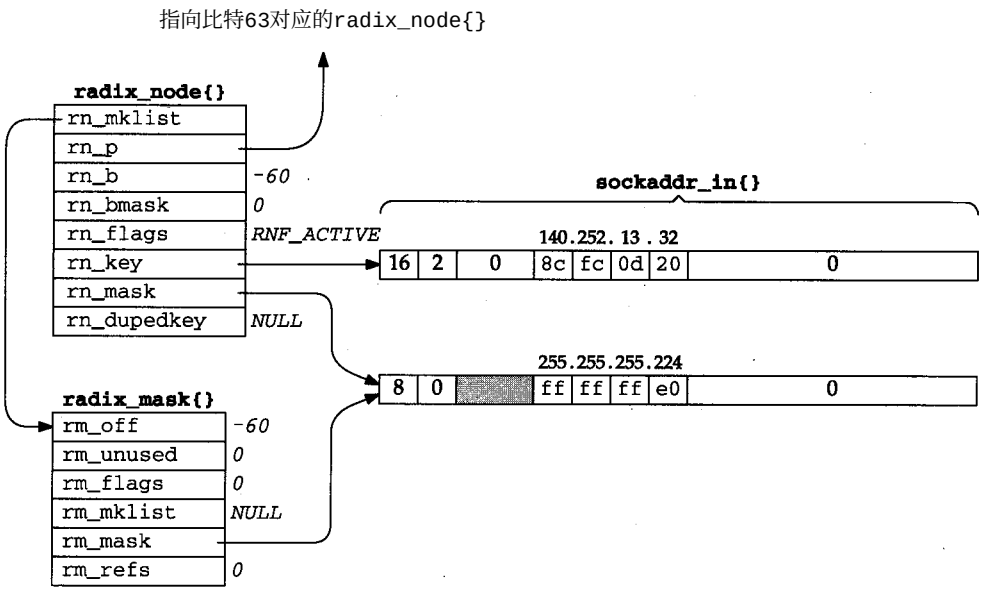

图18-21 与图18-4中的叶子140.252.13.32相对应的radix_node 结构

该例子中还给出了图18-22中描述的radix_mask结构。我们把它的宽度略微缩小了一些，以区分于radix_node结构；这两种结构在后面的很多图例中都会遇到。有关 radix_mask结构的作用将在18.9节中阐述。

值为-60的rn_b相对应的索引为59。rn_key指向一个sockaddr_in结构，它的长度值为16，地址族值为2(AF_INET)。由rn_mask和rm_mask指向的掩码结构所含的长度值为 8，地址族值为0(该族为AF_UNSPEC，但我们从未使用它)。

50-51 当有多个叶子的键值相同时，使用rn_dupedkey指针。具体内容将在18.9节中阐述。

52-58 有关rn_off的内容将在18.8节中阐述。rn_l和rn_r是该内部结点的左、右指针。

图18-22给出了radix_mask结构的定义。

76-83 该结构中包含了一个指向其掩码的指针：rm_mask，实际上是一个保存掩码的插口地址结构的指针。每一个 radix_node结构对应一个radix_mask结构的链表，这就允许每

个结点包含多个掩码：成员 rn_mklist指向链表的第一个结点，然后每个结构的成员rm_mklist指向链表的下一个结点。该结构的定义同时声名了全局变量 rn_mkfreelist，它是可用的radix_mask结构链表的表头。

radix.h

```
76 extern struct radix_mask {
77      short    rm_b;                  /* bit offset; -1-index(netmask) */
78      char     rm_unused;             /* cf. rn_bmask */
79      u_char   rm_flags;              /* cf. rn_flags */
80      struct radix_mask *rm_mklist;    /* more masks to try */
81      caddr_t rm_mask;                /* the mask */
82      int      rm_refs;               /* # of references to this struct */
83 }       *rn_mkfreelist;
```

radix.h

图18-22 radix_mask 结构

18.6 选路结构

访问内核路由信息的关键之处是：

1) rtalloc函数，用于查找通往目的地的路由；

2) route结构，它的值由rtalloc函数填写；

3) route结构所指向的rtentry结构。

图18-8给出了UDP和TCP(参见第22章)中使用的协议控制块(PCB)，其中包含一个route结构，见图18-23。

route.h

```
46 struct route {
47      struct rtentry *ro_rt;          /* pointer to struct with information */
48      struct sockaddr ro_dst;         /* destination of this route */
49 };
```

route.h

图18-23 route 结构

ro_dst被定义成一个一般的插口地址结构，但对于 Internet协议而言，它就是一个sockaddr_in结构。注意，对这种结构类型的绝大多数引用都是一个指针，而 ro_dst是该结构的一个实例而非指针。

这里，我们有必要回顾一下图8-24。从该图可以得知，每次发送IP数据报时，这些路由是如何使用的。

- 如果调用者传递了一个route结构的指针，那么就使用该结构。否则，就要用一个局部route结构，其值设置为0(设置ro_rt为空指针)。UDP和TCP把指向它们的PCB中route结构的指针传递给ip_output。
- 如果route结构指向一个rtentry结构(ro_rt指针为非空)，同时所引用的接口仍然有效；而且如果route结构中的目的地址与IP数据报中的目的地址相等，那么该路由就被使用。否则，目的主机的 IP地址将会设置在插口地址结构 so_dst中，并且调用rtalloc来查找一条通向该目的主机的路由。在TCP链接中，数据报的目的地址始终是路由的目的地址，不会发生变化，但是UDP应用可以通过sendto每次都把数据报发送到不同的目的地。
- 如果rtalloc返回的ro_rt是个空指针，则表明找不到路由，并且 ip_output返回一

个差错。

- 如果在rtentry结构中设有RTF_GATEWAY标志，那么该路由为非直接路由(参见图18-2中的G标志)。接口输出函数的目的地址(dst)就变成网关的IP地址，即rt_gateway成员，而不是IP数据报的目的地地址。

图18-24给出了rtentry结构的定义。

route.h

```
83 struct rtentry {
84     struct radix_node rt_nodes[2];  /* a leaf and an internal node */
85     struct sockaddr *rt_gateway;    /* value associated with rn_key */
86     short   rt_flags;           /* Figure 18.25 */
87     short   rt_refcnt;          /* #held references */
88     u_long  rt_use;             /* raw #packets sent */
89     struct ifnet *rt_ifp;       /* interface to use */
90     struct ifaddr *rt_ifa;      /* interface address to use */
91     struct sockaddr *rt_genmask;    /* for generation of cloned routes */
92     caddr_t rt_llinfo;          /* pointer to link level info cache */
93     struct rt_metrics rt_rmx;   /* metrics: Figure 18.26 */
94     struct rtentry *rt_gwroute; /* implied entry for gatewayed routes */
95 };
96 #define rt_key(r)    ((struct sockaddr *)((r)->rt_nodes->rn_key))
97 #define rt_mask(r)   ((struct sockaddr *)((r)->rt_nodes->rn_mask))
```

route.h

图18-24 rtentry 结构

83-84 在该结构中包含有两个radix_node结构。正如我们在图18-7的例子中所提到的，每次向路由树添加一个新叶子的同时也要添加一个新的内部结点。rt_nodes[0]为叶子，rt_nodes[1]为内部结点。在图18-24最后的两个#define语句提供了访问该叶结点的键和掩码的简写形式。

86 图18-25给出了储存在rt_flags中的各种常量以及在图18-2的“Flags”列中由netstat输出的相应字符。

常 量	netstat标志	描 述
RTF_BLACKHOLE		无差错的丢弃分组(环回驱动器:图5-27)
RTF_CLONING	C	使用中产生新的路由(由ARP使用)
RTF_DONE	d	内核的证实，表示消息处理完毕
RTF_DYNAMIC	D	(由重定向)动态创建
RTF_GATEWAY	G	目的主机是一个网关(非直接路由)
RTF_HOST	H	主机路由(否则，为网络路由)
RTF_LLINFO	L	当rt_llinfo指针无效时，由ARP设置
RTF_MASK	m	子网掩码存在(未使用)
RTF_MODIFIED	M	(由重定向)动态修改
RTF_PROTO1	1	协议专用的路由标志
RTF_PROTO2	2	协议专用的路由标志(ARP使用)
RTF_REJECT	R	有差错的丢弃分组(环回驱动器:图5-27)
RTF_STATIC	S	人工添加的路由(route程序)
RTF_UP	U	可用路由
RTF_XRESOLVE	X	由外部守护进程解析名字(用于X.25)

图18-25 rt_flags 的值

netstat不输出RTF_BLACKHOLE标志。两个标志为小写字符的常量， RTF_DONE和RTF_MASK，在路由消息中使用，但通常并不储存在路由表项中。

85 如果设置了RTF_GATEWAY标志，那么rt_gateway所含的插口地址结构的指针就指向网关的地址(即网关的IP地址)。同样， rt_gwroute就指向该网关的rtentry。后一个指针在ether_output(图4-15)中用到。

87 rt_refcnt是一个计数器，保存正在使用该结构的引用数目。在19.3节的最后部分将具体描述该计数器。在图18-2中，该计数器在“ Refs ”列输出。

88 当分配该结构存储空间时， rt_use被初始化为0。在图8-24中，可发现每次利用该路由输出一份IP数据报时，其值会随之递增。该计数器的值在图 18-2的“ Use ”栏中输出。

89-90 rt_ifp和rt_ifa分别指接口结构和接口地址结构。在图 6-5中曾指出一个给定的接口可以有多个地址，因此， rt_ifa是必需的。

92 rt_llinfo指针允许链路层协议在路由表项中储存该协议专用的结构指针。该指针通常与RTF_LLINFO标志一起使用。图21-1描述了ARP如何使用该指针。

route.h

```
54 struct rt_metrics {
55      u_long  rmx_locks;          /* bitmask for values kernel leaves alone */
56      u_long  rmx_mtu;            /* MTU for this path */
57      u_long  rmx_hopcount;       /* max hops expected */
58      u_long  rmx_expire;         /* lifetime for route, e.g. redirect */
59      u_long  rmx_recvpipe;       /* inbound delay-bandwith product */
60      u_long  rmx_sendpipe;       /* outbound delay-bandwith product */
61      u_long  rmx_ssthresh;       /* outbound gateway buffer limit */
62      u_long  rmx_rtt;            /* estimated round trip time */
63      u_long  rmx_rttvar;         /* estimated RTT variance */
64      u_long  rmx_pksent;         /* #packets sent using this route */
65 };
```

route.h

图18-26 rt_metrics 结构

93 图18-26给出了rt_metrics结构， rtentry结构含有该结构。图27-3显示了TCP使用了该结构的六个成员。

54-65 rmx_locks是一个比特掩码，由它告诉内核后面的八个度量中的哪些禁止修改。该比特掩码的值在图20-13中给出。

ARP(参见第21章)把rmx_expire用作每一个ARP路由项的定时器。与rmx_expire的注释不同的是， rm_expire不是用作重定向的。

图18-28概括了我们上面所阐述的各种结构和这些结构之间的关系，以及所引用的各种插口地址结构。图中给出的rtentry是图18-2中到128.32.33.5的路由。包含在rtentry中的另一个radix_node对应于图 18-4中位于该结点正上方的测试比特 36的内部结点。第一个ifaddr所指的两个sockaddr_dl结构如图3-38所示。另外，从图 6-5中也可注意到ifnet结构包含在le_softc结构中，第二个ifaddr结构包含在in_ifaddr结构中。

18.7 初始化：route_init和rtable_init函数

路由表的初始化过程并非是一目了然的，我们需要回顾一下第 7章中的domain结构。在描述这些函数调用之前，图18-27给出了各协议族中与domain结构相关的字段。

成 员	OSI值	Internet值	选路值	Unix值	XNS值	注 释
dom_family	AF_ISO	AF_INET	PF_ROUTE	AF_UNIX	AF_NS	
dom_init	0	0	route_init	0	0	
dom_rtattach	rn_inithead	rn_inithead	0	0	rn_inithead	比特
dom_rtoffset	48	32	0	0	16	字节
dom_maxrtkey	32	16	0	0	16	

图18-27 domain 结构中与路由选择有关的成员

PF_route域是唯一具有初始化函数的域。同样，只有那些需要路由表的域才有dom_rtattach函数，并且该函数总是rn_inithead。选路域和Unix域并不需要路由表。

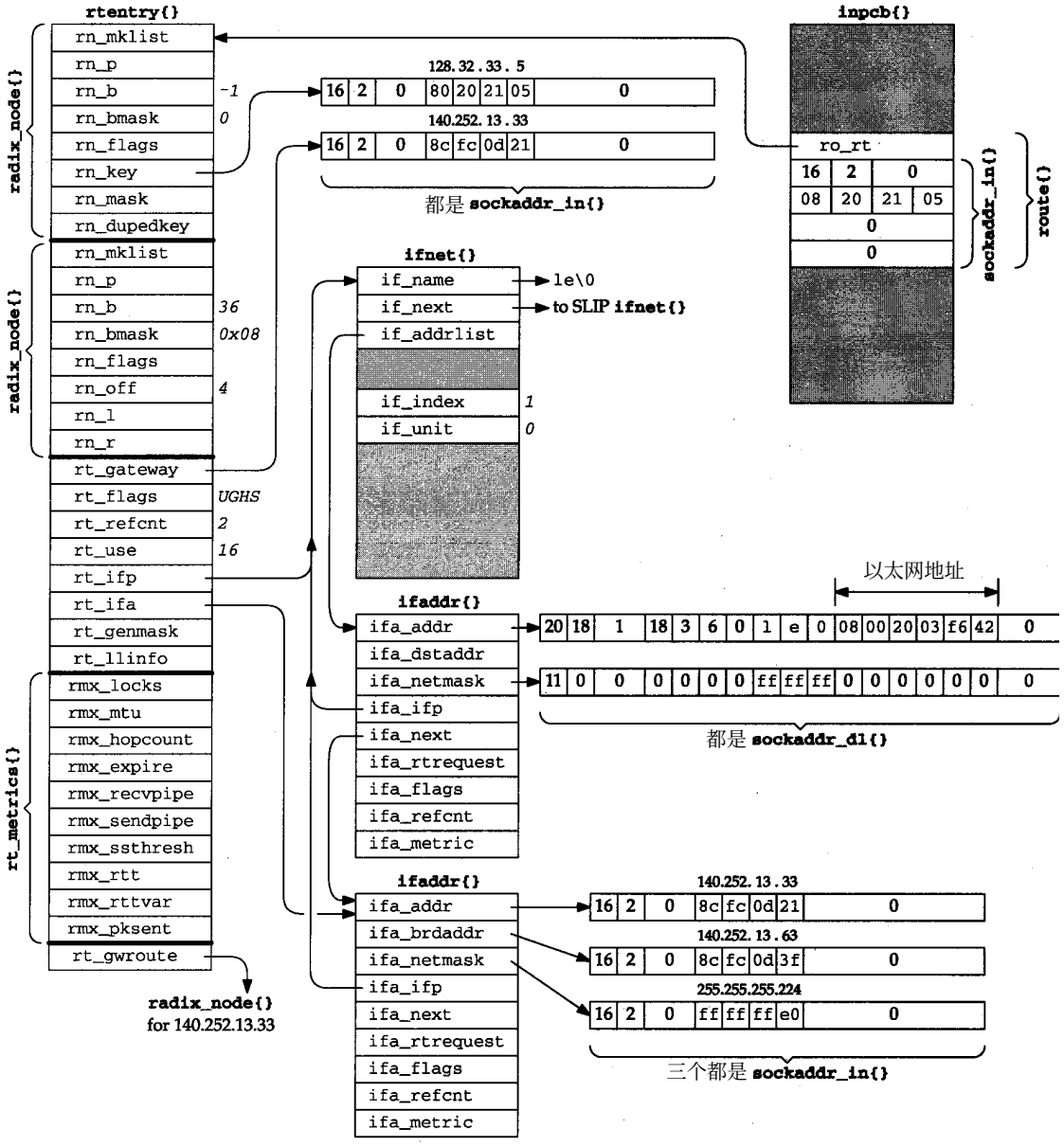

图18-28 选路结构小结

dom_rtoffset成员是以比特为单位的选路过程中被检测的第一个比特的偏移量 (从域的插口地址结构的起始处开始计算)。dom_maxrtkey给出了该结构的字节长度。在本章的前一部分的内容中，我们已经知道，sockaddr_in结构中的IP地址是从比特32开始的。dom_maxrtkey成员是协议的插口地址结构的字节长度：sockaddr_in的字节长度为16。

图18-29列出了路由表初始化过程所包含的步骤。

```
main()              /* kernel initialization */
{
    ...
    ifinit();
    domaininit();
    ...
}

domaininit()        /* Figure 7.15 */
{
    ...
    ADDDOMAIN(unix);
    ADDDOMAIN(route);
    ADDDOMAIN(inet);
    ADDDOMAIN(osi);
    ...
    for ( dp = all domains ) {
        (*dp->dom_init)();
        for ( pr = all protocols for this domain )
            (*pr->pr_init)();
    }
}

raw_init()          /* pr_init() function for SOCK_RAW/PF_ROUTE protocol */
{
                    初始化选路协议控制块的首部
}

route_init()        /* dom_init() function for PF_ROUTE domain */
{
    rn_init();
    rtable_init();
}

rn_init()
{
    for ( dp = all domains )
        if (dp->dom_maxrtkey > max_keylen)
            max_keylen = dp->dom_maxrtkey;
                分配并初始化 rn_zeros, rn_ones, masked_key;
    rn_inithead(&mask_rnhead);  /* allocate and init tree for masks */
}

rtable_init()
{
    for ( dp = all domains )
        (*dp->dom_rtattach)(&rt_tables[dp->dom_family]);
}

rn_inithead()       /* dom_attach() function for all protocol families */
{
                分配并初始化一个 radix_node_head 结构；
}
```

图18-29 初始化路由表时包含的步骤

在系统初始化时，内核的main函数将调用一次domaininit函数。ADDDOMAIN宏用于

创建一个domain结构的链表，并调用每个域的dom_init函数(如果定义了该函数)。正如图18-27所示，route_init是唯一的一个dom_init函数，其代码如图18-30所示。

```
route.c
49 void
50 route_init()
51 {
52     rn_init();  /* initialize all zeros, all ones, mask table */
53     rtable_init((void **) rt_tables);
54 }
route.c
```

图18-30 rout_init 函数

在图18-32中的函数rn_init只被调用一次。

在图18-31中的函数rtable_init也只被调用一次。它接着调用所有域的 dom_rtattach函数，这些函数为各自所属的域初始化一张路由表。

```
route.c
39 void
40 rtable_init(table)
41 void  **table;
42 {
43     struct domain *dom;
44     for (dom = domains; dom; dom = dom->dom_next)
45         if (dom->dom_rtattach)
46             dom->dom_rtattach(&table[dom->dom_family],
47                               dom->dom_rtoffset);
48 }
route.c
```

图18-31 rtable_init 函数：调用每一个域的dom_rtattach 函数

从图18-27中可知，rn_inithead是唯一的一个dom_rtattach函数，关于rn_inithead函数将在下一节中介绍。

18.8 初始化：rn_init和rn_inithead函数

图18-32中的函数rn_init只被route_init调用一次，用于初始化radix函数使用的一些全局变量。

1. 确定max_keylen

750-761 检查所有domain结构，并将全局变量max_keylen设置为最大的dom_maxrtkey值。在图18-27中最大值是32(对应于AF_ISO)，但是，在一个常用的不含OSI和XNS协议的系统中，max_key为16，即sockaddr_in结构的大小。

2. 分配并初始化rn_zeros、rn_ones和maskedKey

762-769 先分配了一个大小为max_keylen的三倍的缓存，并在全局变量rn_zeros中储存该缓存的指针。R_Malloc是一个调用内核的malloc函数的宏，它指定了M_RTABLE和M_DONTWAIT的类型。我们还会遇到Bcmp、Bcopy、Bzero和Free这些宏，它们对参数进行适当分类，并调用名称相似的内核函数。

该缓存被分解成三个部分，每一部分被初始化成如图 18-33所示。

rn_zeros是一个全0比特的数组，rn_ones是一个全1比特的数组，maskedKey数组用于存放被掩码过的查找键的临时副本。

radix.c

```
750 void
751 rn_init()
752 {
753     char    *cp, *cplim;
754     struct domain *dom;
755     for (dom = domains; dom; dom = dom->dom_next)
756         if (dom->dom_maxrtkey > max_keylen)
757             max_keylen = dom->dom_maxrtkey;
758     if (max_keylen == 0) {
759         printf("rn_init: radix functions require max_keylen be set\n");
760         return;
761     }
762     R_Malloc(rn_zeros, char *, 3 * max_keylen);
763     if (rn_zeros == NULL)
764         panic("rn_init");
765     Bzero(rn_zeros, 3 * max_keylen);
766     rn_ones = cp = rn_zeros + max_keylen;
767     maskedKey = cplim = rn_ones + max_keylen;
768     while (cp < cplim)
769         *cp++ = -1;
770     if (rn_inithead((void **) &mask_rnhead, 0) == 0)
771         panic("rn_init 2");
772 }
```

radix.c

图18-32 rn_init 函数

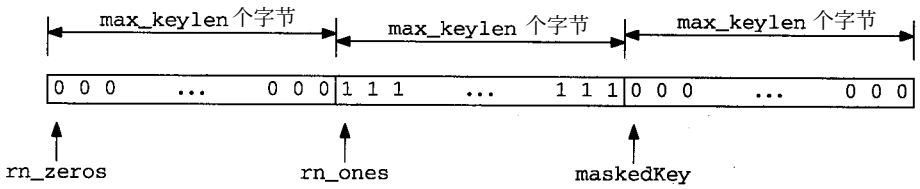

图18-33 rn_zeros 、rn_ones 和maskedKey 数组

3. 初始化掩码树

770-772 调用rn_inithead，初始化地址掩码路由树的首部；并使图 18-8中全局变量mask_rnhead指向该radix_node_head结构。

从图18-27可知，对于所有需要路由表的协议， rn_inithead也是它们的dom_attach函数。图 18-34给出的不是该函数的源代码，而是该函数为Internet协议创建的radix_node_head结构。

这三个radix_node结构组成了一棵树：中间的那个结构是树的顶点 (由rnh_treetop指向它)，第一个结构是树的最左边的叶子，最后一个结构是树的最右边的叶子。这三个结点的父指针(rn_p)都指向中间的那个结点。

rnh_nodes[1].rn_b的值32是待测试的比特位置。它来自于 Internet的domain结构中的dom_rtoffset成员(图18-27)。它的字节偏移量及字节掩码被预先计算出来，这样就不需要在处理过程中完成移位和掩码。其中，字节偏移量从插口地址结构起始处开始计算，它被存放在radix_node结构的rn_off成员中(在这个例子中它的值为 4)；字节掩码存放在rn_bmask成员中(在这个例子中为0x80)。无论何时往树中添加 radix_node结构，都要计

算这些值，以便于在转发过程中加快比较的速度。其他的例子有：在图 18-4中检测比特33的两个结点的偏移量和字节掩码分别为 4和0x40；检测比特63的两个结点的偏移量和字节掩码分别为7和0x01。

两个叶子中的rn_b成员的值 - 33是由 - 1减去该叶子的索引而得到的。

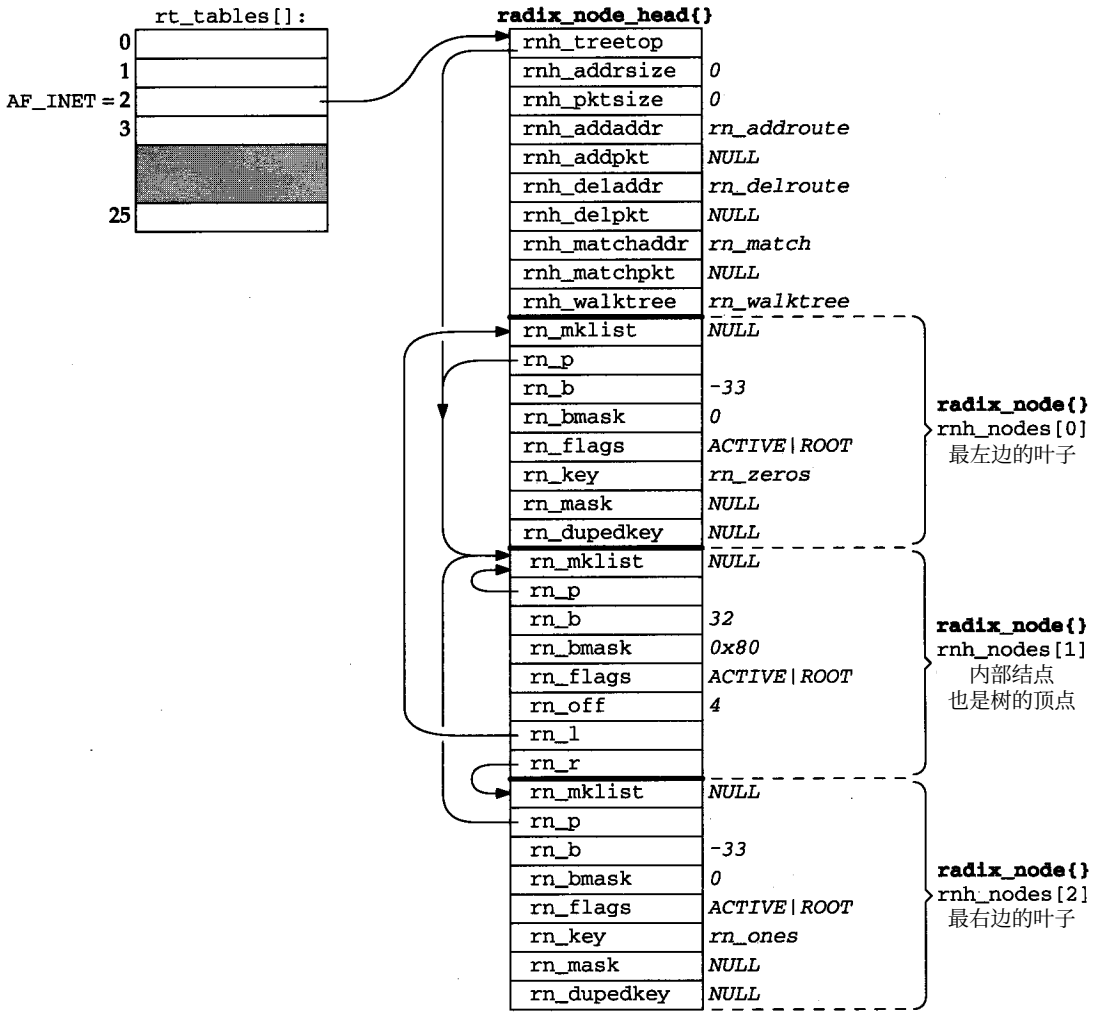

图18-34 rn_inithead 为Internet协议创建的radix_node_head 结构

最左边结点的键是全0(rn_zeros)，最右边结点的键是全1(rn_ones)。

这三个结点都设置了RNF_ROOT标志(我们省略了前缀RNF_)。这说明它们都是构成树的原始结点之一。它们也是唯一具有该标志的结点。

有一个细节我们没有提到，就是网络文件系统 (NFS)也使用路由表函数。对于本地主机的每一个装配点 (mount point)都分配一个radix_node_head结构，并且具有一个指向这些结构的指针数组 (利用协议族检索)，与rt_tables数组相似。每次输出装配点时，针对该装配点，把能装配该文件系统的主机的协议地址添加到适当的树中去。

18.9 重复键和掩码列表

在介绍查找路由表项的源代码之前，必须先理解 radix_node结构中的两个字段：一个是rn_dupedkey，它构成了附加的含重复键的 radix_node结构链表；另一个是rn_mklist，它是含网络掩码的radix_mask结构链表的开始。

先看一下图18-4中树的最左边标有“ end ”和“ default ”的两个框。这些就是重复键。最左边设有RNF_ROOT标志的结点(在图18-34中的rnh_nodes[0])有一个为全0比特的键，但是它和默认路由的键相同。如果创建一个255.255.255.255的路由表项(但该地址是受限的广播地址，不会在路由树中出现)，则我们会在树的最右端结点 (该结点有一个值为全1比特的键)遇到同样的问题。总的来说，如果每次都有不同的掩码，那么Net/3中的radix结点函数就允许重复任何键。

图18-35给出了两个具有全 0比特重复键的结点。在这个图中，我们去掉了 rn_flags中的RNF_前缀，并且省略了非空父指针、左指针和右指针，因为它们与要讨论的内容无关。

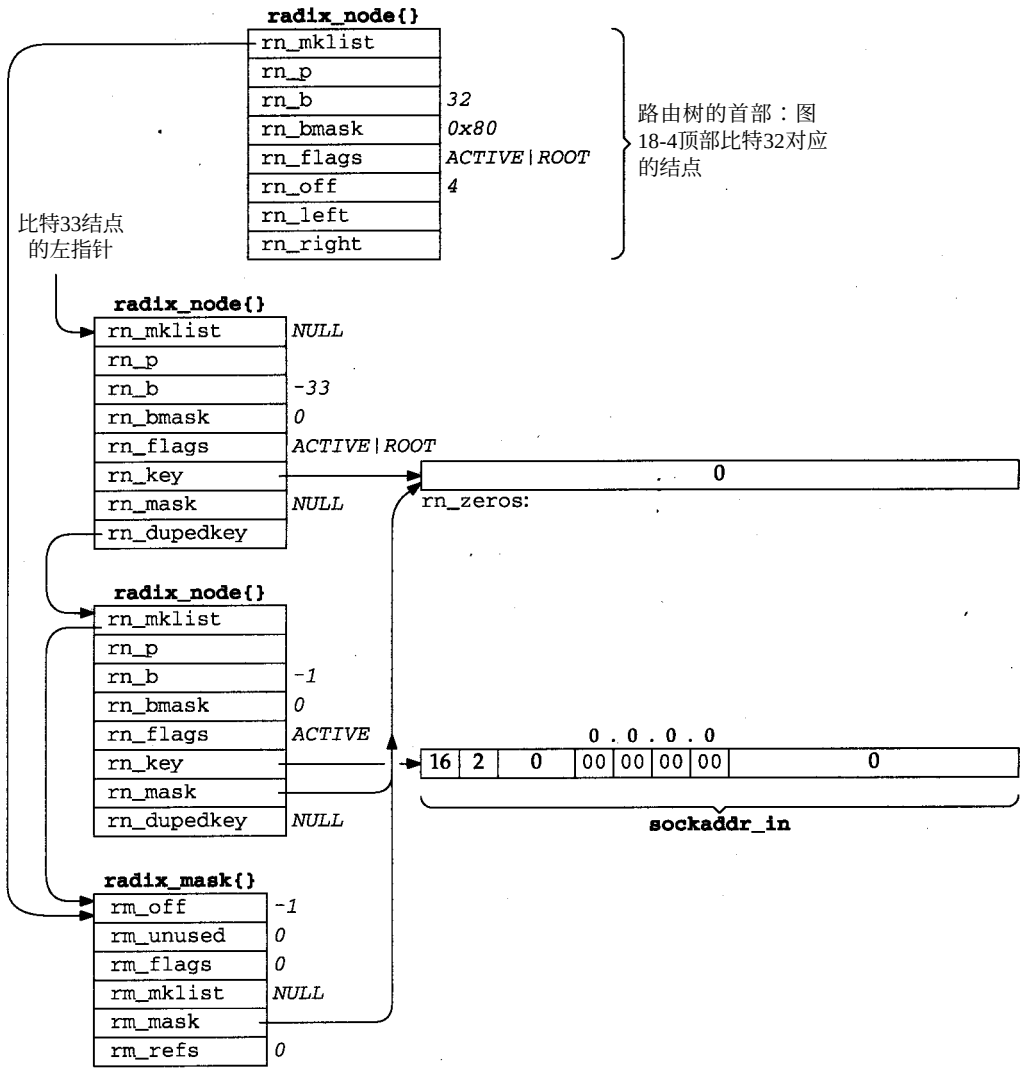

图18-35 值为全0的键的重复结点

图中最上面的结点即为路由树的顶点——图18-4中顶部比特32对应的结点。接下来的两个结点是叶子(它们的rn_b为负值)，其中第一个叶子的rn_dupedkey成员指向第二个结点。第一个叶子是图18-34中的rnh_nodes[0]结构，该结构是树的左边标有“ end ”的结点——它设有RNF_ROOT标志。它的键被rn_inithead设为rn_zeros。

第二个叶子是默认路由的表项。它的 rn_key指向值为0.0.0.0的sockaddr_in结构，并具有一个全0的掩码。由于掩码表中相同的掩码是共享的，因此，该叶子的 rn_mask也指向rn_zeros。

> 通常，键是不共享的，更不会与掩码共享。由于两个标有“ end ”的结点的rn_key指针(具有RNF_ROOT标志)是由rn_inithead(图18-34)创建的，因此这两个指针例外。左边标有end的结点的键指向rn_zeros，右边标有“ end ”的结点的键指向rn_ones。

最后一个是radix_mask结构，树的顶结点和默认路由对应的叶子都指向这个结构。这个列表是树的顶结点的掩码列表，在查找网络掩码时，回溯算法将使用它。 radix_mask结构列表和内部结点一起确定了运用于从该结点开始的子树的掩码。在重复键的例子中，掩码列表和叶子出现在一起，跟着的这个例子也是这样的。

现在我们给出一个特意添加到选路树中的重复键和所得到的掩码列表。在图 18-4中有一个主机路由 127.0.0.1和一个网络路由 127.0.0.0。图中采用了 A类网络路由的默认掩码，即0xff000000。如果我们把跟在A类网络号之后的24 bit分解成一个16 bit子网号和一个8 bit主机号，就可以为子网127.0.0添加一个掩码为0xffffff00的路由：

```
bsdi $ route add 127.0.0.0 -netmask 0xffffff00 140 252 13 33
```

虽然在这种情况下使用网络127没什么实际意义，但是我们感兴趣的是所得到的路由表结构。虽然重复键在Internet协议中并不常见(除了前面例子中的默认路由之外)，但是仍需要利用重复键来为所有网络的0号子网提供路由。

在网络号127的这三个路由表项中存在一个隐含的优先规则。如果查找键是 127.0.0.1，则它和这三个路由表项都匹配，但是只选择主机路由，因为它是最匹配的：其掩码(0xffffffff)含有最多的1。如果查找键是127.0.0.2，它与两个网络路由匹配，但是掩码为0xffffff00的子网0的路由比掩码为0xff0000的路由更匹配。如果查找键为 127.0.2.3，那么只与掩码为0xff000000的路由表项匹配。

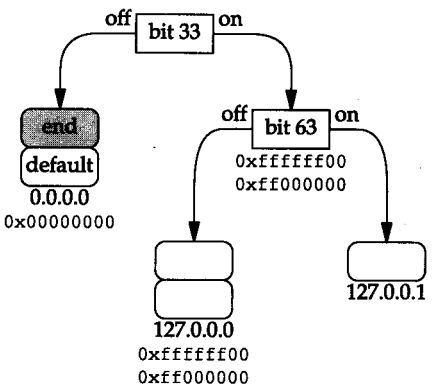

图18-36 反映重复键127.0.0.0的路由树

图18-36给出了添加路由之后得到的树结构，从图18-4中对应比特33的内部结点处开始。由于这个重复键有两个叶子，我们用两个框来表示键值为 127.0.0.0的路由表项。

图18-37给出了所得到的radix_nod和radix_mask结构。

首先看一下每一个radix_node的radix_mask结构的链表。最上端结点 (比特63)的掩码列表由0xffffff00及其后的0xff000000组成。在列表中首先遇到的是更匹配的掩码，这样它就能够更早地被测试到。第二个 radix_node(rn_b值为 - 57的那个)的掩码列表与第一个相同。但是第三个radix_node的掩码列表仅由值为0xff000000的掩码构成。

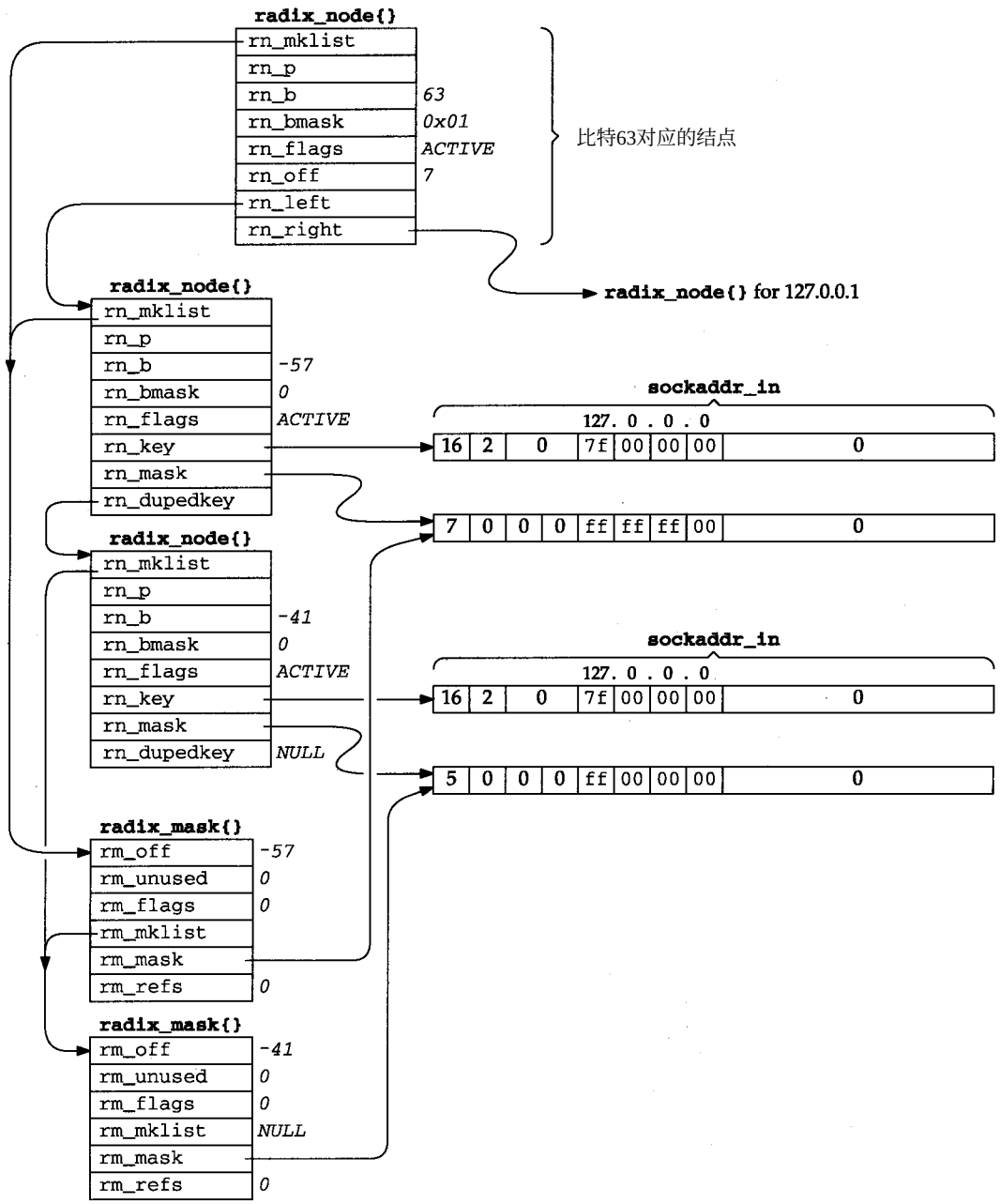

图18-37 网络127.0.0.0的重复键的路由表结构举例

应注意的是，具有相同值的掩码之间可以共享，但是具有相同值的键之间不能共享。这是因为掩码被保存在它们自己的路由树中，可以显式地被共享，而且值相同的掩码经常出现(例如，每个C类网络路由都有相同的掩码`0xffffff00`)，但是值相同的键却不常见。

18.10 rn_match函数

现在我们介绍`rn_match`函数，在Internet协议中，它被称为`rnh_matchaddr`函数。在

后面学习中，我们可以看到它将被rtallocl函数调用(而rtalloc1函数将被rtalloc函数调用)。具体算法如下：

1) 从树的顶端开始搜索，直到到达与查找键的比特相应的叶子。检测该叶子，看能否得到一个精确的匹配(图18-38)。

2) 检测该叶结点，看是否能得到匹配的网络地址。

3) 回溯(图18-43)。

图18-38给出了rn_match的第一部分。

radix.c

```
135 struct radix_node *
136 rn_match(v_arg, head)
137 void    *v_arg;
138 struct radix_node_head *head;
139 {
140     caddr_t v = v_arg;
141     struct radix_node *t = head->rnh_treetop, *x;
142     caddr_t cp = v, cp2, cp3;
143     caddr_t cplim, mstart;
144     struct radix_node *saved_t, *top = t;
145     int     off = t->rn_off, vlen = *(u_char *) cp, matched_off;
146     /*
147      * Open code rn_search(v, top) to avoid overhead of extra
148      * subroutine call.
149      */
150     for (; t->rn_b >= 0;) {
151         if (t->rn_bmask & cp[t->rn_off])
152             t = t->rn_r;        /* right if bit on */
153         else
154             t = t->rn_l;        /* left if bit off */
155     }
156     /*
157      * See if we match exactly as a host destination
158      */
159     cp += off;
160     cp2 = t->rn_key + off;
161     cplim = v + vlen;
162     for (; cp < cplim; cp++, cp2++)
163         if (*cp != *cp2)
164             goto on1;
165     /*
166      * This extra grot is in case we are explicitly asked
167      * to look up the default.  Ugh!
168      */
169     if ((t->rn_flags & RNF_ROOT) && t->rn_dupedkey)
170         t = t->rn_dupedkey;
171     return t;
172 on1:
```

radix.c

图18-38 rn_match 函数：沿着树向下搜索，查找严格匹配的主机地址

135-145 第一个参数v_arg是一个插口地址结构的指针，第二个参数head是该协议的指向radix_node_head结构的指针。所有协议都可调用这个函数 (图18-17)，但调用时使用不同的head参数。

在变量声明中，off是树的顶结点的rn_off成员(对Internet地址，其值为4，见图18-34)，

vlen是查找键插口地址结构中的长度字段(对Internet地址，其值为16)。

1. 沿着树向下搜索到相应的叶子

146-155 这个循环从树的顶结点开始，然后沿树的左右分支搜索，直到遇到一个叶子为止(rn_b小于0)。每次测试相应比特时，都利用了事先计算好的rn_bmask中的字节掩码和事先计算好的rn_off中的偏移量。对于Internet地址而言，rn_off为4、5、6或7。

2. 检测是否精确匹配

156-164 当遇到叶子时，首先检测能否精确匹配。比较插口地址结构中从协议族的 rn_off值开始的所有字节。图18-39给出了Internet插口地址结构的比较情况。

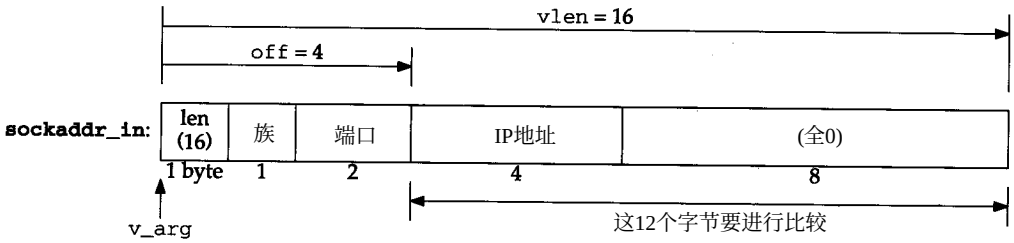

图18-39 比较sockaddr_in 结构时的各种变量

如果发现匹配不成功，就立刻跳到on1。

通常，sockaddr_in的最后8个字节为0，但是地址解析协议代理 (proxy ARP)(21.12节)会设置其中的一个为非零。这就允许一个给定的 IP地址有两个路由表项：一个对应于正常IP地址(最后8个字节为0)，另一个对应于相同IP地址的地址解析协议代理(最后8个字节中有一个为非零)。

图18-39中的长度字节在函数的一开始时就赋值给了vlen，并且我们还会看到rtallocl将利用family成员来选择路由表进行搜索。选路函数未使用 port成员。

3. 显示地检测默认地址

165-172 图18-35给出了存储在键为0的重复叶子中的默认路由。第一个重复的叶子设有RNF_ROOT标志。因此，如果在匹配的结点中设有RNF_ROOT标志，并且该叶子含有重复键，那么就返回指针rn_dupedkey的值(即图18-35中含默认路由的结点的指针)。如果路由树中没有默认路由，则查找过程匹配左边标有“ end ”的叶子(键为全0比特)；或者如果查找时遇到右边标有“ end ”的叶子(键值为全1比特)，那么返回指针t，它指向一个设有RNF_ROOT标志的结点。我们将看到rtallocl会显式地检查匹配结点是否设有这个标志，并判断匹配是否失败。

程序执行到此时，rn_match函数已经到达了某个叶子上，但是它并不是查找键的精确匹配。函数的下一部分将检测该叶子是否为匹配的网络地址，如图 18-40所示。

173-174 cp指向该查找键中那个不相等的字节。matched_off被赋值为该字节在插口地址结构中的位置偏移量。

175-183 do while循环反复与所有重复叶子中的每一个具有网络掩码的叶子进行比较。下面我们通过一个例子来看这段代码。假定我们要在图 18-4所示的路由表中查找 IP地址140.252.13.60。查找会在标有140.252.13.32(比特62和63都为0)的结点处终止，该结点包含一个网络掩码。图18-41给出了图18-40中的for循环开始执行时的结构。

radix.c

```
173     matched_off = cp - v;
174     saved_t = t;
175     do {
176         if (t->rn_mask) {
177             /*
178              * Even if we don't match exactly as a host;
179              * we may match if the leaf we wound up at is
180              * a route to a net.
181              */
182             cp3 = matched_off + t->rn_mask;
183             cp2 = matched_off + t->rn_key;
184             for (; cp < cplim; cp++)
185                 if ((*cp2++ ^ *cp) & *cp3++)
186                     break;
187             if (cp == cplim)
188                 return t;
189             cp = matched_off + v;
190         }
191     } while (t = t->rn_dupedkey);
192     t = saved_t;
```

radix.c

图18-40 rn_match 函数：检测是否为匹配的网络地址

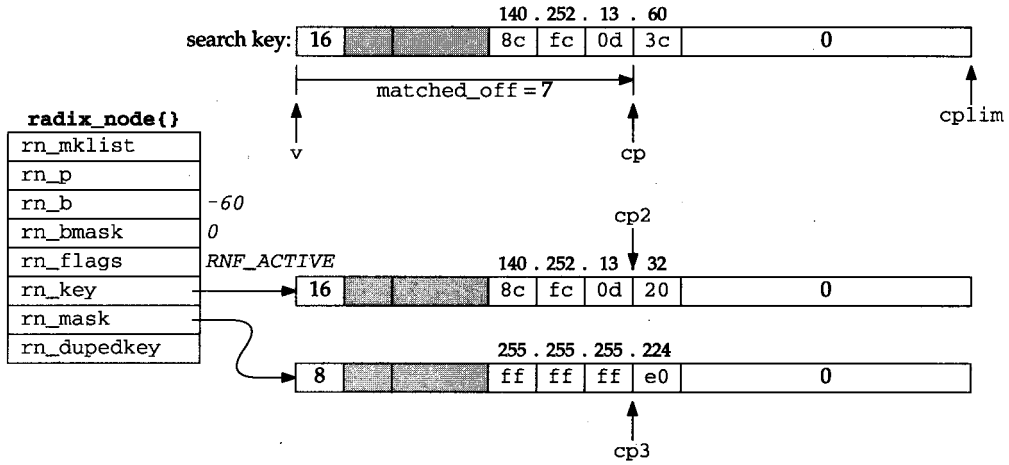

图18-41 比较网络掩码的例子

虽然查找键和路由表键都是sockaddr_in结构，但是掩码的长度并不相同。该掩码长度是非零字节的最小数目。从该点之后直到max_keylen之间的所有字节都为0。

184-190 逐个字节地对查找关键字和路由表键进行异或运算，并将结果同网络掩码进行逻辑与运算。如果所得到的字节出现非零值，就会由于不匹配而终止循环 (习题18.1)。如果循环正常终止，那么与网络掩码进行逻辑与运算后的查找键就和路由表项相匹配。程序将返回指向该路由表项的指针。

查看IP地址的第四个字节，我们可以从图 18-42中看出本例子是如何匹配成功的，以及 IP地址140.252.13.188是如何匹配失败的。采用这两个地址，是因为它们中的比特 57、62和63都为0，查找都在图18-41给出的结点上终止。

第一个例子(140.252.13.60)匹配成功是因为逻辑与运算的结果为 0(并且地址、键和掩码中所有剩余的字节全都为0)。另一个例子匹配不成功是因为逻辑与运算的结果为非零。

	查找键 = 140.252.13.60	查找键 = 140.252.13.188
查找键字节(*cp): 路由表键字节(*cp2):	0011 1100 = 3c 0010 0000 = 20	1011 1100 = bc 0010 0000 = 20
异或: 网络掩码字节(*cp3):	0001 1100 1110 0000 = e0	1001 1100 1110 0000 = e0
逻辑与:	0000 0000	1000 0000

图18-42 用网络掩码进行关键字匹配的例子

191 如果路由表项含有重复键，那么对每一个键都要执行一次该循环体。

rn_match的最后一部分，如图18-43所示，沿路由树向上回溯，以查找匹配的网络地址或默认地址。

radix.c

```
193     /* start searching up the tree */
194     do {
195         struct radix_mask *m;
196         t = t->rn_p;
197         if (m = t->rn_mklist) {
198             /*
199              * After doing measurements here, it may
200              * turn out to be faster to open code
201              * rn_search_m here instead of always
202              * copying and masking.
203              */
204             off = min(t->rn_off, matched_off);
205             mstart = maskedKey + off;
206             do {
207                 cp2 = mstart;
208                 cp3 = m->rm_mask + off;
209                 for (cp = v + off; cp < cplim;)
210                     *cp2++ = *cp++ & *cp3++;
211                 x = rn_search(maskedKey, t);
212                 while (x && x->rn_mask != m->rm_mask)
213                     x = x->rn_dupedkey;
214                 if (x &&
215                     (Bcmp(mstart, x->rn_key + off,
216                         vlen - off) == 0))
217                     return x;
218             } while (m = m->rm_mklist);
219         }
220     } while (t != top);
221     return 0;
222 };
```

radix.c

图18-43 rn_match 函数：沿树向上回溯

193-195 do while 循环沿着路由树一直向上，检测每一层的结点，直至检测到树的顶端为止。

196 指向父结点的指针的值被赋给了指针 t，即向上移动了一层。可见，在每一个结点中包含一个父指针能够简化回溯操作。

197-210 对于回溯到的每一层，只要内部结点的掩码列表非空，就对该层进行检测。rn_mklist是指向radix_node结构的链表的指针，链表中的每一个 radix_node结构都包含一个掩码，这些掩码将应用于从该结点开始的子树。程序中的内部 do while循环将遍历每一个radix_mask结构。

利用前面的例子，140.252.13.188，图18-44给出了在最内层的for循环开始时的各种数据结构。这个循环对每个掩码中的字节和对应的查找键的字节进行逻辑与操作，并将结果保存在全局变量maskedKey中。该掩码值为0xfffffe0，搜索会从图18-4中的叶结点140.252.13.32处回溯两层，到达测试比特62的结点。

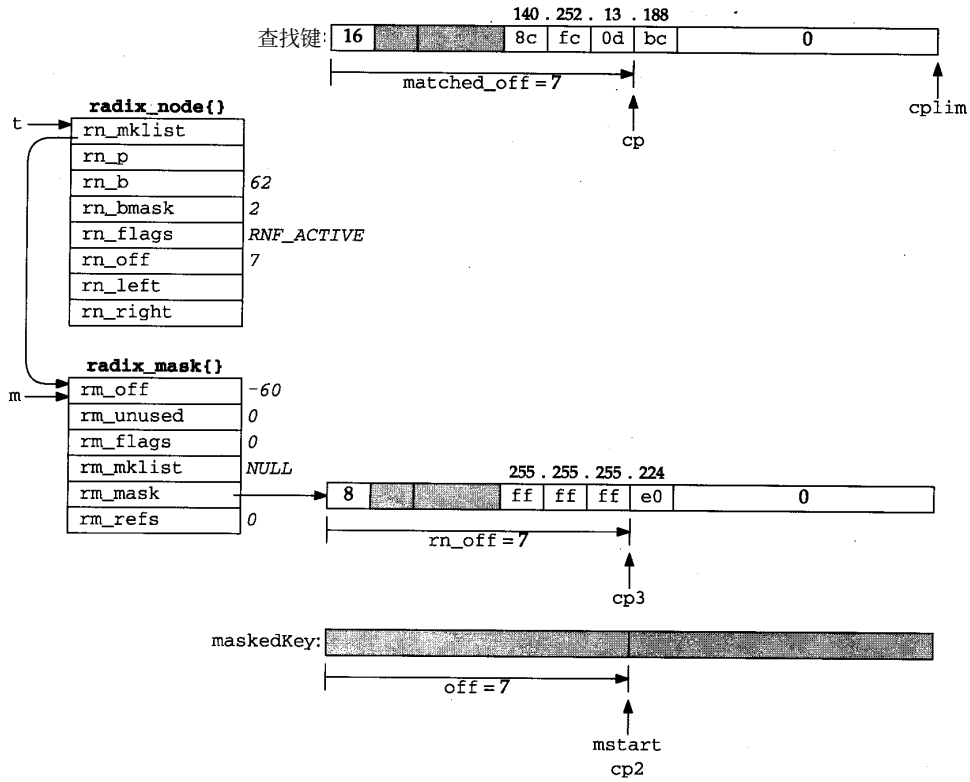

图18-44 利用掩码过的查找键进行再次搜索的准备

for循环完成后，掩码过程也就完成了，再调用rn_search(如图18-48所示)，其调用参数以maskedKey为查找键，以指针t为查找子树的顶点。图18-45给出了我们所举例子中的maskedKey的值。

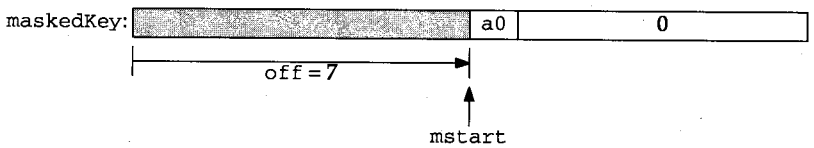

图18-45 调用rn_search 时的maskedKey

字节0xa0是0xbc(188，查找键)和0xe0(掩码)逻辑与运算的结果。

211 rn_search从起点开始沿着树往下搜索，根据查找键来确定沿向左或向右的分支进行搜索，直到到达某个叶子。在这个例子中，查找键有9个字节，如图18-45所示，所到达的是图18-4中标有140.252.13.32的那个叶子，这是因为在字节0xa0中比特62和63都为0。图18-46给出了调用Bcmp检验是否匹配时的数据结构。

由于这两个9字节的字符串不相同，所以这次比较失败。

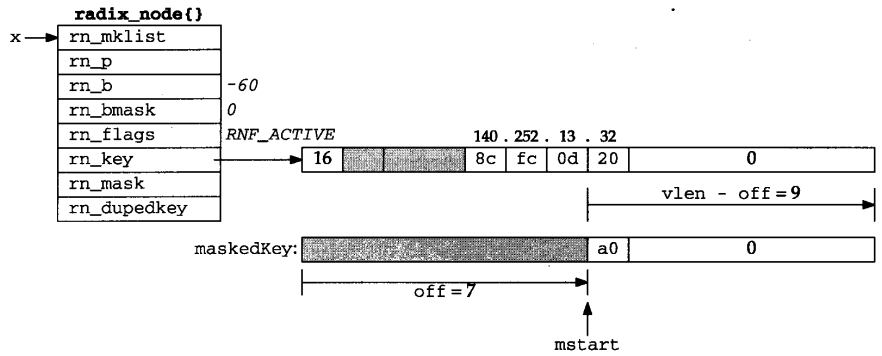

图18-46 maskedKey 和新叶结点之间的比较

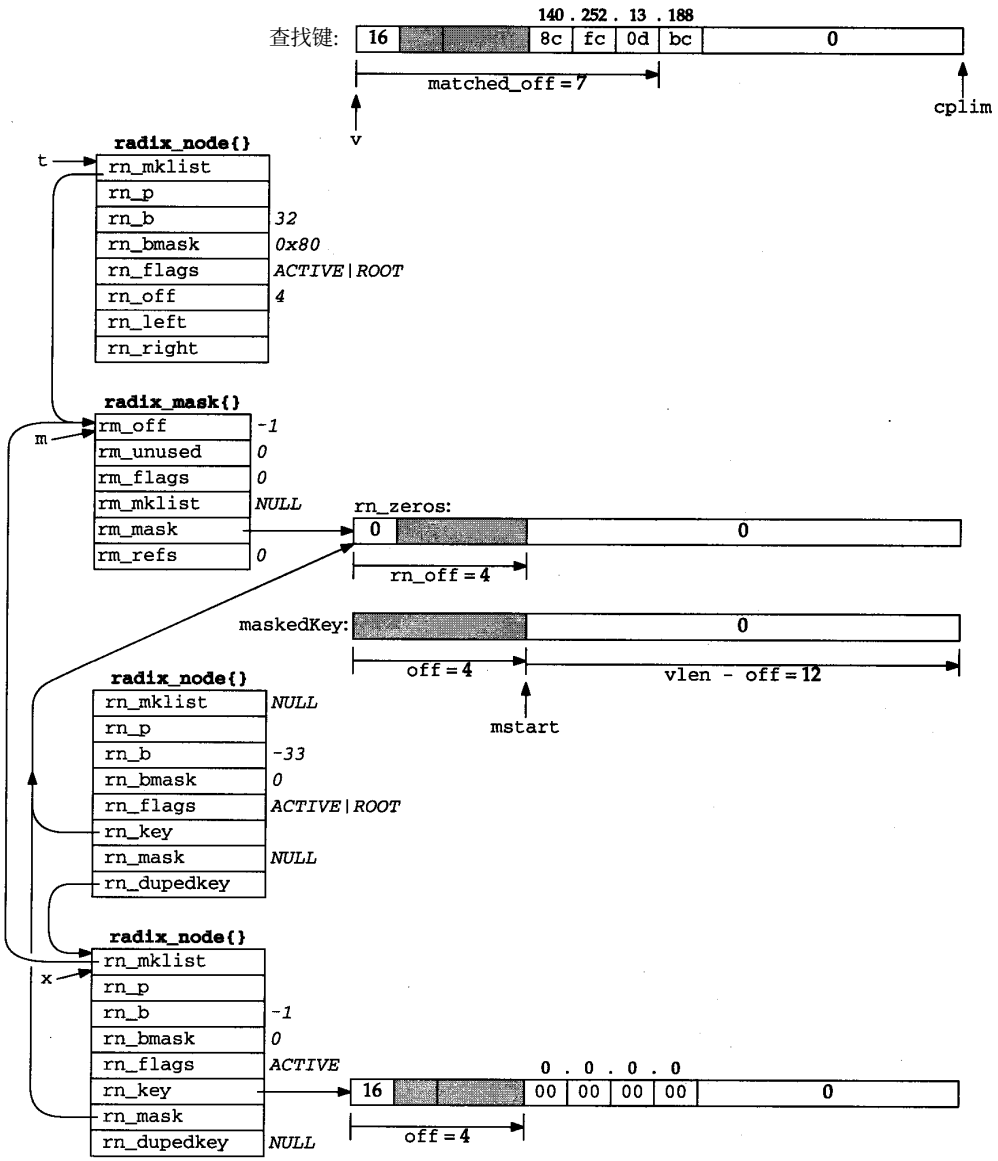

图18-47 回溯到路由树的顶端和查找默认叶子的 rn_search

212-221 该while循环处理各重复键，且处理每个重复键时的掩码不同。唯一被比较的重复键是那个rn_mask指针与m->rm_mask相等的键。下面以图18-36和图18-37为例进行说明。如果查找从比特63的结点处开始，第一次内部do while循环中，m指向radix_mask结构0xffffff00。当rn_search返回指向第一个重复叶子127.0.0.0的指针时，该叶子的rm_mask等于m->rm_mask，因此，就调用Bcmp。如果比较失败，m的值就被设置成指向列表中的下一个radix_mask结构(具有掩码0xff000000)的指针，并且对新掩码再次执行do while循环体。rn_search再一次返回指向第一个重复叶子127.0.0.0的指针，但是它的rn_mask并不等于m->rm_mask。While继续进行到下一个重复叶子，它的rn_mask与m->rm_mask恰好相等。

现在回到查找键为140.252.13.188的例子中，由于从检测比特62的结点处开始的搜索失败，因此，沿着树向上继续回溯，直到到达树的顶点，该顶点就是沿树向上的下一个rn_mklist为非空的结点。

图18-47给出了到达树的顶结点时的数据结构。此时，计算maskedKey(为全0)，并且rn_search从这个结点(树的顶结点)处开始，继续沿着树的左分支向下两层到达图18-4中标有“default”的叶子。

当rn_search返回时，x指向rn_b值为-33的radix_node，这是从树的顶端开始沿两个左分支向下之后遇到的第一个叶子。但是x->rn_mask(为空)与m->rm_mask不等，因此，将x->rn_dupedkey赋给x。用于测试的while循环再次执行，但是，此时x->rn_mask等于m->rm_mask，因此该while循环终止。Bcmp对从mstart开始的12个值为0的字节和从x->rn_key加4开始的12个值为0的字节进行比较，结果相等，函数返回指针x，该指针指向默认路由的路由项。

18.11 rn_search函数

在前面一节中，我们已经知道rn_match调用了rn_search来搜索路由表的子树。如图18-48所示。

radix.c

```
79 struct radix_node *
80 rn_search(v_arg, head)
81 void    *v_arg;
82 struct radix_node *head;
83 {
84     struct radix_node *x;
85     caddr_t v;

86     for (x = head, v = v_arg; x->rn_b >= 0;) {
87         if (x->rn_bmask & v[x->rn_off])
88             x = x->rn_r;          /* right if bit on */
89         else
90             x = x->rn_l;          /* left if bit off */
91     }
92     return (x);
93 };
```

radix.c

图18-48 rn_search 函数

这个循环和图18-38中的相似。它在每一个结点上比较查找键中的一位比特，如果该比特

为0，就通向左边的分支，如果该比特为1，就通向右边的分支。在遇到一个叶子时终止搜索，并返回指向该叶子的指针。

18.12 小结

每一个路由表项都由一个键来标识：在IP协议中就是目的IP地址，该IP地址可以是一个主机地址或者是一个具有相应网络掩码的网络地址。一旦键的搜索确定了路由表项，在该表项中的其他信息就会指定一个路由器的IP地址，到目的地址的数据报就会发往该指定地址，还会指明要用到的接口的指针、度量等等。

由Internet协议维护的信息是`route`结构，该`route`结构只有两个成员构成：指向路由表项的指针和目的地址。在UDP、TCP和原始IP使用的每个Internet协议控制块中，我们都会遇到由Internet协议维护的`route`结构。

Patricia树数据结构非常适合于路由表。由于路由表的查找要比添加或者删除路由频繁得多，因此从性能的角度来看，在路由表中使用Patricia树就更加有意义。Patricia树虽然不利于添加和删除这些附加工作，但是加快了查找的速度。 [Sklower 1991]给出的radix树方法和Net/1散列表的比较结果表明，用radix树方法构造测试树用比Net/1散列表法快一倍，搜索速度快三倍。

习题

18.1 我们说过，在图18-3中，查找键与路由表项匹配的一般条件是，它和路由表掩码的逻辑与运算的结果等于路由表键。但是在图18-40中采用了不同的测试方法。请建立一个逻辑真值表以证明这两种方法等价。

18.2 假设某个Net/3系统中的路由表需要20 000个表项(IP地址)。在不考虑掩码的情况下，请估算大约需要多大的存储器？

18.3 `radix_node`结构对路由表键的长度限制是多少？