

[19] 中华人民共和国国家知识产权局

[51] Int. Cl.



[12] 发明专利说明书

H04L 12/56 (2006.01)

H04L 12/24 (2006.01)

H04L 12/26 (2006.01)

专利号 ZL 03154134.8

[45] 授权公告日 2009年4月29日

[11] 授权公告号 CN 100484084C

[22] 申请日 2003.8.12 [21] 申请号 03154134.8

[73] 专利权人 华为技术有限公司

地址 518129 广东省深圳市龙岗区坂田华为总部办公楼

[72] 发明人 王 军

[56] 参考文献

CN1270728A 2000.10.18

US6061712A 2000.5.9

US2003130981A1 2003.7.10

US6278995B1 2001.8.21

审查员 陈 琼

[74] 专利代理机构 北京同达信恒知识产权代理有限公司

代理人 黄志华

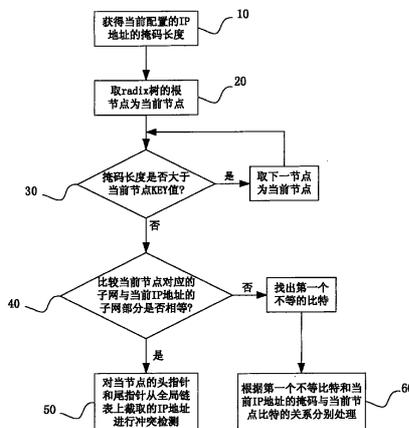
权利要求书 3 页 说明书 14 页 附图 4 页

[54] 发明名称

一种检索 IP 地址的方法

[57] 摘要

本发明公开了一种检索 IP 地址的方法，用于从 IP 地址集中搜索与配置的当前 IP 地址相关的地址；IP 地址形成的全局地址链表按包含关系排序，并以 IP 地址的掩码长度为节点按递升顺序形成树索引结构，节点的头指针和尾指针指向所述全局地址链表中的 IP 地址，该两个指针从全局地址链表上截取的一段地址集合全部被该节点所代表的 IP 子网所包含；检测时根据配置的当前 IP 地址的掩码长度从树索引结构中查找到相应的节点，并从该节点头指针和尾指针从全局链表中括住的 IP 地址中进行冲突判断等操作。



1、一种检索 IP 地址的方法，根据提供的当前 IP 地址至少从 IP 地址集中搜索相关的 IP 地址，所述 IP 地址集中的 IP 地址形成全局地址链表，其特征在于该方法包含下述步骤：

a、获得当前 IP 地址的掩码长度；

b、根据节点对应的配置的 IP 地址的掩码长度按递升顺序形成树索引结构，从所述树索引结构的根节点开始，将节点代表的掩码长度与当前 IP 地址的掩码长度进行比较，如果找到第一个大于或等于当前 IP 地址的掩码长度的当前节点，继续步骤 c；否则进行步骤 e；

c、比较当前 IP 地址的子网与当前节点对应的 IP 地址的子网是否相同，如果相同则进行步骤 d，如果不相同则进行步骤 e；

d、取得当前节点的头指针和尾指针之间从全局地址链表中所截取的所有 IP 地址，所述所截取的 IP 地址被当前节点对应的 IP 地址的子网所包含，并采用判断冲突的策略对每一个 IP 地址进行判断；如果发现冲突则提示并结束检索，否则进行步骤 e；

e、确认当前 IP 地址与现有 IP 地址不冲突和将当前 IP 地址加入到所述全局地址链表中。

2、如权利要求 1 所述的方法，其特征在于，所述的树索引结构为一种能够实现最长匹配的二叉树的数据结构。

3、如权利要求 1 或 2 所述的方法，其特征在于，所述全局地址链表按 IP 地址的包含关系排序。

4、如权利要求 3 所述的方法，其特征在于，如果全局地址链表中有多个掩码长度相等并且子网相同的 IP 地址，则该多个 IP 地址在全局链表中连续排列。

5、如权利要求 1 所述的方法，其特征在于，步骤 b 中，当前 IP 地址的掩码长度比当前节点代表的掩码长度大时，判断当前 IP 地址在当前节点用于表示 IP 地址掩码长度的 KEY 值所指定位置的比特值，如果为 1 则取右子节点为当前节

点继续比较，否则取左子节点为当前节点继续比较。

6、如权利要求1所述的方法，其特征在于，步骤c中，当前IP地址的子网与当前节点对应的IP地址的子网不相同时，通过当前IP地址与当前节点对应的IP地址逐个比特比较，确定第一个不等比特的序号，并根据该序号与当前IP地址的掩码和当前节点比特的关系分别处理：

(1)如果所述序号等于当前节点代表的掩码长度，则将当前IP地址作为当前节点的子树；

(2)如果所述序号等于当前IP地址的掩码长度，则将当前节点作为当前IP地址的子树；

(3)如果所述序号比当前节点代表的掩码长度和当前IP地址的掩码长度都小，则新建一个节点作为分叉节点，当前节点与当前IP地址互为兄弟，都作为分叉节点的左右子树。

7、一种检索IP地址的方法，从IP地址集中搜索到当前IP地址后删除，所述IP地址集中的IP地址形成全局地址链表，其特征在于该方法包含下述步骤：

a、获得当前IP地址的掩码长度；

b、根据节点对应的配置的IP地址的掩码长度按递升顺序形成树索引结构，从所述树索引结构的根节点开始，将节点代表的掩码长度与当前IP地址的掩码长度进行比较，找到当前IP地址的掩码小于或等于节点代表的掩码长度的当前节点；

c、比较当前IP地址与当前节点对应的IP地址的子网是否相同，如果相同则进行步骤d，如果不相同则提示没有当前IP地址并结束；

d、取得当前节点的头指针和尾指针从全局地址链表中所截取的所有IP地址，所述所截取的IP地址被当前节点对应的IP地址的子网所包含，从这些IP地址中查找当前IP地址，如果找到则删除该当前IP地址，并进行步骤e；如果未找到则提示无待删除的IP地址并结束；

e、根据全局地址链表中当前IP地址中保存的回索引树内部节点的指针找到相应的节点，按下述情况分别对节点进行处理：

(1) 如果节点同时存在左右子树, 则该节点保留作为分叉节点, 结束处理;

(2) 如果节点没有子树, 则删除本节点, 将父节点指向本节点的指针清空; 如果此时父节点还有 IP 地址与之相连, 则结束处理, 否则取父节点为当前节点, 进行步骤 (3);

(3) 如果节点只有左子树或只有右子树, 则删除本节点, 同时将左子树或右子树挂到本节点在其父节点的相应指针上。

8、如权利要求 7 所述的方法, 其特征在于, 所述的树索引结构为一种能够实现最长匹配的二叉树的数据结构。

9、如权利要求 7 所述的方法, 其特征在于, 所述全局地址链表按 IP 地址的包含关系排序。

10、如权利要求 7 所述的方法, 其特征在于, 如果全局地址链表中有多个掩码长度相等并且子网相同的 IP 地址, 则该多个 IP 地址在全局链表中连续排列。

11、如权利要求 7 所述的方法, 其特征在于, 步骤 b 中, 当前 IP 地址的掩码长度比当前节点代表的掩码长度大时, 判断当前 IP 地址在当前节点用于表示 IP 地址掩码长度的 KEY 值所指定位置的比特值, 如果为 1 则取右子节点为当前节点继续比较, 否则取左子节点为当前节点继续比较。

13、如权利要求 7 所述的方法, 其特征在于, 处理节点后还包括对节点的头指针和尾指针按下述情况进行处理:

(1) 如果删除的 IP 地址仅被节点的头指针指向, 则将头指针移向后一个 IP 地址;

(2) 如果删除的 IP 地址仅仅被尾指针指向, 则将尾指针移向前一个 IP 地址;

(3) 如果删除的 IP 地址同时被头指针和尾指针指向, 则删除头指针和尾指针。

一种检索 IP 地址的方法

技术领域

本发明涉及一种检索 IP 地址的方法。

背景技术

给路由设备配置 IP 地址时，出于安全性的考虑一般都要检测配置的 IP 地址与现有的 IP 地址是否冲突。如果冲突则配置不成功并提示用户，若不冲突则接纳配置的 IP 地址并生成相应的路由。对于冲突检测，可以分成两类问题分别解决：1) 冲突检测的策略；2) IP 地址的定位。

所谓冲突检测的策略，不同的产品对此有不同的要求，例如，有的产品不允许 IP 子网之间相互包含，而有的产品则允许其任意包含。这种策略只与产品的需求相关，而与 IP 地址的组织结构没有关系。对于冲突检测就是检查新配置的 IP 地址和子网是否与以前曾经配置过的 IP 地址和子网之间存在可能导致歧义的关系。例如新配置的 IP 地址与以前的 IP 地址完全一样，它违背了 IP 地址的唯一性；或者在不同接口上配置了具有相同子网的 IP 地址，使得相同子网跨越了多个不同接口，从而导致进行 IP 路由最长匹配查找时无法得到唯一结果，等等。实际配置 IP 地址时不但需要指明 IP 地址本身，而且还需要规定它所在的子网掩码，例如 255.255.0.0，它是由连续的“1”和“0”组成的与 IP 地址等长的数值，通过 IP 地址与掩码相“与”即可得到所谓的 IP 子网。理论上说，具有相同 IP 子网的地址所代表的主机在物理上应该处于一个连通的二层广播域中，即好象它们是通过一个共享介质的 Ethernet 相连一样。因为只有这样现在通用的 IP 最长匹配的查表算法才能根据目的地址正确地转发报文。由此可见，正是有了上述的种种限制，使得在配置 IP 地址时如果不加冲突检测，就可能由于配置的错误导致系统转发的异常，特别是在配置复杂时更是如此。而增加冲突检测机制实际上就是通过系统内置的检查机制对几种常见的配置错误进行检查，在用户没

有意识到错误时给用户以提示，预防错误的发生。

所谓的 IP 地址的定位，是指从大量的 IP 地址中找出与当前配置地址相关的地址集合，只需要在这个地址集合中检测冲突即可。确定这个地址集合与冲突检测的策略没有关系，而与地址的组织结构则密切相关。其实，这种定位 IP 地址的操作不但在冲突检测中用到，而且在系统运行过程中象路由协议的启动，静态路由的生成等这样的操作也都需要频繁的使用，所以其效率的高低将直接关系到系统的性能。特别对于交换机产品，因为它支持 VLAN 接口，而 VLAN 接口数目很多（2—4094 个），每个 VLAN 接口上最多可以配置 10 个 IP 地址，这样 IP 地址的数目将可能很大。如果不能从大量的地址中将那些与新配置的地址毫无关系的地址剔除掉，则定位 IP 地址的过程将是低效的。只有剔除更多无关的地址才能最大限度的缩小搜索的范围，从而提高效率，减少无用操作。

现有技术一的技术方案：

解决冲突检测和检索地址的一个最直接的方法就是遍历全局地址链，即将本设备配置的所有的 IP 地址全部串在一跟链表上。如图 1 所示：配置 IP 地址时，首先沿着这条链表遍历其中每个节点，对每个节点使用特定的冲突检测策略进行判定。发现冲突，则立刻返回并提示用户出错，否则就在这个链表的末尾增加一个节点，其中保存配置的 IP 地址和子网掩码。这个链表有一个表头节点。一开始其前后向指针都指向自己，随着地址的不断加入，这个链表不断增长，但总是一个双向循环链表，最后一个节点的后继指针指向表头节点，所以插入和删除都很方便。

在该方案中，IP 地址是按照配置的先后顺序加入链表的，没有考虑 IP 地址之间内在的相互关系，因此定位 IP 地址时无法剔除那些不相关的地址，只有将当前已经配置的所有 IP 地址都遍历一遍，对其中的每个 IP 地址都要按照预先定义的冲突检测策略进行判断，因此执行效率很低，当频繁使用这个方法定位 IP 地址时将非常耗时。

现有技术二的技术方案：

要想实现快速定位，可以参照数据库中的做法——按照数据某种联系建立索引结构。通过搜索按地址之间关系建立的索引，可以快速查到相关的地址集合，并在此基础上判定该 IP 地址与新配置的 IP 地址之间是否存在冲突。由此可见，建立索引结构有以下两个方面的需求：（1）索引结构的搜索功能强大，定位速度快；（2）索引结构的维护尽量简单，时间与空间开销小。平衡二差树（AVL）就是这样一种索引结构。

平衡二差树（AVL）一个显著的特点是 AVL 中任何一个子树（包括其本身）的左右子树的高度差的绝对值不超过 1。这样在相同叶子节点的情况下，树的深度是最小的，也就是说，查询的次数是最少的，其时间复杂度为 $O(\log_2 N)$ 。现在已有方案与标准的平衡二差树的基础上改进后的方案。标准的 AVL 算法只要求左右子树的高度差不超过 1，但在叶子节点为偶数时，左右子树的高度将会差 1，相应地沿不同子树的搜索路径也会存在微小的差异。而改进后 AVL 算法能够作到左右子树完全对称，无论沿什么路径走过的深度是一样的。其实，改进后的 AVL 严格说来应该算作三差树，即它在左右子树的中间还有一个指针，专门用来存放 IP 地址个数为偶数时多出的一个 IP 地址。这样当 IP 地址为奇数时，则是一个左右子树高度差为 0 的标准二差树；而当个数为偶数时，将多余的一个 IP 地址挂在中间的一个指针上以平衡左右子树。

参阅图 3，无论是标准的 AVL 还是改进的 AVL，它们的一个特点就是只能进行数值的精确比较，对于 IP 地址而言只能进行 32 比特的精确匹配，不能象查路由表那样进行最长匹配。因此，如果只想确定新配置的 IP 地址本身是否已经配置，使用 AVL 树可以很方便的查找到。但由于 IP 地址的冲突检测的策略中除了上面提到的 IP 地址本身不能相等外，IP 子网也不能相同。所以在比较了 IP 地址的 AVL 树之外，还需要搜索 IP 子网的 AVL 树。因为，IP 子网掩码地取值从 1 到 31（实际上最大只能取到 30），相应地，在内部除了保存 IP 地址地 AVL 树之外还需要保存 31 棵子网的 AVL 树。实际检索时，先不考虑子网掩码用 IP 地址检查地址的 AVL 树，如果没有发现冲突，则将 IP 地址与掩码向“与”取出网络部分，根据掩码的长度到相应的子网 AVL 树中搜索是否存在相同子网，如果

也找不到相同的节点，则认为不冲突。也就是说，判定一个 IP 地址是否冲突，最多需要查询两次 AVL 树，参阅图 3 所示流程图。因为，这种方案的任何左右子树的高度差为 0，因此在增加或删除一个 IP 地址时，可能涉及整个 AVL 树的变动，而每级子树的变动算法相同，因此，实现时采用递归的方法。具体过程如下：

(1) 插入时，如果新增加的 IP 地址比当前节点的数值相等，则表明该节点在 AVL 树中已经存在不必插入，立刻返回，否则转 (2)

(2) 如果本节点存在中间节点，转 (5)；否则转 (3)

(3) 如果新增 IP 地址比当前节点数值小，转 (4)，否则用右子树节点为根节点递归调用本函数。递归返回后，再从右子树中摘除最小数值的 IP 地址，将其放到当前根节点上，而将根节点中的 IP 地址挂到中间节点上。结束

(4) 用左子树节点为根节点递归调用本函数。递归返回后，再从左子树中摘除最大数值的 IP 地址，将其挂到中间节点上。结束

(5) 如果新增 IP 地址比当前节点数值小，转 (6)，否则用右子树节点为根节点递归调用本函数。递归返回后，用左子树节点为根节点再次递归调用本函数将中间节点插入左子树中，结束

(6) 用左子树节点为根节点递归调用本函数。递归返回后，将中间节点放到当前根节点上，而将当前节点上的 IP 地址插入右子树中，结束。

就单个的数值比较而言，AVL 树无疑是最快速的。但因为 AVL 树的本质只能进行数值的精确匹配，而不能进行包含关系的比较，而 IP 地址本身是一种层次化的编址方案，它天生就具有相互包含的关系。因此，用 AVL 树进行 IP 地址的冲突检测必然会存在一些问题。例如，(1) 它除了比较 IP 地址本身之外，还需要用 IP 子网再次查找 AVL 树以确定是否存在一致的子网；(2) AVL 树只能检测出 IP 地址相等和子网相等，而不能检测出子网包含关系，而对于某些产品而言，可能它就是不允许子网包含的。另外，AVL 树没有利用 IP 地址之间的包含关系，使得它在支持一些新的需求方面显得力不从心，例如如果想要检索出本设备在某个范围内配置的所有 IP 地址等。从上面维护 AVL 树的定位过程可以

看出，那是一个复杂的递归过程，每增加或删除一个 IP 地址可能带来整个 AVL 树的变动。而且，它需要同时维护一个 IP 地址的 AVL 树和 31 棵子网的 AVL 树，无论从时间还是空间方面的开销都是较大的。当 IP 地址数目不大时，还可以承受，但对于象交换机等支持 VLAN 接口的产品而言，开销非常巨大。其根本原因也是将 IP 地址与子网割裂开来，导致不能同时维护 IP 地址及其子网。

发明内容

针对现有技术的不足，本发明的目的在于提供一种检索 IP 地址的方法。

本发明的技术方案：一种检索 IP 地址的方法，根据提供的当前 IP 地址至少从 IP 地址集中搜索相关的 IP 地址，所述 IP 地址集中的 IP 地址形成全局地址链表，该方法包含下述步骤：

a、获得待处理的 IP 地址的掩码长度；

b、根据节点对应的配置的 IP 地址的掩码长度按递升顺序形成树索引结构，从所述树索引结构的根节点开始，将节点代表的掩码长度与当前 IP 地址的掩码长度进行比较，如果找到第一个大于或等于当前 IP 地址的掩码长度的当前节点，则继续步骤 c；否则转步骤 e；

c、比较当前 IP 地址的子网与当前节点对应的 IP 地址的子网是否相同，如果相同则进行步骤 d，如果不相同则进行步骤 e；

d、取得当前节点的头指针和尾指针之间从全局地址链表中所截取的所有 IP 地址，所述所截取的 IP 地址被当前节点对应的 IP 地址的子网所包含，并采用判断冲突的策略对每一个 IP 地址进行判断；如果发现冲突则提示并结束检索，否则进行步骤 e；

e、确认当前 IP 地址与现有 IP 地址不冲突和将当前 IP 地址加入到所述全局地址链表中。

一种检索 IP 地址的方法，从 IP 地址集中搜索到当前 IP 地址后删除，所述 IP 地址集中的 IP 地址形成全局地址链表，该方法包含下述步骤：

a、获得当前 IP 地址的掩码长度；

b、根据节点对应的配置的 IP 地址的掩码长度按递升顺序形成树索引结构，从所述

树索引结构的根节点开始，将节点代表的掩码长度与当前 IP 地址的掩码长度进行比较，找到当前 IP 地址的掩码小于或等于节点代表的掩码长度的当前节点；

c、比较当前 IP 地址与当前节点对应的 IP 地址的子网是否相同，如果相同则进行步骤 d，如果不相同则提示没有当前 IP 地址并结束；

d、取得当前节点的头指针和尾指针从全局地址链表中所截取的所有 IP 地址，所述所截取的 IP 地址被当前节点对应的 IP 地址的子网所包含，从这些 IP 地址中查找当前 IP 地址，如果找到则删除该当前 IP 地址，并进行步骤 e；如果未找到则提示无待删除的 IP 地址并结束；

e、根据全局地址链表中当前 IP 地址中保存的回指索引树内部节点的指针找到相应的节点，并对该节点进行处理。

其中：所述的树索引结构为一种能够实现最长匹配的二叉树的数据结构。

所述全局地址链表按 IP 地址的包含关系排序，树索引结构中节点的头指针和尾指针从该全局地址链表中所截取的所有 IP 地址被该节点所代表的 IP 子网所包含。

本发明通过组织 radix 树和 IP 地址链表，可以很方便地定位需要的 IP 地址集合。因为搜索时，先查询 radix 树，它的时间复杂度为 $O(\log_2 N)$ 。无论配置多少个 IP 地址，最多只需要查询 32 次即可找到需要的 IP 地址结构。给出任何一个 IP 子网就可以快速找出当前系统中所有被其包含的 IP 地址集合，而不必搜索整个地址链。这一点是 AVL 树所不具备的。

AVL 树不能提供查询 IP 子网之间包含关系的功能。如果某个产品不允许 IP 地址相互包含的话，则 AVL 树就无能为力了。而 radix 树本身就是用包含关系组织的，越靠近根节点代表的子网范围越大，父节点子网必然包含子节点对应的子网。因此，本发明自然就支持检测 IP 地址的包含关系。这样，通过查询一棵树就可以同时完成 IP 地址检测和 IP 子网检测以及子网包含的多种功能，而不象 AVL 算法要查询多棵树。

在维护索引结构方面，在增加和删除 IP 地址时维护一棵 radix 树的开销比 AVL 算法的维护两棵树要低。而且，仅就维护一棵树的开销来说，radix 树也比

AVL 树要低。这是因为 radix 树并不要求树结构的完全对称，它修改的范围仅仅局限在一个节点的上下两层而与树中其他节点没有关系，因此 radix 树的算法是一个不断向下搜索的单向过程；而 AVL 树的结构则要严格的多，修改一个节点可能导致整棵树结构的变动，因此 AVL 树需要通过一个递归过程完成各层节点的修改。增加 IP 地址时，radix 树最多需要遍历 32 层节点就可以找到插入节点的地方，然后更新上下两级节点的指针即可，而不象 AVL 的递归算法那样，增加一个节点需要考虑给上下各个层面上节点带来的变化，虽然不一定所有节点都变化，但至少需要将整棵树全部遍历一遍。删除 IP 地址时，因为在地址结构中有回指 radix 树节点的指针，因此根本就不需要搜索 radix 树就可以确定需要删除的内部节点，同样删除一个内部节点也只需要修改上下两级节点的指针即可。

附图说明

图 1 为现有技术一的技术方案的链表结构示意图；

图 2 为现有技术二的技术方案的索引结构示意图；

图 3 为现有技术二检测 IP 地址的流程图；

图 4 为交换式路由器结构框图；

图 5 为本发明中 IP 地址的全局链表和索引结构关联示意图；

图 6 为本发明的流程图。

具体实施方式

参阅图 4 所示，为了提高路由器的转发速率，交换式路由器采用了转发和控制相分离的结构。流量从入接口卡进入后由网络处理器或 ASIC 硬件电路查询转发表后经过交换网交换到出接口卡上转发出去，这一切工作大部分由硬件完成，不需要路由处理部件的参与，可以达到很高的转发速率。路由处理部件中运行着路由器的主控软件，主要负责接收用户的配置命令，通过路由协议生成转发表项下发到网络处理器或 ASIC 硬件电路中供转发使用。而要生成转发表

项的首要前提就是给路由器的接口配置 IP 地址和掩码,其它路由器配置的 IP 地址被本设备通过路由协议学到就形成了到达目的网段的转发表项。IP 地址配置的正确与否直接关系到流量能否正确地到达目的地,因此配置 IP 地址时需要做一些检测工作,以防新配置的 IP 地址与原有的 IP 地址发生冲突,将问题尽量控制在源头上,以免将错误的信息扩散到整个网络中引起混乱。

IP 地址与掩码有密切的关系,判定冲突检测时,不但需要判定 IP 地址是否相同,而且还需要判定子网是否相同,甚至某些产品还不允许子网的包含以及更为复杂的策略。而这些策略与采用哪一种索引结构无关,只是需要对所有可能的 IP 地址都进行上述的策略判断,不要遗漏即可。

参阅图 5,虚线中的部分就是本发明的树索引结构,该树为一种可以实现最长匹配的二叉树的数据结构(简称 radix 树),用于快速定位 IP 地址及其集合。而全局地址链表中的 IP 地址按照其包含关系排序。

radix 树是这样组织的:根据配置的 IP 地址的掩码长度,在 radix 树中插入一个内部节点,其中的 KEY 值就是 IP 地址掩码的长度。同时将该 IP 地址对应的数据结构按照 IP 子网的大小插入全局地址链表中,并用该内部节点上的头指针 begin_ptr 和尾指针 end_ptr 同时指向全局地址链表中的 IP 地址结构。这两个指针从全局地址链表上截取的一段地址集合全部都被该内部节点所代表的 IP 子网所包含,即这段 IP 地址的掩码都比该内部节点指定的 KEY 值长,而且它们 IP 地址通过 KEY 值代表的掩码相“与”后的网络部分都相同。例如有一个内部节点所代表的 IP 子网为 10.1.0.0/255.255.0.0,则其 KEY 值为 16,那么它的头指针 begin 和尾指针 end 指针括住的 IP 地址都是如下形式, 10.1.X.X/255.255.X.X, 其中 X 可以为 0,也可以不为 0,也就是所谓的被子网 10.1.0.0/255.255.0.0 所包含。

检索时通过搜索 radix 树可以快速定位到内部节点,再通过内部节点中的头指针 begin_ptr 和尾指针 end_ptr 指针可以直接获取需要的 IP 地址结构。另外,为了避免删除 IP 地址时再次搜索 radix 树,在 IP 地址结构中增加一个反向指向 radix 树内部节点的指针,这样删除时通过该指针可以直接找到需要删除的内部

节点,如果该节点上没有其它相同子网的 IP 地址时就可以直接删除内部节点(删除内部节点的详细过程在下面介绍)。如果配置了多个掩码长度相等,且子网相同的 IP 地址时,因为它们都回指向 radix 树中同一个内部节点,所以将它们插在全局链表中紧邻的位置上,头尾位置分别由 radix 树内部节点的头指针 begin_ptr 和尾指针 end_ptr 指定。例如在 ethernet1 上配置了 10.1.1.1/255.255.255.0 的主 IP 地址,然后又在同样接口上配置了 10.1.1.2/255.255.255.0 的从 IP 地址,则这两个 IP 地址中的反向指针指向同一个内部节点,它们按插入的先后顺序排在全局链表的相连位置上,头指针 begin_ptr 指向 10.1.1.1/255.255.255.0,尾指针 end_ptr 指向 10.1.1.2/255.255.255.0。

在被头指针 begin_ptr 和尾指针 end_ptr 括住的 IP 地址集合不但整体上有这样的特点,而且在其内部也是按照一定的规则排序的。重新排序后的 IP 全局链表有这样的特点:任何一个节点左右指针括住的 IP 地址集合有如下的三个子集组成:[根节点上配置的所有 IP 地址][左子树中节点配置的 IP 地址][右子树节点配置的 IP 地址]。其中所谓[根节点上配置的所有 IP 地址]就是上面提到的配置在根节点上的掩码相等、子网相同的地址链表;而所谓的[左右子树配置的 IP 地址集合]就是分别以左右儿子节点为根形成的子树所截取 IP 地址段,它们也是按照这样的顺序组织,也就是说这个关系是一种递归的顺序,直到叶子节点为止。例如,当前系统中包含以下三个 IP 地址: 10.1.1.1/255.255.0.0、10.1.127.1/255.255.255.0 和 10.1.129.1/255.255.255.0。那么 radix 树中就存在三个节点,根节点 KEY 值为 16,代表 10.1.1.1/255.255.0.0;左节点的 KEY 值为 24,代表 10.1.127.1/255.255.255.0;而右儿子的 KEY 值也是 24,代表 10.1.129.1/255.255.255.0。这样,根节点的头指针 begin_ptr 和尾指针 end_ptr 所截取的 IP 地址链即按如下顺序组织: [10.1.1.1/255.255.0.0] [10.1.127.1/255.255.255.0...] [10.1.129.1/255.255.255.0...]

当然,这个集合不一定完全包含这三个部分,例如,如果当前节点没有左子树,则没有第二部分;没有右子树,则没有第三部分;而根节点上没有配置 IP 地址,则没有第一部分;左右子树都没有时,则 begin_ptr 和 end_ptr 就指向配置的

掩码等于内部节点KEY值的IP地址上，如果这样的IP地址只配置了一个，则begin_ptr和end_ptr指针都指向该IP地址。相应地，若radix树的某个部分被删除了，节点的左右指针也需要做相应调整。例如将一个节点上配置的所有IP地址删除后，该节点的头指针begin_ptr就需要重新调整为其左儿子或右儿子的头指针begin_ptr。

检测冲突的过程也就是定位该IP地址在radix树中具体位置的过程，检测完成后其实就已经找到了插入radix的准确位置。它是以欲检测冲突的IP地址为操作对象，沿着从radix根节点向叶子节点的方向持续推进。每经过一个radix节点都可以得到一个该节点指定的KEY值，将这个KEY值与当前操作的IP地址的掩码进行比较。如果KEY值大于等于当前操作的IP掩码而且该radix节点上配置了IP地址，则标明当前操作的IP地址肯定应该插在该radix节点的上面（因为radix树自上往下，KEY值是不断递增的），不再需要再往下推进了；否则以这个KEY值为序号（该序号从0开始）检测操作对象上对应比特的数值。如果该比特为0，则继续向当前radix的左子树进发；否则向右子树进发。就这样，在不断的推进和检测中直至走到叶子节点为止。例如，配置如下的IP地址10.1.1.1/255.255.255.0。在当前的radix树中有四个节点，它们的KEY值分别为8、16、24和32。这样在推进的路径上首先遇到KEY值为8的节点，比较后发现KEY值（8）比当前地址的掩码（255.255.255.0）小，则表明需要继续推进。然后以KEY为序号检测第9个比特（因为KEY值是以0开始的，因此实际是第9比特），结果发现是0，则沿着左子树进发。第二个节点依次类推，直到第三个节点。此时它的KEY值也是24，与IP掩码相等而且配置了IP地址，则不再往下寻找。接下来就是比较当前操作的IP地址与第三个节点对应的IP地址逐个比特进行比较，找出第一个不同比特所在的序号diff_bit，该序号diff_bit<=MIN（当前IP地址掩码，当前节点的KEY值）。根据序号diff_bit的不同数值，需要进行不同的处理。参阅6所示具体流程如下：

步骤10：获得配置的当前IP地址的掩码长度；

步骤20: 取radix树的根节点为当前节点;

步骤30: 如果配置的子网掩码比当前节点KEY值小或相等, 则转步骤40; 否则判断当前配置的IP地址在当前节点KEY值所指定的比特处取值。如果为1, 则取右子节点为当前节点; 否则取左子节点为当前节点, 然后在转步骤30重新比较, 直至转步骤40;

步骤40: 比较当前节点对应的IP地址与新配置的IP地址同掩码相“与”后得到的子网部分是否相等, 如果相等转步骤50; 否则找出第一个不等的比特diff_bit, 转步骤60, 此时表明新配置的IP地址所形成的子网没有包含任何已有的子网, 但仍然可能被更大的子网所包含, diff_bit就是当前IP地址插入radix树时需要分叉的内部节点的KEY值。

步骤50: 表明以前配置相同子网的IP地址, 此时只需要在该节点的头指针begin_ptr与尾指针end_ptr括住的地址集合中查找, 对其中的每个IP地址调用判断冲突的策略函数即可。如果发现冲突, 提示IP地址配置错误并返回; 在比较完集合中的IP地址后没有发现冲突, 则可以认为该地址合法。

步骤60: 根据第一个不等比特和新配置的掩码与当前节点比特的关系, 可以分成三种情况:

a、如果不等的比特diff_bit等于当前节点KEY值, 则将新配置的IP地址作为当前节点的子树;

b、如果不等的比特diff_bit等于新配置IP地址的掩码长度, 则将当前节点作为新配置的IP地址的子树;

c、如果不等比特diff_bit比当前节点KEY值和新配置IP地址的掩码长度都小, 则新建一个节点作为分叉节点, 当前节点与新配置的IP地址互为兄弟, 都作为分叉节点的左右子树。

无论哪一种情况, 都需要根据不同检测冲突的策略自下而上的检测各级父节点的IP地址是否与新配置的地址冲突。例如, 如果允许子网包含, 则只需要检测IP地址本身是否相同即可, 否则除了IP地址本身外还需要检测IP子网是否相同

或包含。

从上面的结构可以看出，radix树的出发点是维护IP地址之间的包含关系，它提供了最为详细的系统内IP地址之间的内在联系。无论各个产品的冲突检测有什么特殊需求，只需要改变策略函数即可，而不必改变radix树的结构。这主要是因为IP地址是一种层次化的地址，本身就隐含着这种包含关系，这与某些地址系统中非层次化的编址方法是不同的。在IP地址族更加关注IP子网而不是IP地址，因为可以将单个的IP地址看作是具有32位全1掩码的IP子网，因此Internet中也将IP子网作为其逻辑组成的基本单位，每增加一个网络时就需要新分配一个IP子网的网段。单单就IP地址本身而言是难以准确描述它所代表的全部含义的，只有通过掩码形成不同的子网结构才能发挥IP地址的巨大用处，例如路由表中的路由就是通过IP地址加掩码才唯一确定的。radix树正是符合了这一点才能适应各个不同的需求，而AVL却正是没有考虑IP地址与掩码之间的联系，只单纯地比较两个数值的大小才使得支持不同产品的需求变得很困难。

检查无冲突时，就需要将新配置的IP地址根据掩码长度插入到radix树中，其方法与上面提到的冲突检测的方法基本相同，这里不再赘述，同时根据其对应的radix内部节点中的指针begin_ptr和指针end_ptr位置将其插入到全局地址链表中。

删除一个IP地址时需要做相反的动作，将IP地址从全局地址链表和radix树摘除。从全局链表是一个双向链表，摘除其中一个节点非常简单；而从radix树上摘除首先需要找到对应的内部节点，此时就需要用到IP地址中保存的回指radix树内部节点的指针。根据该指针立刻可以找到需要删除的节点，然后分为三种情况分别处理：

- 1) 如果节点同时存在左右子树，则本节点需要保留作为分叉节点，可以立刻返回

- 2) 如果节点左右子树都没有，则删除本节点。将父节点的指向本节点中指针清空。如果此时父节点还有IP地址与之相连，则父节点需要保留立刻返回，否

则取父节点为当前节点，转3)；

3) 如果只有左子树或只有右子树，则删除本节点，同时将左子树或右子树挂到本节点在其父节点的相应指针上。

无论是上述哪一种情况，在处理完内部节点后，需要检查是否需要移动父节点的头指针begin_ptr和尾指针end_ptr的指针。究竟是否需要移动可以参照下面的原则进行：

(1) 如果待删除的IP地址既没有被内部节点的头指针begin_ptr指向也没有被尾指针end_ptr指向，则无须调整这两个指针，即可返回。

(2) 如果仅仅被头指针begin_ptr指向，则表明该IP地址是其所在子网的第一个IP地址，此时只需要将尾指针begin_ptr指针移向后一个IP地址。如果当前节点位于其父节点的左子树上，则将父节点作为当前节点重复执行该原则，否则直接返回。

(3) 如果仅仅被尾指针end_ptr指向，则表明该IP地址是其所在子网的最后一个IP地址，此时只需要将尾指针end_ptr移向前一个IP地址。如果当前节点位于其父节点的右子树上，则将父节点作为当前节点重复执行该原则，否则直接返回。

(4) 如果同时被头指针begin_ptr和尾指针end_ptr指向，则表明它是该子网中配置的唯一一个IP地址，此时需要将内部节点一并删除，并调整其父节点的首尾指针。此时将父节点设为当前节点，再次使用该原则即可。

由此可见，无论插入还是删除都很简单，最多只需要修改父子两级节点中的指针，而不涉及其他节点，从而将修改限制在一个很小的范围之内，这与AVL树的递归算法又很大的不同，这也是radix算法的优点之一。

radix树本身就是一种层次化的结构，特别适合组织象IP地址这样需要最长匹配的地址结构。但radix树的算法只定义了树的内部节点，而对树的外部节点没有规定。这就给radix树的使用者以极大的灵活性来组织外部数据。本方案通过头指针begin_ptr和尾指针end_ptr两个指向全局链表的指针将一个子网中包含

的所有IP地址囊括在内，从而实现快速定位相关地址集合的目的，相应地遍历相关地址集合中的IP地址时只需要扫描头指针begin_ptr和尾指针end_ptr定义的一段链表即可。除本方法之外，还可以存在别的方法组织radix树的外部节点。例如，可以在全局IP地址链表之外，另外分配结构保存IP地址和掩码，同时将具有相同子网号的这些数据结构串成一个链表挂到radix的内部节点上。这样的方法同样可以完成地址冲突检测功能，但需要占用额外内存，而且遍历相关地址集合中的IP地址时需要采用遍历radix树的方法，比本方案中的遍历双向链表的方法复杂。

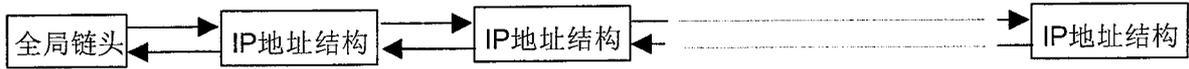


图 1

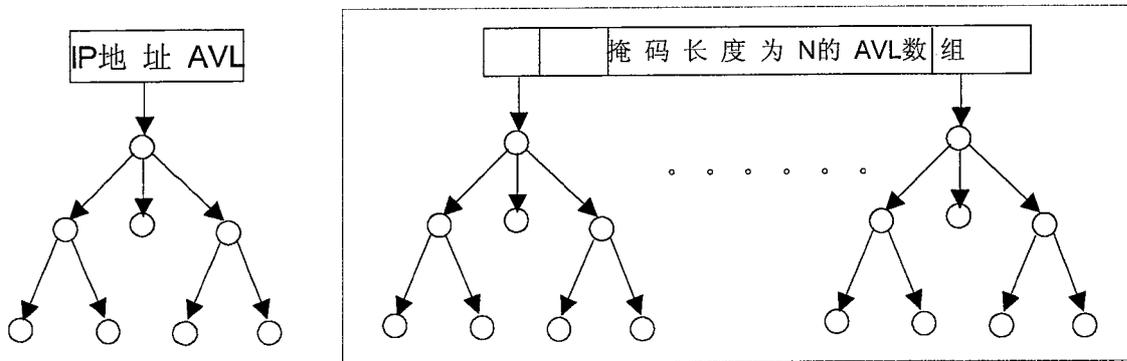


图 2

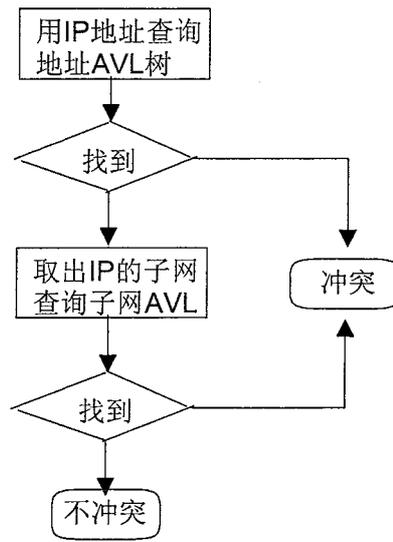


图 3

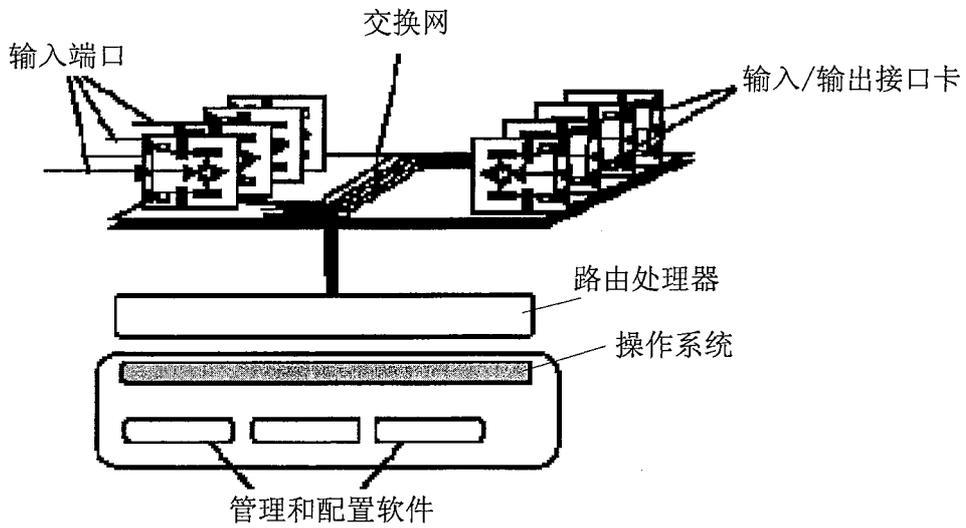


图 4

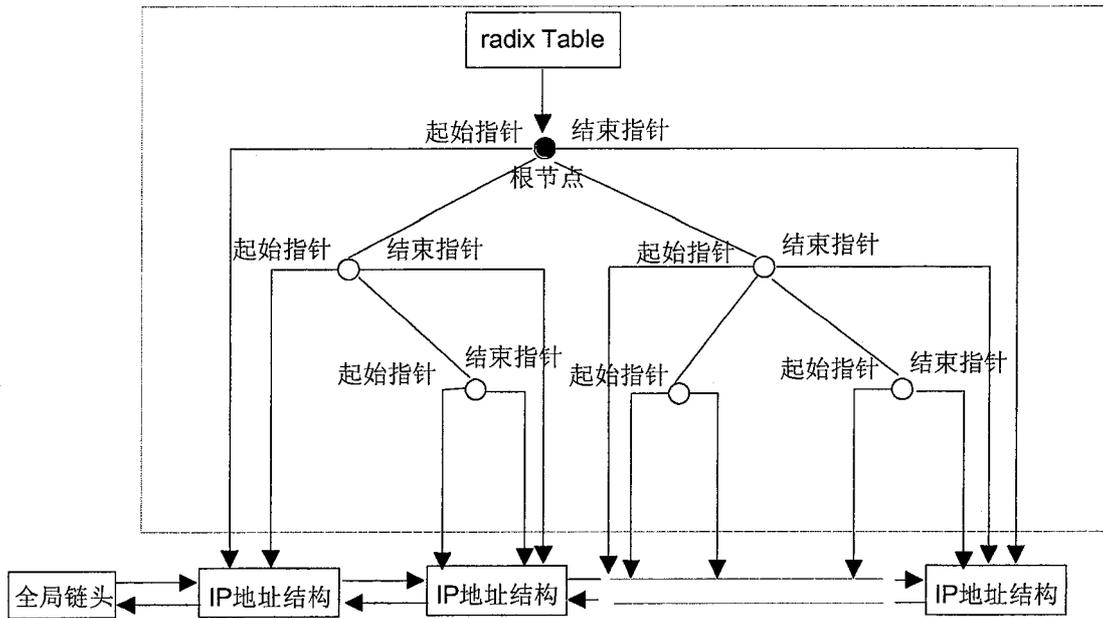


图 5

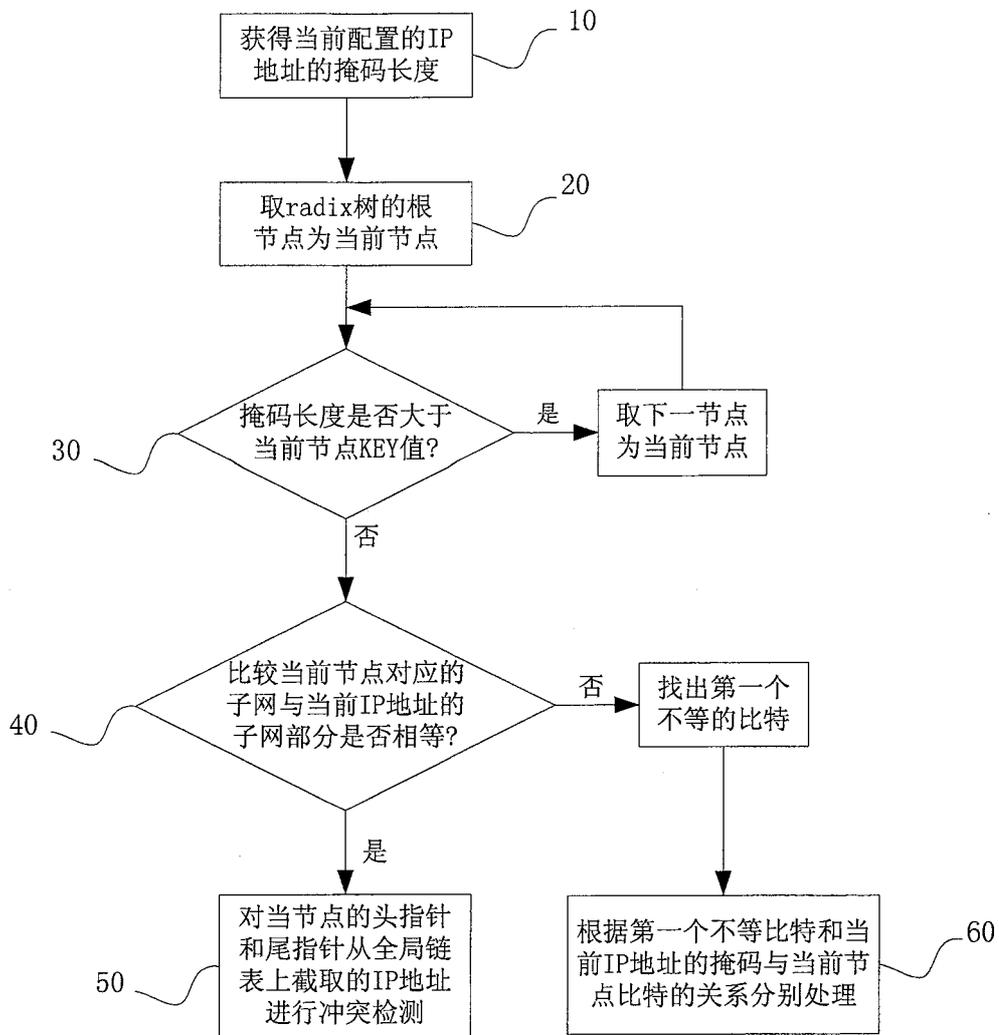


图 6