高效率IP路由查找实验

2020年11月23日

蔡润泽

本实验 `Github` 地址

实验内容

实验内容一

实现最基本的前缀树查找

实验内容二

调研并实现某种IP前缀查找方案

测试与验证

- 基于forwarding-table.txt数据集(Network, Prefix Length, Port)
- 本实验只考虑静态数据集，不考虑表的添加或更新
- 以前缀树查找结果为基准，检查所实现的IP前缀查找是否正确
  - 可以将forwarding-table.txt中的IP地址作为查找的输入
- 对比基本前缀树和所实现IP前缀查找的性能
  - 内存开销、平均单次查找时间

设计思路

最基本的前缀树查找

`RouterEntry* line_parser (char * line)` 函数

负责将txt文件中读取的每一行字符串，转换成路由表项，返回 `RouterEntry*` ，每一个路由表项结构包括三个变量，分别是网络号，掩码长度以及转发端口号。

`int net_parser(char * s)` 函数

负责将网络号从字符串类型，转换成一个 `int` 类型。

`TreeNode * init_tree()` 函数

初始化树结构，即建立一个树的根节点，并返回根节点。

`int add_node (RouterEntry* entry)` 函数

该函数根据需要插入的路由表项，依次建立起前缀树。具体的实现如下：

```
int add_node (RouterEntry* entry) {
    TreeNode * ptr = root;
    int mask = 0x80000000;
    unsigned int prefix_bit = 0x80000000;
    int insert_num = 0;
    for (int i = 1; i <= entry->prefix_len; i++) {
        if ((entry->net&prefix_bit) == 0) {
            if (ptr->left == NULL) {
                ptr->left = insert_tree(entry, i, ptr);
                insert_num ++;
            }
            ptr = ptr->left;
        } else {
            if (ptr->right == NULL) {
                 ptr->right = insert_tree(entry, i, ptr);
                 insert_num ++;
            }
            ptr = ptr->right;
        }
        mask = mask >> 1;
        prefix_bit = prefix_bit >> 1;
    }
    if (insert_num == 0) {
        printf("net:%x update -> port: %d\n ", entry->net&mask, ptr->port);
    }
    if ((entry->net&mask) != (ptr->net&mask)) {
        return -1;
    }
    total_mem -= sizeof(RouterEntry);
    free(entry);
    return ptr->port;
}
```

`TreeNode * insert_tree(RouterEntry* entry, int p_len, TreeNode* parent)` 函数

该函数被`add_node`函数所调用，在建立前缀树的过程中，若所需的节点不在树中，则建立该节点。对于内部节点，端口设置为`-1`来进行表示。

树在建立的时候存在双向路径，即存在父节点保存到子节点的指针，子节点也保存到父节点的指针，以便于后续的回溯查找。该函数具体实现如下：

```
TreeNode * insert_tree(RouterEntry* entry, int p_len, TreeNode* parent) {
    TreeNode * treeNode = (TreeNode *) malloc(sizeof(TreeNode));
    total_mem += sizeof(TreeNode);
    if (treeNode == NULL) {
        exit(0);
    }
    treeNode->net = entry->net & get_mask(p_len);
    treeNode->prefix_len = p_len;
    if (entry->prefix_len != p_len) {
        treeNode->port = -1;
    } else {
        treeNode->port = entry->port;
    }
    treeNode->parent = parent;
    return treeNode;
}
```

`int lookup (int netID)` 函数

该函数负责通过网络号根据前缀树进行转发端口的查找。

若查询到的端口号为`-1`则表明查询路径最终落在了中间节点上，此时通过调用`look_back`函数进行回溯查找，找到最近的最长前缀匹配的网络号，并返回该网络对应的转发端口。具体的实现如下：

```
int lookup (int netID) {
    TreeNode * ptr = root;
    unsigned int prefix_bit = 0x80000000;
    int i;
    int port;
    for (i = 1; i < 32; i++) {
        if ((netID & prefix_bit) == 0) {
            if (ptr->left == NULL) {
                if ((netID & get_mask(ptr->prefix_len)) != (ptr->net & get_mask(ptr->pref
                    port = -1;
                    break;
                }
                port = ptr->port;
                break;
            }
            ptr = ptr->left;
        } else {
            if (ptr->right == NULL) {
                if ((netID & get_mask(ptr->prefix_len)) != (ptr->net & get_mask(ptr->pref
                    port = -1;
                    break;
                }
                port = ptr->port;
                break;
            }
            ptr = ptr->right;
        }
        prefix_bit = prefix_bit >> 1;
    }
    if ((netID&get_mask(i)) != (ptr->net&get_mask(i))) {
        port = -1;
    }
    if(port == -1){
        port = look_back(ptr);
    }
    return port;
}
```

`int look_back(TreeNode * node)` 函数

该函数被`lookup`函数调用，通过子节点到父节点的指针，负责找到最近的最长前缀匹配的网络号，并返回该网络对应的转发端口。

```
int look_back(TreeNode * node) {
    TreeNode * ptr = node;
    while (ptr->port == -1) {
        ptr = ptr->parent;
    }
    return ptr->port;
}
```

Leaf Pushing (改进)方法

在上述的最基本的前缀树查找当中，树节点的数据结构如下：

```
typedef struct treeNode {
    int net;
    int prefix_len;
    int port;
    struct treeNode * left;
    struct treeNode * right;
    struct treeNode * parent;
} TreeNode;
```

存在可以删去的冗余项，而 方法则大大简化了树节点的结构体，改进后的树节点结构如下：

```
typedef struct treeNode {
    int port;
    struct treeNode * left;
    struct treeNode * right;
} TreeNode;
```

上述树节点的结构体删去了前缀掩码长度、以及网络号，并删去了父指针。中间节点也能通过保存最近的前缀匹配的端口，来减少回溯查找的时间。

本设计基于 Leaf Pushing 的改进

传统的叶推算法在树节点中需要记录储存port数据结构的指针，并需要添加更新最近前缀节点指针的函数。在思考了Leaf Pushing的基本思想后，本设计直接将port的数据放在了树节点中，减少了多余的开销。另外也优化了更新最近符合前缀网络号端口的函数，若新添加的节点如果是中间节点，则其直接继承父节点的端口号。

改进后的相关函数如下：

`TreeNode * insert_tree(RouterEntry* entry, int p_len, TreeNode* parent)` 函数

新的`insert_tree`函数删去了保存父指针等操作，而是只保存端口号。倘若某个节点是中间节点，建立的时候需要保存父节点的端口号即可。这样具体路由表项所保存的端口号可以被子节点中的中间节点所继承。

```
TreeNode * insert_tree(RouterEntry* entry, int p_len, TreeNode* parent) {
    TreeNode * treeNode = (TreeNode *) malloc(sizeof(TreeNode));
    total_mem += sizeof(TreeNode);
    if (treeNode == NULL) {
        exit(0);
    }
    if (entry->prefix_len != p_len) {
        treeNode->port = parent->port;
    } else {
        treeNode->port = entry->port;
    }
    return treeNode;
}
```

`int add_node (RouterEntry* entry)` 函数

此函数根据结构体的优化，删去了一些冗余的判断条件，新的实现如下：

```
int add_node (RouterEntry* entry) {
    TreeNode * ptr = root;
    unsigned int prefix_bit = 0x80000000;
    int insert_num = 0;
    for (int i = 1; i <= entry->prefix_len; i++) {
        if ((entry->net&prefix_bit) == 0) {
            if (ptr->left == NULL) {
                ptr->left = insert_tree(entry, i, ptr);
            }
            ptr = ptr->left;
        } else {
            if (ptr->right == NULL) {
                 ptr->right = insert_tree(entry, i, ptr);
            }
            ptr = ptr->right;
        }
        prefix_bit = prefix_bit >> 1;
    }
    total_mem -= sizeof(RouterEntry);
    free(entry);
    return ptr->port;
}
```

`int lookup (int netID)` 函数

该函数删去了回溯查找的部分，中间节点以及保存了最近的最长前缀匹配所对应的端口号，因此只需要进行简单地前缀查找即可。具体实现如下：

```
int lookup (int netID) {
    TreeNode * ptr = root;
    unsigned int prefix_bit = 0x80000000;
    int i;
    int port;
    for (i = 1; i < 32; i++) {
        if ((netID & prefix_bit) == 0) {
            if (ptr->left == NULL) {
                port = ptr->port;
                break;
            }
            ptr = ptr->left;
        } else {
            if (ptr->right == NULL) {
                port = ptr->port;
                break;
            }
            ptr = ptr->right;
        }
        prefix_bit = prefix_bit >> 1;
    }
    return port;
}
```

结果验证

基本前缀树实验结果

实验结果如下：

```
(base) Vincent@RTsAIs-MacBook-Pro EXP11-Lookup % ./lookup
totoal mem: 65863400
time_use is 122.854580001
(base) Vincent@RTsAIs-MacBook-Pro EXP11-Lookup % ./lookup
totoal mem: 65863400
time_use is 113.616628599
```

上图可知，实现基本前缀树所消耗的内存为65863400字节。所需要的查找时间（除去从txt中解析字符串的时间）为118ns。

Leaf Pushing实验结果

实验结果如下：

```
(base) Vincent@RTsAIs-MacBook-Pro EXP11-Lookup % cd leaf_pushing
(base) Vincent@RTsAIs-MacBook-Pro leaf_pushing % ./lookup
totoal mem: 39518040
time_use is 102.700743106
(base) Vincent@RTsAIs-MacBook-Pro leaf_pushing % ./lookup
totoal mem: 39518040
time_use is 116.112752586
```

上图可知，实现基本前缀树所消耗的内存为39518040字节。所需要的查找时间（除去从txt中解析字符串的时间）为117ns。

两者对比发现Leaf Pushing在存在很多路由表项插入的情况下，内存消耗减少40%。但是时间减小的并不明显。这是因为存在很多路由表项来构建前缀树时，前缀树趋于完全，因此可以减少查询的次数不多。相较于通过算法进行改进，通过编译优化进行速度提升效果更高。

Leaf Pushing开启O2编译优化的的结果如下：

```
(base) Vincent@RTsAIs-MacBook-Pro leaf_pushing % gcc -o lookup lookup.c -O2
(base) Vincent@RTsAIs-MacBook-Pro leaf_pushing % ./lookup
totoal mem: 39518040
time_use is 11.470420501
```

可以看到此时通过对控制流和指令排布调度等方面的优化，查找的处理速度提高了一个数量级。