# 高效 IP 路由查找实验

学号： 2021K8009929010

姓名： 贾城昊

## 一、 实验题目：高效 IP 路由查找实验

## 二、 实验任务：

了解 IP 路由查找效率的实际需求,学习经典的前缀树 IP 路由查找算法以及其优化方法,实现最基本的前缀树查找。调研并实现某种高级 IP 前缀查找方案。最后对两种方案的内存开销、平均查找时间进行对比分析。

## 三、 实验流程

1. 实现最基本的前缀树 IP 路由查找。
2. 调研并实现某种高级 IP 前缀查找方案。
3. 对比分析两种方案的内存开销、平均查找时间。

## 四、 实验过程

### （一）基本 IP 前缀树

#### 1. 基本前缀树的构建

基本前缀树的构建基于 forwarding-table.txt 数据集，首先对根节点进行初始化，然后读取 forwarding-table.txt 数据集中每一行的信息，得到 IP 地址，前缀和端口。然

后调用自己编写的 insert_tree_node 函数对基本前缀树进行插入。其中解析出的 IP 地址用一个 32 位的 char 数组存储，每一位代表其二进制对应位的值，从而方便 insert_tree_node 函数进行处理。

```c
// Constructing an basic trie-tree to lookup according to `forward_file`
void create_tree(const char* forward_file){
    //fprintf(stderr,"TODO:%s",__func__);
    root =(node_t*)malloc(sizeof(node_t));
    root->type = I_NODE;
    root->lchild = NULL;
    root->rchild = NULL;

    FILE *file;
    char line[100];

    file = fopen(forward_file, "r");
    if (file == NULL) {
        fprintf(stderr, "open forward file failed\n");
        return;
    }

    while (fgets(line, sizeof(line), file)) {
        char ip[12];
        int prefix;
        int port;

        sscanf(line, "%s %d %d", &ip[0], &prefix, &port);

        int ip_values[4];
        char binaryArray[33];
        sscanf(ip, "%d.%d.%d.%d", &ip_values[0], &ip_values[1], &ip_values[2], &ip_values[3]);

        int index = 0;
        for (int i = 0; i < 4; i++) {
            for (int j = 7; j >= 0; j--) {
                binaryArray[index++] = (ip_values[i] >> j) & 1 ? '1' : '0';
            }
        }
        binaryArray[32] = '\0';

        insert_tree_node(root, binaryArray, prefix, (uint32_t)port);
    }

    fclose(file); // 关闭文件
    return;
}
```

insert_tree_node 函数是针对一个 IP 地址及其对应的前缀和端口对前缀树进行插入的过程。具体操作是根据前缀的大小进行循环，每次获取 IP 地址对应的位的值（从高位到低位），若 IP 对应位的值是 0，则查看当前节点的左孩子节点是否为空，若不为空只用把当前节点改为其左孩子节点即可，否则需要给其左孩子开辟新的空间，并进行初始化（即把节点类型设置为中间节点，把孩子节点设置为空），然后把当前节点改为其左孩子节点即可；若 IP 对应位的值是 1，与上述过程类似，改为判断当前节点的右孩子节点即

可。最后跳出循环后，把当前节点的类型改为匹配节点，并把当前节点的端口改为输入的端口。

具体代码如下所示：

```
void insert_tree_node(node_t * root, char* binaryArray, int prefix, uint32_t port) {
    node_t* current_node = root;

    for(int i = 0;i < prefix;i++){
        if(binaryArray[i] == LEFT){
            if(current_node->lchild){
                current_node = current_node->lchild;
            }
            else{
                current_node->lchild = (node_t*)malloc(sizeof(node_t));
                current_node = current_node->lchild;

                current_node->type = I_NODE;
                current_node->lchild = NULL;
                current_node->rchild = NULL;
            }
        }
        else if(binaryArray[i] == RIGHT){
            if(current_node->rchild){
                current_node = current_node->rchild;
            }
            else{
                current_node->rchild = (node_t*)malloc(sizeof(node_t));
                current_node = current_node->rchild;

                current_node->type = I_NODE;
                current_node->lchild = NULL;
                current_node->rchild = NULL;
            }
        }
        else{
            printf("error\n");
        }
    }

    current_node->port = port;
    current_node->type = M_NODE;
}
```

## 2. 基本前缀树的查找

基本前缀树的查找过程与插入有类似的地方，首先把对应的 IP 输入改为 32 位 char 数组的形式，然后进行 32 次循环，根据 IP 地址的对应位，把当前节点跳转到左孩子节点或者右孩子节点，如果发现要跳转的孩子节点为空，则说明查找结束，跳出循环。并且在每次循环时，需要判断当前节点的类型是否为匹配节点。

由于 CIDR 机制，需要查找最长的前缀匹配，所以用一个变量存储匹配到的节点的端口，在循环中每次匹配到了便更新匹配到的端口，这样最后得到的就是最长前缀匹配对应的端口值。

具体代码如下：

```
uint32_t match_port = -1;
node_t* current_node = root;

for(int k = 0;k < 32; k++){
    if(current_node->type == M_NODE){
        match_port = current_node->port;
    }

    if(binaryArray[k] == LEFT){
        if(current_node->lchild){
            current_node = current_node->lchild;
        }
        else{
            break;
        }
    }
    else{
        if(current_node->rchild){
            current_node = current_node->rchild;
        }
        else{
            break;
        }
    }
}

res[i] =  match_port;
//printf("%d\n", match_port);
```

## （二）高效 IP 前缀树

### 1. 关键数据结构与基本思路

节点查找的基本思路是，首先根据一个 Trie_map[MAP_NUM]数组，用于根据 IP 地址的前 16 位映射到对应的根节点。然后，根据 IP 地址剩下的 16 位，每两位作为索引从根节点搜索对应的孩子节点。最后，返回最后匹配的节点的端口即可。

因此每个节点的需要设置 type，port 和 prefix_diff 位分别用于表示该节点的类型（中间节点还是匹配节点），对应的端口以及前缀的偏移（prefix_diff 的作用在后面详细介绍），同时每个节点还会设置四个孩子节点。

```
typedef struct TrieNodeOpt {
    struct TrieNodeOpt* children[4];
    // Node infos
    bool type;
    int port;
    char prefix_diff;
}TrieNodeOpt;
```

四个孩子节点分别表示 IP 对应两位是 00，01，10，11。由于前缀可能不是 2 的倍数，所以设置 prefix_diff 来表示匹配节点的前缀是奇数还是偶数。

（本查找算法是 IP 地址的 2 位为单位进行查找，所以 prefix_diff 只可能为 0 或者 1。本人曾经尝试以 IP 地址每 4 位进行查找，此时 prefix_diff 便可以表示其模 4 的值，但由于最后发现每 4 位进行查找构建树的话，内存开销太大，所以最后改为了以 2 位为单位进行查找）

**2. 高效前缀树的构建**

在获得 IP 地址，前缀和端口后，首先需要判断前缀地址长度是否小于 16，如果小于 16，只需要对 Trie_map 数组进行操作就行。通过其前缀，获得其 IP 前缀对应的映射项（由于 Trie_map 数组是根据 IP 的前 16 位进行划分，所以当前缀小于 16 的时候，会对应多个映射表项）；否则，需要根据 Trie_map 找到插入的根节点，进行插入操作。

在更新映射表项时，更新的条件是其存储的对应节点的类型是中间节点（代表之

前没有 IP 表项对应该节点）或者其 prefix_diff 大于 16 － prefix（代表之前该节点匹配的 IP 的前缀小于当前匹配的 IP 的前缀，由于最长前缀匹配，需要更新）。代码如下所示

```
    if (prefix >= MAP_SHIFT) {
        TrieNodeOpt *root = Trie_map[(0xffff0000 & binaryIP) >> MAP_SHIFT];
        insert_node_advance(root, binaryIP, port, prefix);
    } else {
        uint32_t mask = (~(0xFFFFFFFF >> prefix));
        uint32_t start = (binaryIP & mask) >> MAP_SHIFT;
        uint32_t end = start + (1 << (MAP_SHIFT - prefix));
        for (int i = start; i < end; i++) {
            if (Trie_map[i]->type == I_NODE || (Trie_map[i]->prefix_diff >= 16 - prefix)) {
                Trie_map[i]->type = M_NODE;
                Trie_map[i]->port = port;
                Trie_map[i]->prefix_diff = 16 - prefix;
            }
        }
    }
}
```

对根节点进行插入操作的步骤，首先根据 IP 的每两位循环查找对应的子节点，如果发现自己的为空，则需要为其分配空间并对其进行初始化。跳出循环后，判断前缀是否为 2 的倍数，若是，则直接修改节点的类型与节点对应的端口（由最长匹配算法可知需要更新）；否则，则对对应两个子节点进行处理，若为空则需要进行初始化，并把 prefix_diff 字段置 1。

具体代码如下：

```c
// insert node to advance_tree
void insert_node_advance(TrieNodeOpt *root, uint32_t ip, int port, int prefix) {
    TrieNodeOpt *curr = root, *next = NULL;
    int off;
    int cur_bit;
    int cur_prefix;

    for (cur_prefix = 32 - MAP_SHIFT; cur_prefix < prefix - 1; cur_prefix += 2) {
        off = 30 - cur_prefix;
        cur_bit = (ip & (0x3 << off)) >> off;
        next = curr->children[cur_bit];
        // If the children node's space type not been allocated.
        if (next == NULL) {
            next = (TrieNodeOpt*)malloc(sizeof(TrieNodeOpt));
            node_advance_num++;
            next->port = 0;
            next->type = I_NODE;
            next->children[0] = next->children[1] = next->children[2] = next->children[3] = NULL;
            curr->children[cur_bit] = next;
        }
        curr = next;
    }

    if (cur_prefix == prefix - 1) {
        off = 30 - cur_prefix;
        cur_bit = (ip & (0x3 << off)) >> off;
        int start_bit = cur_bit & 0x2;
        for(int i = 0;i < 2;i++){
            next = curr->children[start_bit + i]; //最低位设置为0
            if (next == NULL) {
                next = (TrieNodeOpt*)malloc(sizeof(TrieNodeOpt));
                node_advance_num++;
                next->port = port;
                next->type = M_NODE;
                next->children[0] = next->children[1] = next->children[2] = next->children[3] = NULL;
                next->prefix_diff = 1;
                curr->children[start_bit + i] = next;
            }
        }
    }
    else{
        curr->port = port;
        curr->type = M_NODE;
    }
}
```

### 3. 高效 IP 前缀树的查找

查找的操作与插入类似，用一个 match 变量存储最后匹配的节点。获取 IP 值后，根据前 16 位找的对应的根节点，并判断是否匹配。然后根据该根节点，每次根据 IP 地址的两位查找子节点，并判断是否匹配，若匹配，则更新 match。最后返回最后匹配的节点对应的端口值，完成一次查找。

具体代码如下所示：

```c
// Look up the ports of ip in file `ip_to_lookup.txt` using the advanced tree input is read from `read_te
uint32_t *lookup_tree_advance(uint32_t* ip_vec){
    //fprintf(stderr,"TODO:%s",__func__);
    //return NULL;

    uint32_t* res = (uint32_t*)malloc(sizeof(uint32_t) * TEST_SIZE);

    for(int i = 0;i < TEST_SIZE;i++){
        uint32_t ip = ip_vec[i];

        TrieNodeOpt *curr = Trie_map[ip >> MAP_SHIFT];
        TrieNodeOpt *match = unmatch;

        uint32_t curr_bit;
        int cur_prefix = 32 - MAP_SHIFT;

        if (curr->prefix_diff > 0) {
            if (curr->type) {
                match = curr;
            }
            curr_bit = (ip & (0x3 << 14)) >> 14;
            curr = curr->children[curr_bit];
            cur_prefix += 2;
        }

        while(curr) {
            if (curr->type) {
                match = curr;
            }
            int off = 30 - cur_prefix;
            curr_bit = (ip & (0x3 << off)) >> off;
            curr = curr->children[curr_bit];
            cur_prefix += 2;
        }

        res[i] = match->port;
        //printf("%u\n", res[i]);
    }

    return res;
}
```

# 五、 实验结果与分析

## （一）运行正确性与查找时间

运行 main 函数，得到两种方法的耗时对比以及正确性如下：

```
sai@Computer:~/workspace/Network/lab10/10-ip_lookup$ ./ip_trie_tree
Constructing the basic tree......
Reading data from basic lookup table......
Looking up the basic port......
Constructing the advanced tree......
Reading data from advanced lookup table......
Looking up the advanced port......
Dumping result......
basic_pass-1
basic_lookup_time-42330us
advance_pass-1
advance_lookup_time-7035us
```

可以看出两种前缀树的构建与查找正确，并且改进后的 IP 前缀树查找花费的时间远小于基本的 IP 前缀树。

## （二）对时间开销的分析

改进后的 IP 前缀树花费时间远小于基本的 IP 前缀树，这是因为改进后的 IP 前缀树首先通过 IP 前 16 位直接找到对应的根节点，这减少了较多的查找时间。同时，改进和的 IP 前缀树每次根据 IP 的 2 位跳转到子节点，而基本的 IP 前缀树每次只根据 1 位跳转子节点，这导致改进的 IP 前缀树需要查找的路径大大减少，所以查找时间大大减少。

## （三）空间开销对比与分析

对 main 函数进行修改，并添加计算空间开销的函数：分别用两个全局变量记录节点的数量，每次 malloc 时，将对应的全局变量加一，最后统一计算即可，代码如下：

```
unsigned calc_space_advance_trie() {
    return (node_advance_num * sizeof(TrieNodeOpt));
}

unsigned calc_space_trie(){
    return (node_num * sizeof(node_t));
}
```

运行 main 函数，得到空间开销如下：

```
sai@Computer:~/workspace/Network/lab10/10-ip_lookup$ ./ip_trie_tree
Constructing the basic tree......
Reading data from basic lookup table......
Looking up the basic port......
Constructing the advanced tree......
Reading data from advanced lookup table......
Looking up the advanced port......
Dumping result......
basic_pass-1
basic_lookup_time-42766us
advance_pass-1
advance_lookup_time-7530us
basic_space_complexity-52.690720 MB
advance_space_complexity-37.756936 MB
```

可以看到改进后的 IP 前缀树的空间开销也比基本的 IP 前缀树少。这是因为其每次是根据 IP 的两位进行查找，并且前 16 位直接映射，从而导致改进后的 IP 前缀的层数比基本前缀树低的多。并且，改进后的 IP 前缀树节点只比基本的 IP 前缀树多了一个 char 数据，所以每个节点的大小也并不会比基本 IP 前缀树的节点大太多。

## 六、 实验总结

通过本次实验，我了解了实际使用的 IP 路由查找算法，也实现了最基础的前缀树查找方法。之后通过调研也了解到一些高级的查找算法,例如 Poptrie、 Tree Bitmap 等等,并对前缀树进行了改进，加深了我对 IP 路由查找的理解。

本次实验中，本人最初尝试实现 Poptrie，但是起初发现所编写的程序能在本地跑过但不能在 OJ 上跑过，最后发现是不同 gcc 对 memcpy 中未定义行为的处理不同，即 src 和 dest 地址重叠（本地 gcc 版本位 4.1.2，服务器上为 9.4）。但是修改后发现运行时间较长，无法达到 OJ 的要求，具体原因可能是因为运行过程中存在大量的位移操作来获取查找的子节点的索引（每次都需要通过循环获得两个 64 位无符号数的 1 的个数）。出于对运行时间的考虑，本人最后选择了改变方法对前缀树进行改进。

（值得一提的是，本人最初实现的 Poptrie 的内存开销很小，大概只需要 10MB 的空间，比现在的方法小的多，但花费时间要比现在多）