

Report 10 — Novmeber 12

Lecturer: Wu Qinghua

Completed by: 2022K8009929010 Zhang Jiawei

10.1 实验内容

1. 实现最基本的前缀树查找
2. 调研并实现某种优化的 IP 前缀查找方案
3. 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)

10.2 实验过程

10.2.1 基本前缀树查找

基本前缀树查找包括构建前缀树和查找前缀树两个部分。基本前缀树节点的数据结构已经在头文件中给出,我们只需要完成构建和查找的函数即可。

前缀树的构建基于数据库。我读取数据库中每一行的 IP 地址、前缀长度和端口号,然后将其插入到前缀树中。我用一个 32 位的数组来存储读取到的 IP 地址的二进制表示,从高位到低位,每次读取一位,如果该位为 1,则向右子树移动,若右子树为空则申请空间构建右子节点,否则向左子树移动,若左子树为空则申请空间构建左子节点。当读取完前缀长度对应的 IP 地址后,将端口号存储在到达的节点中,且设置节点类型为匹配节点。代码如下:

```
// 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;
    size += sizeof(node_t);

    FILE *fp = fopen(forward_file, "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: cannot open file %s\n", forward_file);
        exit(1);
    }
}
```

```
char line[100];
while (fgets(line, 100, fp) != NULL) {
    char ip_str[15];
    uint32_t port, prefix;
    sscanf(line, "%s %u %u", ip_str, &prefix, &port);
    int ip_dec[4];
    sscanf(ip_str, "%d.%d.%d.%d", &ip_dec[0], &ip_dec[1], &ip_dec[2],
           &ip_dec[3]);

    uint32_t ip_bin[32];
    for (int i = 0; i < 4; i++)
        for (int j = 7; j >= 0; j--)
            ip_bin[i * 8 + 7 - j] = (ip_dec[i] >> j) & 1;

    node_t *cur = root;
    for (int i = 0; i < prefix; i++) {
        if (ip_bin[i] == LEFT) {
            if (cur->lchild == NULL) {
                cur->lchild = (node_t *)malloc(sizeof(node_t));
                cur->lchild->type = I_NODE;
                cur->lchild->lchild = NULL;
                cur->lchild->rchild = NULL;
                size += sizeof(node_t);
            }
            cur = cur->lchild;
        }
        else if (ip_bin[i] == RIGHT) {
            if (cur->rchild == NULL) {
                cur->rchild = (node_t *)malloc(sizeof(node_t));
                cur->rchild->type = I_NODE;
                cur->rchild->lchild = NULL;
                cur->rchild->rchild = NULL;
                size += sizeof(node_t);
            }
            cur = cur->rchild;
        }
        else {
            fprintf(stderr, "Error: invalid ip\n");
            exit(1);
        }
    }

    cur->type = M_NODE;
```

```
        cur->port = port;
    }

    fclose(fp);
    return;
}
```

查找前缀树的过程是根据输入的 IP 地址, 从根节点开始, 根据 IP 地址的每一位向下查找, 直到无法继续向下查找为止。查找的过程中, 如果遇到匹配节点, 则更新查找到的端口号。这样的操作是满足最长前缀匹配的, 代码如下:

```
// Look up the ports of ip in file `ip_to_lookup.txt` using the basic tree, input
// is read from `read_test_data` func
uint32_t *lookup_tree(uint32_t* ip_vec)
{
    // fprintf(stderr, "TODO:%s", __func__);
    uint32_t *port_vec = (uint32_t *)malloc(sizeof(uint32_t) * TEST_SIZE);
    for (int i = 0; i < TEST_SIZE; i++)
    {
        uint32_t ip_bin[32];
        for (int j = 0; j < 32; j++)
            ip_bin[j] = (ip_vec[i] >> j) & 1;

        node_t *cur = root;
        port_vec[i] = -1;
        int idx = 31;
        while (idx >= 0) {
            if (cur->type == M_NODE)
                port_vec[i] = cur->port;

            if (ip_bin[idx] == LEFT) {
                if (cur->lchild == NULL)
                    break;
                cur = cur->lchild;
            }
            else if (ip_bin[idx] == RIGHT) {
                if (cur->rchild == NULL)
                    break;
                cur = cur->rchild;
            }
            else {
                fprintf(stderr, "Error: invalid ip\n");
                exit(1);
            }
        }
    }
}
```

```
        }
        idx--;
    }
}

return port_vec;
}
```

10.2.2 优化前缀树查找

这里我采取的思路是,先根据 IP 地址的前 16 位进行分类,如果直接映射到对应的前 16 位,则继续以 2 位为索引查找后 16 位,否则直接返回。这样的操作可以减少查找的次数,提高查找的效率。但是,前缀长度可能不是 2 的倍数,所以我多增加了一个域表示匹配节点的前缀是奇数还是偶数。节点的数据结构如下:

```
typedef struct node_advance{
    bool type; //I_NODE or M_NODE
    uint32_t port;
    int prefix_diff; // 0 for even, 1 for odd
    struct node_advance* child[4];
}node_advance_t;
```

构建前缀树时需要考虑的内容比较多。如果前缀长度小于 16,则会映射到多个表项中,需要一一遍历这些表项根节点进行更新,更新条件是根节点为中间节点或者前缀长度不大于新节点的前缀长度。前缀长度不小于 16 时,就需要对相应根节点的子节点进行插入操作,根据 IP 的每两位进行索引,如果子节点为空则申请空间构建子节点,继续向下插入。此时若遇到前缀长度为奇数则将前缀差设置为 1,然后更新节点类型为匹配节点,端口号存储在该节点中。代码如下:

```
// Constructing an advanced trie-tree to lookup according to `forward_file`
void create_tree_advance(const char* forward_file)
{
    // fprintf(stderr,"TODO:%s",__func__);
    notmatch = (node_advance_t *)malloc(sizeof(node_advance_t));
    notmatch->port = -1;
    for (int i = 0; i < MAP_NUM; i++){
        triemap[i] = (node_advance_t *)malloc(sizeof(node_advance_t));
        triemap[i]->port = 0;
        triemap[i]->prefix_diff = 0;
        triemap[i]->type = I_NODE;
        for (int j = 0; j < 4; j++)
            triemap[i]->child[j] = NULL;
        size_advance += sizeof(node_advance_t);
    }
}
```

```
FILE *fp = fopen(forward_file, "r");
if (fp == NULL) {
    fprintf(stderr, "Error: cannot open file %s\n", forward_file);
    exit(1);
}

char line[100];
while (fgets(line, 100, fp) != NULL) {
    char ip_str[15];
    uint32_t port, prefix;
    sscanf(line, "%s %u %u", ip_str, &prefix, &port);
    int ip_dec[4];
    sscanf(ip_str, "%d.%d.%d.%d", &ip_dec[0], &ip_dec[1], &ip_dec[2],
           &ip_dec[3]);

    uint32_t ip_bin = 0;
    for (int i = 0; i < 4; i++)
        ip_bin |= (ip_dec[i] << (3 - i) * 8);

    if (prefix >= MAP_SHIFT){
        node_advance_t *root = triemap[(ip_bin & 0xffff0000) >> MAP_SHIFT];
        node_advance_t *cur = root, *next = NULL;
        int offset, cur_bit, cur_prefix;

        for (cur_prefix = 32-MAP_SHIFT; cur_prefix < prefix-1; cur_prefix+=2){
            offset = 30 - cur_prefix;
            cur_bit = (ip_bin >> offset) & 0x3;
            next = cur->child[cur_bit];
            if (next == NULL){
                next = (node_advance_t *)malloc(sizeof(node_advance_t));
                next->port = 0;
                next->prefix_diff = 0;
                next->type = I_NODE;
                for (int j = 0; j < 4; j++)
                    next->child[j] = NULL;
                cur->child[cur_bit] = next;
                size_advance += sizeof(node_advance_t);
            }
            cur = next;
        }

        if (cur_prefix == prefix - 1){
            offset = 30 - cur_prefix;
```

```
    cur_bit = (ip_bin >> offset) & 0x3;
    int start_bit = cur_bit & 0x2;
    for (int i = 0; i < 2; i++){
        next = cur->child[start_bit + i];
        if (next == NULL){
            next = (node_advance_t *)malloc(sizeof(node_advance_t));
            next->port = port;
            next->prefix_diff = 1;
            next->type = M_NODE;
            for (int j = 0; j < 4; j++){
                next->child[j] = NULL;
            }
            cur->child[start_bit + i] = next;
            size_advance += sizeof(node_advance_t);
        }
    }
}
else{
    cur->port = port;
    cur->type = M_NODE;
}
}
else{
    uint32_t mask = 0xffffffff << (32 - prefix);
    uint32_t start = (ip_bin & mask) >> MAP_SHIFT;
    uint32_t end = start + (1 << (MAP_SHIFT - prefix));
    for (int i = start; i < end; i++){
        if (triemap[i]->type == I_NODE || triemap[i]->prefix_diff >= 16 -
            prefix){
            triemap[i]->port = port;
            triemap[i]->type = M_NODE;
            triemap[i]->prefix_diff = 16 - prefix;
        }
    }
}
}
}

fclose(fp);
return;
}
```

查找前缀树的过程比较类似，只是在查找的过程中，我设计一个 match 节点来存储匹配到的节点内容，需要根据 IP 的前 16 位找到对应的根节点，然后根据 IP 的后 16 位进行查找。查找的过程中，如果遇到匹配节点，则更新 match 节点的状态。代码如下：

```
// Look up the ports of ip in file `ip_to_lookup.txt` using the advanced tree
// input is read from `read_test_data` func
uint32_t *lookup_tree_advance(uint32_t* ip_vec)
{
    // fprintf(stderr,"TODO:%s",__func__);
    uint32_t *port_vec = (uint32_t *)malloc(sizeof(uint32_t) * TEST_SIZE);

    for (int i = 0; i < TEST_SIZE; i++)
    {
        uint32_t ip = ip_vec[i];
        node_advance_t *cur = triemap[ip >> MAP_SHIFT];
        node_advance_t *match = notmatch;

        uint32_t cur_bit;
        int cur_prefix = 32 - MAP_SHIFT;

        if (cur->prefix_diff > 0){
            if (cur->type == M_NODE)
                match = cur;
            cur_bit = (ip >> 14) & 0x3;
            cur = cur->child[cur_bit];
            cur_prefix += 2;
        }

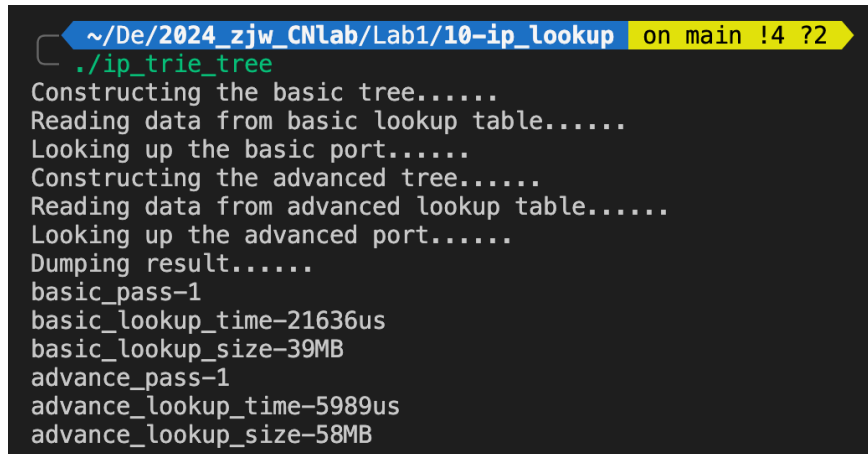
        while (cur != NULL){
            if (cur->type == M_NODE)
                match = cur;
            int offset = 30 - cur_prefix;
            cur_bit = (ip >> offset) & 0x3;
            cur = cur->child[cur_bit];
            cur_prefix += 2;
        }

        port_vec[i] = match->port;
    }

    return port_vec;
}
```

10.3 实验结果

运行程序,得到的结果如下图所示:



```
~/De/2024_zjw_CNlab/Lab1/10-ip_lookup on main !4 ?2
./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-21636us
basic_lookup_size-39MB
advance_pass-1
advance_lookup_time-5989us
advance_lookup_size-58MB
```

图 10.1. 实验结果

可以看出,在查找时间方面,优化前缀树查找的时间大概只有基本前缀树查找的 1/4,这是因为优化前缀树查找的次数更少,查找效率更高。但是在内存方面,优化前缀树查找的内存开销大概是基本前缀树查找的 1.5 倍,这是因为优化前缀树查找的节点数据结构更复杂,需要更多的内存空间,这也是查找效率提高的代价。

10.4 实验总结

本次实验主要是实现了基本前缀树查找和优化前缀树查找,通过对比两者的性能,可以看出优化前缀树查找的效率更高,但是内存开销也更大。这是一个典型的时间和空间的权衡问题,需要根据实际情况选择合适的方案。