高效 IP 路由查找实验报告

李昊宸

2017K8009929044

（一）网络路由机制实现

一、实验内容

1. 实现最基本的前缀树查找
2. 调研并实现某种 IP 前缀查找方案
3. 基于 forwarding-table.txt 数据集(Network, Prefix Length, Port)
   1）只考虑静态数据集，不考虑表的添加或更新
   2）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
   3）对比基本前缀树和所实现 IP 前缀查找的性能

二、实验流程

1. 搭建实验环境

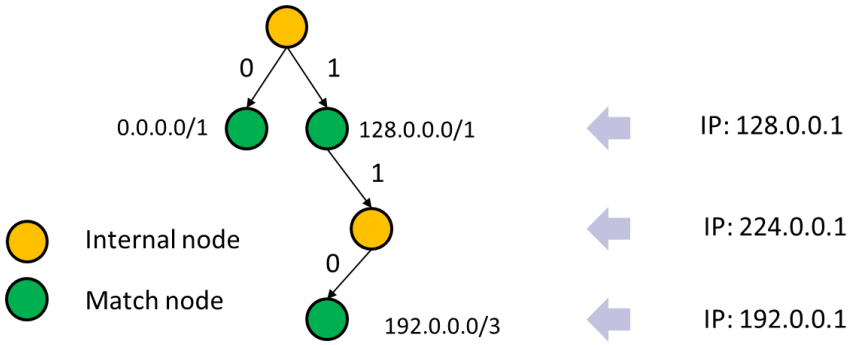
pref-tree.h：前缀树相关头文件
main.c： 前缀树查找的代码实现
forwarding-table.txt：路由表的内容
make_test-table.py：将 forwarding-table 进行压缩，去除具有相同 IP 的表项，只保留掩码最长的一项，保存到 test-table.txt 中

本次实验与上周路由转发实验有一定的联系。上周实验中，路由表的查找方式为线性查找，根本不能满足线速查找转发的要求，所以本次要实现高效的转发算法。

IP 路由查找机制：给定一个 IP，需要将 IP 与前缀长度（掩码）按位与，得到网段的 IP 后在路由表中进行查找。如果没找到，将前缀长度加 1（掩码增长 1 位），继续寻找，最后返回匹配的最长前缀的表项。

前缀树查找： 树的每一个节点表示 IP 地址前缀中的每一位，树的每一层表示第某位前缀的所有节点。查找时，从树的根节点开始遍历，逐位匹配，直到节点没有相应的子节点。

最基本的 1bit 前缀树查找：



图一 1bit 前缀树查找示例

1bit 前缀树的**建立**：

每读取到一个表项，初始化当前节点为根节点，从该表项记录的 IP 的最高位（二进制）开始，如果该位为 1，就访问当前节点节点的右子节点，若不存在就新分配一个节点成为当前节点的右子节点，设置其端口号为当前节点的端口号，然后访问新建的右子节点；如果该位为 0，就访问当前节点节点的左子节点，若不存在就新分配一个节点成为当前节点的左子节点，设置其端口号为当前节点的端口号，然后访问新建的左子节点。然后查看 IP（二进制）的下一位，重复以上过程，直到达到掩码长度为止。访问停止后，设置当前节点的端口为表项中记录的端口。

1bit 前缀树的**查找**：

初始化当前节点为根节点，从 IP 的最高位（二进制）开始，如果该位为 1，就访问当前节点的右子节点，若不存在就返回当前节点；如果该位为 0，就访问当前节点的左子节点，若不存在就返回当前节点。如果没返回，就继续查看 IP（二进制）的下一位，重复以上过程直到返回。

示例：

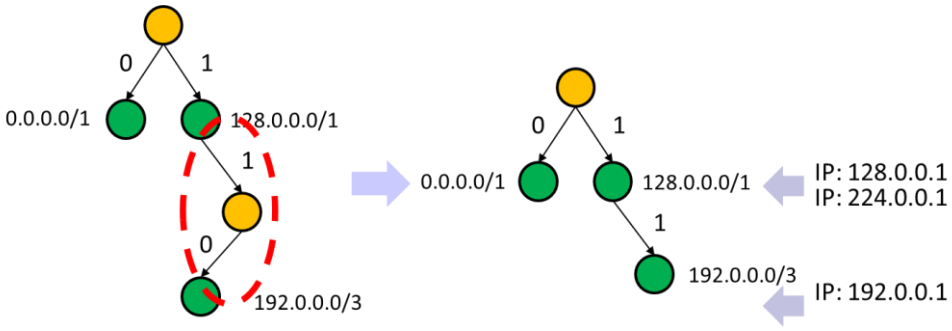
查找 128.0.0.1：第一位为 1，根节点选择右子节点为当前节点；第二位为 0，而当前节点没有左子节点，就返回当前节点，查找结束。

查找 224.0.0.1：第一位为 1，根节点选择右子节点为当前节点；第二位为 1，当前节点选择右子节点为当前节点；第三位为 1，而当前节点没有左子节点，就返回当前节点，查找结束。需要注意的是，虽然当前节点（黄色）是中间生成的节点，在路由表项中没有被记录，但是该节点继承了其父节点的端口信息，所以返回的端口与父节点端

口相同。

查找 192.0.0.1：第一位为 1，根节点选择右子节点为当前节点；第二位为 1，当前节点选择右子节点为当前节点；第三位为 0，当前节点选择左子节点为当前节点；第四位为 0，而当前节点没有左子节点，就返回当前节点，查找结束。

优化一　压缩中间节点的 1bit 前缀树查找：压缩中间节点以减小数据结构，访问数据时 Cache 命中的概率越高



图二 压缩中间节点的 1bit 前缀树查找示例

压缩中间节点的 1bit 前缀树的**建立**：

每读取到一个表项，初始化当前节点为根节点，从该表项记录的 IP 的最高位（二进制）开始，如果该位为 1，就访问当前节点节点的右子节点，若不存在就新分配一个节点成为当前节点的右子节点，设置其端口号为当前节点的端口号，设置其左压缩节点数、右压缩节点数为 0，左压缩比特、右压缩比特为 0，不可压缩标记为 0，然后访问新建的右子节点；如果该位为 0，就访问当前节点节点的左子节点，若不存在就新分配一个节点成为当前节点的左子节点，设置其端口号为当前节点的端口号，设置其左压缩节点数、右压缩节点数为 0，左压缩比特、右压缩比特为 0，不可压缩标记为 0，然后访问新建的左子节点。然后查看 IP（二进制）的下一位，重复以上过程，直到达到掩码长度为止。访问停止后，设置当前节点的端口为表项中记录的端口。

树建立完成后，进行**不可压缩补充标记：**遍历所有的节点，如果节点没有子节点（即该节点为叶子节点）或者节点有两个子节点，就将其不可压缩标记设置为 1。

标记完成后，进行**压缩**（tree_zip）：

tree_zip：

如果当前节点存在左子节点，并且左子节点的不可压缩标记为 0（也即左子节点只有一个子节点），就进行压缩操作：

如果左子节点存在的是左子节点，就设置当前节点的左子节点为左子节点的左子节点，当前节点的左压缩节点数加 1，左压缩比特左移 1 位（后加 0），释放原左子节点，再次调用 tree_zip 重新处理当前节点。

如果左子节点存在的是右子节点，就设置当前节点的左子节点为左子节点的右子节点，当前节点的左压缩节点数加 1，左压缩比特左移 1 位后加 1，释放原左子节点，再次调用 tree_zip 重新处理当前节点。

如果当前节点存在左子节点，并且左子节点的不可压缩标记为 1，就调用 tree_zip 处理左子节点。

如果当前节点存在右子节点，并且右子节点的不可压缩标记为 0（也即右子节点只有一个子节点），就进行压缩操作：

如果右子节点存在的是左子节点，就设置当前节点的右子节点为右子节点的左子节点，当前节点的右压缩节点数加 1，右压缩比特左移 1 位（后加 0），释放原右子节点，再次调用 tree_zip 重新处理当前节点。

如果右子节点存在的是右子节点，就设置当前节点的右子节点为右子节点的右子节点，当前节点的右压缩节点数加 1，右压缩比特左移 1 位后加 1，释放原右子节点，再次调用 tree_zip 重新处理当前节点。

如果当前节点存在右子节点，并且右子节点的不可压缩标记为 1，就调用 tree_zip 处理右子节点。

如果当前节点不存在子节点，就返回。

综上递归调用，压缩树使用的命令是 tree_zip（root_node）。

压缩中间节点的 1bit 前缀树的**查找**：

初始化当前节点为根节点，从 IP 的最高位（二进制）开始：

如果该位为 1，就检查当前节点是否存在右子节点，若不存在就返回当前节点。若存在，就查看当前节点记录的右压缩节点数，如果为 0，就访问右子节点；如果非 0（记为 a），那么比较 IP 接下来的 a 位与右压缩比特是否相同，如果全部相同

就访问右子节点，如果有不同就返回当前节点。

该位为 0，就检查当前节点是否存在左子节点，若不存在就返回当前节点。若存在，就查看当前节点记录的左压缩节点数，如果为 0，就访问左子节点；如果非 0（记为 a），那么比较 IP 接下来的 a 位与左压缩比特是否相同，如果全部相同就访问左子节点，如果有不同就返回当前节点。

如果没有返回，就继续查看 IP 接下来第一个没有比较过的位，重复以上过程。

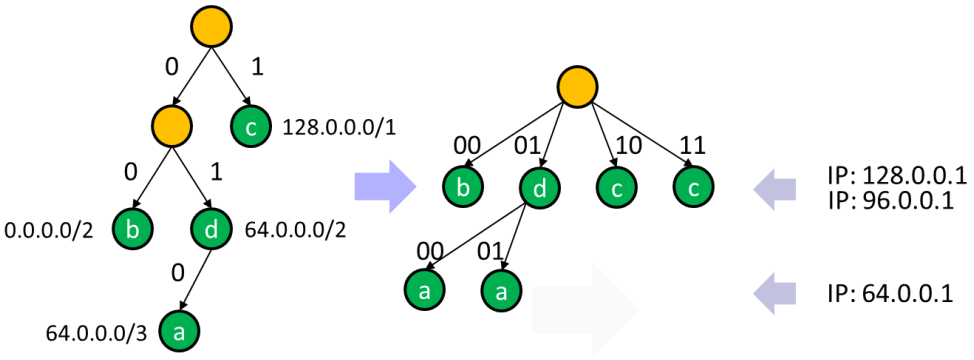
示例：

查找 128.0.0.1：第一位为 1，根节点右压缩节点数为 0，选择右子节点为当前节点；第二位为 0，而当前节点没有左子节点，就返回当前节点，查找结束。

查找 224.0.0.1：第一位为 1，根节点右压缩节点数为 0，选择右子节点为当前节点；第二位为 1，当前节点右压缩节点数为 1，右压缩比特为 0，查看 IP 下一位，为 0，与右压缩比特不等，就返回当前节点，查找结束。

查找 192.0.0.1：第一位为 1，根节点右压缩节点数为 0，选择右子节点为当前节点；第二位为 1，当前节点右压缩节点数为 1，右压缩比特为 0，查看 IP 下一位，为 0，当前节点选择右子节点为当前节点；第四位为 0，而当前节点没有左子节点，就返回当前节点，查找结束。

优化二　2bits 前缀树查找：前缀树中每次不只匹配 1 bit，而是多 bit 一起匹配，可以减少内存访问足迹



图三 2bits 前缀树查找示例

2bits 前缀树的**建立：**

每读取到一个表项，初始化当前节点为根节点，从该表项记录的 IP 的最高位（二进制）开始，如果接下来两位为 11，就访问当前节点节点的 11 子节点，若不存在就新分配一个节点成为当前节点的 11 子节点，设置其端口号为当前节点的端口号，然后访

问新建的 11 子节点；如果接下来两位为 10，就访问当前节点节点的 10 子节点，若不存在就新分配一个节点成为当前节点的 10 子节点，设置其端口号为当前节点的端口号，然后访问新建的 10 子节点；如果接下来两位为 01，就访问当前节点节点的 01 子节点，若不存在就新分配一个节点成为当前节点的 01 子节点，设置其端口号为当前节点的端口号，然后访问新建的 01 子节点；如果接下来两位为 00，就访问当前节点节点的 00 子节点，若不存在就新分配一个节点成为当前节点的 00 子节点，设置其端口号为当前节点的端口号，然后访问新建的 00 子节点。然后查看 IP（二进制）的下两位，重复以上过程，直到达到或超过掩码长度为止。

访问停止后，如果掩码为偶数位，就设置当前节点的端口为表项中记录的端口。如果掩码为奇数位，说明还要再补充子节点。如果 IP 接下来的一位为 1，就查看当前节点的 11 和 10 子节点：如果 11 子节点不存在，就新分配一个节点成为当前节点的 11 子节点，设置端口号为表项中记录的端口；如果 10 子节点不存在，就新分配一个节点成为当前节点的 10 子节点，设置端口号为表项中记录的端口。如果 IP 接下来的一位为 0，就查看当前节点的 01 和 00 子节点：如果 01 子节点不存在，就新分配一个节点成为当前节点的 01 子节点，设置端口号为表项中记录的端口；如果 00 子节点不存在，就新分配一个节点成为当前节点的 00 子节点，设置端口号为表项中记录的端口。

2bits 前缀树的**查找**：

初始化当前节点为根节点，从 IP 的最高位（二进制）开始，如果接下来两位为 11，就访问当前节点的 11 子节点，若不存在就返回当前节点；如果接下来两位为 10，就访问当前节点的 10 子节点，若不存在就返回当前节点；如果接下来两位为 01，就访问当前节点的 01 子节点，若不存在就返回当前节点；如果接下来两位为 00，就访问当前节点的 00 子节点，若不存在就返回当前节点。如果没返回，就继续查看 IP（二进制）的下两位，重复以上过程直到返回。

示例：

查找 128.0.0.1：第一二位为 10，根节点选择 10 子节点为当前节点；第三四位为 00，而当前节点没有 00 子节点，就返回当前节点，查找结束。

查找 96.0.0.1：第一二位为 01，根节点选择 01 子节点为当前节点；第三四位为 10， 而当前节点没有 10 子节点，就返回当前节点，查找结束。

查找 64.0.0.1：第一二位为 01，根节点选择 01 子节点为当前节点；第三四位为 00，当前节点选择 00 子节点为当前节点；第五六位为 00，而当前节点没有 10 子节点，就返回当前节点，查找结束。

2.启动脚本

```
make all
sudo python make_test-table.py
./pref-tree
```

三、实验结果及分析

1. 实验结果

```
moto@CN-VirtualBox:/mnt/shared/code$ ./pref-tree
For all IP in forwarding-table.txt:
  1_bit_prefix_tree:
    1646585 nodes
    39518040 B.
1_bit_prefix_zip_tree:
    1306854 nodes
    31364496 B.
2_bits_prefix_tree:
    1202748 nodes
    48109920 B.
Search all IP in test-table.txt:
  1_bit_prefix_tree_search totally takes  0.858134 seconds.
  1_bits_prefix_zip_tree_search totally takes  0.902650 seconds.
  2_bits_prefix_tree_search totally takes  0.831286 seconds.
```

图四 三种前缀树的准确性、内存和查找时间结果

2. 实验分析

对于最基本的 1bit 前缀树，生成了最多数量的节点，平均访问节点次数最多，耗时较多。

对于压缩中间节点的 1bit 前缀树，生成了中等数量的节点，内存开销最少，平均访问节点次数较少，理论上耗时应该比 1bit 前缀树更少，但是一方面，在算法的实现上，是先生成了全部的节点，然后再进行删除操作，实际上并没能明显的提高 Cache 的命中概率；另一方面，查找算法实现的较为耗时，可以寻找更好的查找算法。两方面共同导致了查找时间的增加。

对于 2bits 前缀树，生成了最少数量的节点，但是拥有最大的内存开销。平均访问节点次数最少，耗时也最少，是因为减小了内存访问足迹，减少了 CPU 指令周期数。

三次运行检查为，将输入的 IP 进行查询，将返回的端口与标准测试端口进行比对，如果出错就打印。运行结束后，并没有打印结果，说明查找比对没有错误。标准测试端口是在原转发表的基础上，对于相同的 IP，选取掩码最长的表项中记录的端口作为标准测试端口进行比对，所以满足 CIDR 机制。

（二）实验代码详解

本次实验较为直观，每一个算法说明的都较为详细，此处仅仅直接贴上代码。

（1）void build_prefix_tree(Tree prefix_tree)：建立最基本的1bit 前缀树

```
void build_prefix_tree(Tree prefix_tree)
{
    prefix_tree->port = 255;
    FILE *fp = fopen(source, "r");
    uint32_t ip_pointer = 1 << 31;
    TNode *node = NULL;
    IP_entry *ip_entry = (IP_entry*)malloc(sizeof(IP_entry));
    char s[25];

    while (1)
     {
        fgets(s, 30, fp);
        if(feof(fp))
        {
            break;
        }
        memset(ip_entry, 0, sizeof(IP_entry));
        uint32_t IP[4] = {0};
        int i = 0;
        sscanf(s, "%u.%u.%u.%u %u %u", &IP[0], &IP[1], &IP[2], &I
P[3], &ip_entry->mask, &ip_entry->port);
        ip_entry->ip = IP[0];
        ip_entry->ip = (ip_entry->ip << 8) + IP[1];
        ip_entry->ip = (ip_entry->ip << 8) + IP[2];
        ip_entry->ip = (ip_entry->ip << 8) + IP[3];
        node = prefix_tree;

        int IP_bit[32] = {0};
        for(i = 0; i < 32; i++)
        {
            IP_bit[i] = ip_entry->ip & (ip_pointer >> i);
            if(i >= ip_entry->mask)
                IP_bit[i] = -1;
        }
        for (int j = 0; j < ip_entry->mask; j++)
        {
            if (IP_bit[j]) {
```

```
                    if (!node->RChild)
                    {
                        TNode *tmp_node = (TNode*)malloc(sizeof(TNode
));
                        tmp_node->LChild = tmp_node->RChild = NULL;
                        tmp_node->port = node->port;
                        node->RChild = tmp_node;
                    }
                    node = node->RChild;
                }
                else
                {
                    if (!node->LChild)
                    {
                        TNode *tmp_node = (TNode*)malloc(sizeof(TNode
));
                        tmp_node->LChild = tmp_node->RChild = NULL;
                        tmp_node->port = node->port;
                        node->LChild = tmp_node;
                    }
                    node = node->LChild;
                }
            }
            node->port = ip_entry->port;
        }
        fclose(fp);
    }
```

（2）uint32_t lookup_prefix_tree(Tree prefix_tree, uint32_t IP)：最基本 1bit 前缀树的查找

```
    uint32_t lookup_prefix_tree(Tree prefix_tree, uint32_t IP)
    {
        uint32_t ip_pointer = 1 << 31;
        TNode *node = prefix_tree;
        TNode *parent_node = NULL;
        int IP_bit[32] = {0};
        for(int i = 0; i < 32; i++)
            {
                IP_bit[i] = IP & (ip_pointer >> i);
            }
        for (int i = 0; node; i++) {
            parent_node = node;
```

```
            node = IP_bit[i] ? node->RChild : node->LChild;
        }
        return parent_node->port;
    }
```

（3）void sign_zipable(Tree_zip prefix_tree)：压缩中间节点的1bit 前缀树的不可压缩标记

```
    void sign_zipable(Tree_zip prefix_tree)
    {
        int i = 0;
        if(prefix_tree->LChild)
        {
            i++;
            sign_zipable(prefix_tree->LChild);
        }
        if(prefix_tree->RChild)
        {
            i++;
            sign_zipable(prefix_tree->RChild);
        }
        if(i == 2 || i == 0) prefix_tree->unzipable = 1;
        count_zipable = (prefix_tree->unzipable)? count_zipable:count
    _zipable+1;
    }
```

（4）void tree_zip(Tree_zip prefix_tree)：压缩中间节点的 1bit 前缀树的压缩

```
    void tree_zip(Tree_zip prefix_tree)
    {
        if(prefix_tree->LChild)
        {
            if(!prefix_tree->LChild->unzipable)  //left child node ca
    n be zipped
            {
                if(prefix_tree->LChild->LChild)
                {
                    if(!prefix_tree->LChild->RChild)
                    {
                        TNode_zip *tmp_node = prefix_tree->LChild;
                        prefix_tree->LChild = tmp_node->LChild;
```

```
                    prefix_tree->Lnum = prefix_tree->Lnum + 1;
                    prefix_tree->L_mask = (prefix_tree->L_mask <<
1) + 0;
                    free(tmp_node);
                    tree_zip(prefix_tree);
                }
                else
                {
                    prefix_tree->LChild->unzipable = 1;
                    tree_zip(prefix_tree->LChild);
                }
            }
            else
            {
                if(!prefix_tree->LChild->RChild)
                {
                    return;
                }
                else
                {
                    TNode_zip *tmp_node = prefix_tree->LChild;
                    prefix_tree->LChild = tmp_node->RChild;
                    prefix_tree->Lnum = prefix_tree->Lnum + 1;
                    prefix_tree->L_mask = (prefix_tree->L_mask <<
1) + 1;
                    free(tmp_node);
                    tree_zip(prefix_tree);
                }
            }

        }
        else
        {
            tree_zip(prefix_tree->LChild);
        }
    }
    if(prefix_tree->RChild)
    {
        if(!prefix_tree->RChild->unzipable)  //right child node c
an be zipped
        {
            if(prefix_tree->RChild->LChild)
            {
                if(!prefix_tree->RChild->RChild)
```

```
                {
                    TNode_zip *tmp_node = prefix_tree->RChild;
                    prefix_tree->RChild = tmp_node->LChild;
                    prefix_tree->Rnum = prefix_tree->Rnum + 1;
                    prefix_tree->R_mask = (prefix_tree->R_mask <<
1) + 0;
                    free(tmp_node);
                    tree_zip(prefix_tree);
                }
                else
                {
                    prefix_tree->RChild->unzipable = 1;
                    tree_zip(prefix_tree->RChild);
                }
            }
            else
            {
                if(!prefix_tree->RChild->RChild)
                {
                    return;
                }
                else
                {
                    TNode_zip *tmp_node = prefix_tree->RChild;
                    prefix_tree->RChild = tmp_node->RChild;
                    prefix_tree->Rnum++;
                    prefix_tree->R_mask = (prefix_tree->R_mask <<
1) + 1;
                    free(tmp_node);
                    tree_zip(prefix_tree);
                }

            }

        }
        else
        {
            tree_zip(prefix_tree->RChild);
        }
    }
}
```

。

（5）void build_prefix_zip_tree(Tree_zip prefix_tree)：压缩中间节点的 1bit 前缀树的建立

```
void build_prefix_zip_tree(Tree_zip prefix_tree)
{
    prefix_tree->port = 255;
    prefix_tree->unzipable = 1;
    FILE *fp = fopen(source, "r");
    uint32_t ip_pointer = 1 << 31;
    TNode_zip *node = NULL;
    IP_entry *ip_entry = (IP_entry*)malloc(sizeof(IP_entry));
    char s[25];

    while (1)
     {
        fgets(s, 30, fp);
        if(feof(fp))
        {
            break;
        }
        memset(ip_entry, 0, sizeof(IP_entry));
        uint32_t IP[4] = {0};
        int i = 0;
        sscanf(s, "%u.%u.%u.%u %u %u", &IP[0], &IP[1], &IP[2], &IP[3], &ip_entry->mask, &ip_entry->port);
        ip_entry->ip = IP[0];
        ip_entry->ip = (ip_entry->ip << 8) + IP[1];
        ip_entry->ip = (ip_entry->ip << 8) + IP[2];
        ip_entry->ip = (ip_entry->ip << 8) + IP[3];
        node = prefix_tree;

        int IP_bit[32] = {0};
        for(i = 0; i < 32; i++)
        {
            IP_bit[i] = ip_entry->ip & (ip_pointer >> i);
            if(i >= ip_entry->mask)
                IP_bit[i] = -1;
        }
        for (int j = 0; j < ip_entry->mask; j++)
        {
            if (IP_bit[j]) {
                if (!node->RChild)
                {
```

```
                    TNode_zip *tmp_node = (TNode_zip*)malloc(size
of(TNode_zip));
                    tmp_node->LChild = tmp_node->RChild = NULL;
                    tmp_node->Lnum = 0;
                    tmp_node->L_mask = 0;
                    tmp_node->Rnum = 0;
                    tmp_node->R_mask = 0;
                    tmp_node->unzipable = 0;
                    tmp_node->port = node->port;
                    node->RChild = tmp_node;
                }
                node = node->RChild;
            }
            else
            {
                if (!node->LChild)
                {
                    TNode_zip *tmp_node = (TNode_zip*)malloc(size
of(TNode_zip));
                    tmp_node->LChild = tmp_node->RChild = NULL;
                    tmp_node->Lnum = 0;
                    tmp_node->L_mask = 0;
                    tmp_node->Rnum = 0;
                    tmp_node->R_mask = 0;
                    tmp_node->unzipable = 0;
                    tmp_node->port = node->port;
                    node->LChild = tmp_node;
                }
                node = node->LChild;
            }
        }
        node->port = ip_entry->port;
        node->unzipable = 1;
    }
    sign_zipable(prefix_tree);
    tree_zip(prefix_tree);
    fclose(fp);
}
```

（6）uint32_t lookup_prefix_zip_tree(Tree_zip prefix_tree, uint32_t IP)：压缩中间节点的 1bit 前缀树的查找

```
uint32_t lookup_prefix_zip_tree(Tree_zip prefix_tree, uint32_t IP
```

```
)
{
    uint32_t ip_pointer = 1 << 31;
    TNode_zip *node = prefix_tree;
    TNode_zip *parent_node = NULL;
    int IP_bit[32] = {0};
    for(int i = 0; i < 32; i++)
        {
            IP_bit[i] = IP & (ip_pointer >> i);
        }
    for (int i = 0; node; i++) {
        parent_node = node;
        int flag = IP_bit[i] ? 1 : 0;
        node = IP_bit[i] ? node->RChild : node->LChild;
        int j = IP_bit[i] ? parent_node->Rnum : parent_node->Lnum;
        while(j > 0)
        {
            i++;
            int ip_i = IP_bit[i] ? 1 : 0 ;
            if( flag &(ip_i != ((parent_node->R_mask >> (j-
1)) & 1) ))
            {
                return parent_node->port;
            }
            if(!flag &(ip_i != ((parent_node->L_mask >> (j-
1)) & 1) ))
            {
                return parent_node->port;
            }
            j--;

        }
    }
    return parent_node->port;
}
```

（7）void *checking_db_thread(void *param)：检查数据库节点失效进程

。

（8）void build_2bits_prefix_tree(Tree_2bits prefix_tree)：2bits 前缀树的建立

```
void build_2bits_prefix_tree(Tree_2bits prefix_tree)
{
    prefix_tree->port = 255;
    FILE *fp = fopen(source, "r");
    uint32_t ip_pointer = 1 << 31;
    Tnode_2bit *node = NULL;
    IP_entry *ip_entry = (IP_entry*)malloc(sizeof(IP_entry));
    char s[25];

    while (1)
    {
        fgets(s, 30, fp);
        if(feof(fp))
        {
            break;
        }
        memset(ip_entry, 0, sizeof(IP_entry));
        uint32_t IP[4] = {0};
        int i = 0;
        sscanf(s, "%u.%u.%u.%u %u %u", &IP[0], &IP[1], &IP[2], &I
P[3], &ip_entry->mask, &ip_entry->port);
        ip_entry->ip = IP[0];
        ip_entry->ip = (ip_entry->ip << 8) + IP[1];
        ip_entry->ip = (ip_entry->ip << 8) + IP[2];
        ip_entry->ip = (ip_entry->ip << 8) + IP[3];
        node = prefix_tree;

        int IP_bit[32] = {0};
        for(i = 0; i < 32; i++)
        {
            IP_bit[i] = ip_entry->ip & (ip_pointer >> i);
            if(i >= ip_entry->mask)
                IP_bit[i] = -1;
        }

        for (int j = 0; j < ip_entry->mask-1; j += 2)
        {
            if (IP_bit[j] )
            {
                if (IP_bit[j + 1])
                {
                    if (!node->Child11)
                    {
                        Tnode_2bit *tmp_node = (Tnode_2bit*)mallo
```

```
c(sizeof(Tnode_2bit));
                        tmp_node->Child00 = tmp_node->Child01 = N
ULL;
                        tmp_node->Child10 = tmp_node->Child11 = N
ULL;
                        tmp_node->port = node->port;
                        node->Child11 = tmp_node;
                    }
                    node = node->Child11;
                }
                else
                {
                    if (!node->Child10)
                    {
                        Tnode_2bit *tmp_node = (Tnode_2bit*)mallo
c(sizeof(Tnode_2bit));
                        tmp_node->Child00 = tmp_node->Child01 = N
ULL;
                        tmp_node->Child10 = tmp_node->Child11 = N
ULL;
                        tmp_node->port = node->port;
                        node->Child10 = tmp_node;
                    }
                    node = node->Child10;
                }
            }
            else
            {
                if (IP_bit[j + 1])
                {
                    if (!node->Child01)
                    {
                        Tnode_2bit *tmp_node = (Tnode_2bit*)mallo
c(sizeof(Tnode_2bit));
                        tmp_node->Child00 = tmp_node->Child01 = N
ULL;
                        tmp_node->Child10 = tmp_node->Child11 = N
ULL;
                        tmp_node->port = node->port;
                        node->Child01 = tmp_node;
                    }
                    node = node->Child01;
                }
                else
```

```
                {
                    if (!node->Child00)
                    {
                        Tnode_2bit *tmp_node = (Tnode_2bit*)mallo
c(sizeof(Tnode_2bit));
                        tmp_node->Child00 = tmp_node->Child01 = N
ULL;
                        tmp_node->Child10 = tmp_node->Child11 = N
ULL;
                        tmp_node->port = node->port;
                        node->Child00 = tmp_node;
                    }
                    node = node->Child00;
                }
            }
        }
        if (ip_entry->mask % 2)
        {
            if (IP_bit[ip_entry->mask-1])
            {
                if (!node->Child11)
                {
                    Tnode_2bit *tmp_node = (Tnode_2bit*)malloc(si
zeof(Tnode_2bit));
                    tmp_node->Child00 = tmp_node->Child01 = NULL;
                    tmp_node->Child10 = tmp_node->Child11 = NULL;
                    tmp_node->port = ip_entry->port;
                    node->Child11 = tmp_node;
                }
                if (!node->Child10)
                {
                    Tnode_2bit *tmp_node = (Tnode_2bit*)malloc(si
zeof(Tnode_2bit));
                    tmp_node->Child00 = tmp_node->Child01 = NULL;
                    tmp_node->Child10 = tmp_node->Child11 = NULL;
                    tmp_node->port = ip_entry->port;
                    node->Child10 = tmp_node;
                }
            }
            else
            {
                if (!node->Child01)
                {
                    Tnode_2bit *tmp_node = (Tnode_2bit*)malloc(si
```

```
zeof(Tnode_2bit));
                    tmp_node->Child00 = tmp_node->Child01 = NULL;
                    tmp_node->Child10 = tmp_node->Child11 = NULL;
                    tmp_node->port = ip_entry->port;
                    node->Child01 = tmp_node;
                }
                if (!node->Child00)
                {
                    Tnode_2bit *tmp_node = (Tnode_2bit*)malloc(si
zeof(Tnode_2bit));
                    tmp_node->Child00 = tmp_node->Child01 = NULL;
                    tmp_node->Child10 = tmp_node->Child11 = NULL;
                    tmp_node->port = ip_entry->port;
                    node->Child00 = tmp_node;
                }
            }
        } else {
            node->port = ip_entry->port;
        }
    }
    fclose(fp);
}
```

(9) uint32_t lookup_2bits_prefix_tree(Tree_2bits prefix_tree, uint32_t IP)：2bits 前缀树的查找

```
uint32_t lookup_2bits_prefix_tree(Tree_2bits prefix_tree, uint32_
t IP)
{
    uint32_t ip_pointer = 1 << 31;
    Tnode_2bit *node = prefix_tree;
    Tnode_2bit *parent_node = NULL;

    int IP_bit[32] = {0};
        for(int i = 0; i < 32; i++)
        {
            IP_bit[i] = IP & (ip_pointer >> i);

        }

    for (int i = 0; node; i += 2)
    {
        parent_node = node;
```

```
            if (IP_bit[i])
            {
                if(IP_bit[i + 1])
                    node = node->Child11;
                else node = node->Child10;
            }
            else
            {
                if(IP_bit[i + 1])
                    node = node->Child01;
                else node = node->Child00;
            }
        }
        return parent_node->port;
    }
```