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

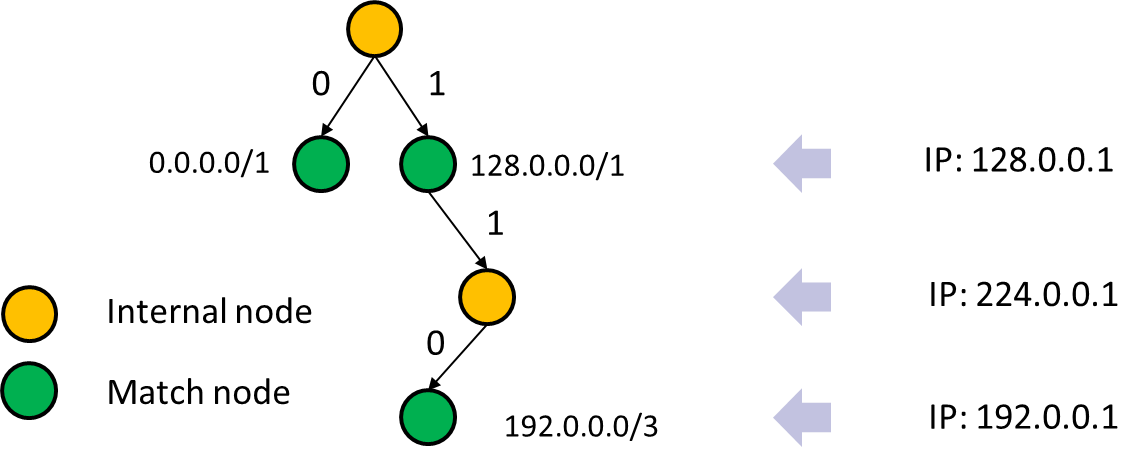
make\_test-table.py：将forwarding-table进行压缩，去除具有相同IP的表项，只保留掩码最长的一项，保存到test-table.txt中

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。

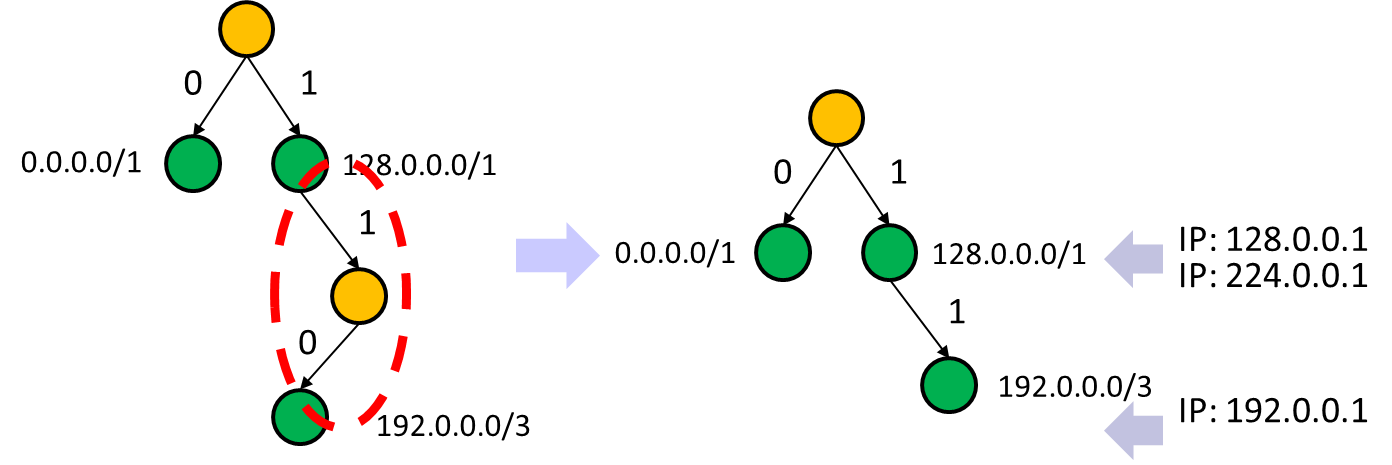
查找192.0.0.1：第一位为1，根节点选择右子节点为当前节点；第二位为1，当前节点选择右子节点为当前节点；第三位为0，当前节点选择左子节点为当前节点；第四位为0，而当前节点没有左子节点，就返回当前节点，查找结束。

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

IP: 128.0.0.1

IP: 224.0.0.1

IP: 192.0.0.1



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。

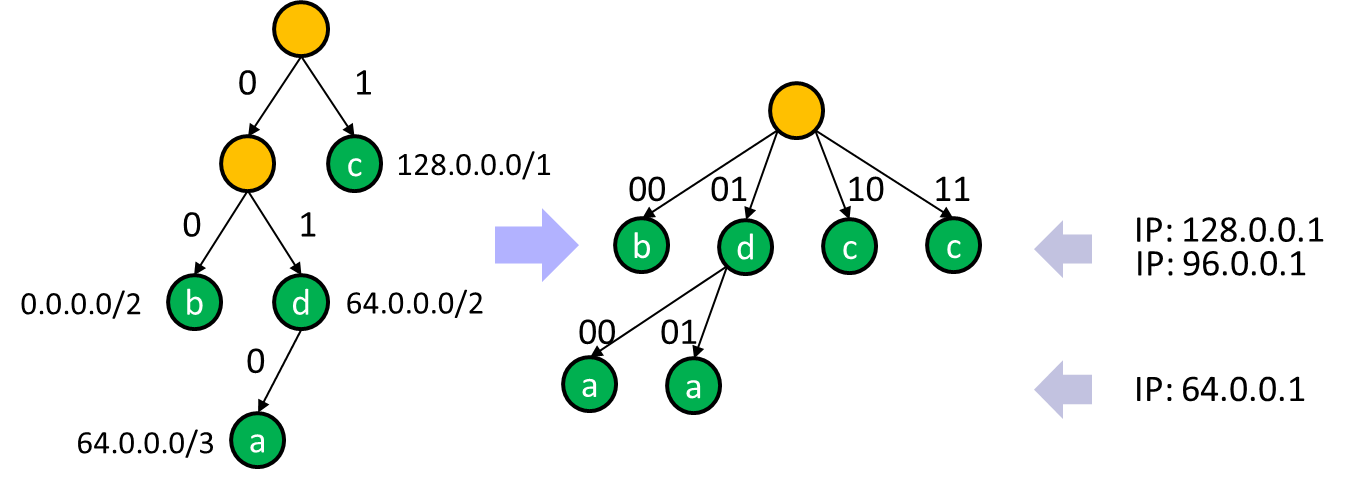
示例：

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



IP: 128.0.0.1

IP: 96.0.0.1

IP: 64.0.0.1

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

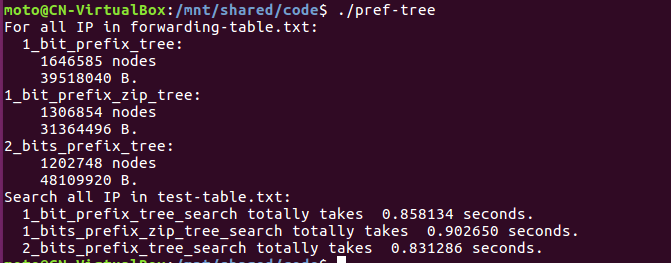
make all

sudo python make\_test-table.py

./pref-tree

三、实验结果及分析

1. 实验结果



图四 三种前缀树的准确性、内存和查找时间结果

1. 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], &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 \*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 can 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 can 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**(sizeof(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**(sizeof(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], &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-1; j += 2)

        {

            if (IP\_bit[j] )

            {

                if (IP\_bit[j + 1])

                {

                    if (!node->Child11)

                    {

                        Tnode\_2bit \*tmp\_node = (Tnode\_2bit\*)**malloc**(sizeof(Tnode\_2bit));

                        tmp\_node->Child00 = tmp\_node->Child01 = NULL;

                        tmp\_node->Child10 = tmp\_node->Child11 = NULL;

                        tmp\_node->port = node->port;

                        node->Child11 = tmp\_node;

                    }

                    node = node->Child11;

                }

                else

                {

                    if (!node->Child10)

                    {

                        Tnode\_2bit \*tmp\_node = (Tnode\_2bit\*)**malloc**(sizeof(Tnode\_2bit));

                        tmp\_node->Child00 = tmp\_node->Child01 = NULL;

                        tmp\_node->Child10 = tmp\_node->Child11 = NULL;

                        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\*)**malloc**(sizeof(Tnode\_2bit));

                        tmp\_node->Child00 = tmp\_node->Child01 = NULL;

                        tmp\_node->Child10 = tmp\_node->Child11 = NULL;

                        tmp\_node->port = node->port;

                        node->Child01 = tmp\_node;

                    }

                    node = node->Child01;

                }

                else

                {

                    if (!node->Child00)

                    {

                        Tnode\_2bit \*tmp\_node = (Tnode\_2bit\*)**malloc**(sizeof(Tnode\_2bit));

                        tmp\_node->Child00 = tmp\_node->Child01 = NULL;

                        tmp\_node->Child10 = tmp\_node->Child11 = NULL;

                        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**(sizeof(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**(sizeof(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**(sizeof(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**(sizeof(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;

}