10-高效 IP 路由查找实验报告

陈彦帆 2018K8009918002

1、实验内容

（1）实现最基本的前缀树插入和查找。

（2）调研并实现某种 IP 前缀插入和查找方案。

（3）基于 forwarding-table.txt 数据集，检查所实现的 IP 前缀查找是否正确，对比基本前缀树和所实现 IP 前缀查找的性能。

2、实验流程

与实验内容相同。

3、分析与实现

（1）基本前缀树

原理：路由查找需要寻找与给定 IP 匹配的前缀最长的项。前缀树的每一个结点表示当前匹配了 IP 从最高位到某一位的一个前缀。通过 1bit 基本前缀树查找时，从 IP 地址的最高位开始，逐位匹配并进入对应的子树，直到结点无相应子结点。由于 CIDR 机制，一 IP 前缀可能包含另一 IP 前缀，故需要查找至匹配失败，以保证最长前缀匹配。

插入时，先对给定前缀进行查找，当对应位无法匹配时，创建一个新结点作为当前结点对应位的子树，直到完成查找，将对应的路由表项插入到最后匹配的结点中。

实现细节：

① 采用静态链表，可用一个 4 字节的 int 类型表示子树的地址，而非 8 字节的指针，压缩了结点占用空间。

② 将子树指针和结点的路由信息分开存储，压缩了前缀树空间。虽然每访问一个结点需要两次访存，但这两次访存请求可以并行发出，经过实际测试，性能略优于将子树指针和路由信息存在一起的方案。

③ 对于数据集中存在的重叠冲突的情况，由于每次查找都需要查找至匹配失败，可以保证匹配的是最长前缀，前缀较短的冲突项不会对查找造成影响。

具体实现：

① 数据结构

```
#define MAXN 1650000
#define MAXIP 700000
int trie[MAXN][2]; //两个子树指针
char port[MAXN];   //存储端口信息，为 0 表示该结点无匹配项。因此存储的端口号需要加一。
int cnt, cntip;    //全局总结点数和总 ip 数
```

② 插入

对给定前缀进行查找，当对应位无法匹配时，分配静态链表中的一项，作为当前结点对应位的子树。最后将端口信息存进匹配失败结点对应的 port 表项中。

```
int insert(unsigned ip, int mask, char p)
{
    int cur = 0, bit;
    if(MAXN<cnt+40) return 0;
    for(int i=0;i<mask;i++){
        bit = ip>>(31-i) & 1;
        if(trie[cur][bit]==0)
            trie[cur][bit] = ++cnt;
        cur = trie[cur][bit];
    }
    if(port[cur]) return 0; //存在冲突项
    else port[cur] = p+1;   //为 0 表示该结点无匹配项。因此存储的端口号需要加一。
    return 1;
}
```

③ 查找

每进入一个结点，需要查找 port 表查看当前是否有匹配的端口，若有，则更新找到的端口，直至匹配失败，以保证最长前缀匹配。由于插入时端口号加一，故查找时需要减一。

```
char lookup(unsigned ip)
{
    int cur = 0;
    char p = 0;
    for(int i=0;i<32;i++){
        int bit = ip>>(31-i) & 1;
        cur = trie[cur][bit];
        if(cur==0)
            break;
        if(port[cur])
            p = port[cur];
    }
    return p-1;
}
```

④ 验证

全部插入完成后，对所有插入项进行逐一验证以确保功能正确。

```
int verify(unsigned ip, int mask, char ps)
{
    int cur = 0;
    char p = 0;
    for(int i=0;i<mask;i++){
        int bit = ip>>(31-i) & 1;
        if((cur = trie[cur][bit])>0) {}
        else break;
    }
    p = port[cur];
    return p==ps+1;
}
```

⑤ 测试与计时

对所有插入项进行查找（前缀后的主机号均为 0），返回所有端口之和作为验证，计算总查找时间，将总时间除以插入项总数得到平均查找时间。计算总时间时，采用 gettimeofday 系统调用，其精度为 us 级，而查找的总时间约为几十 ms，精度符合要求。

（2）改进的 IP 前缀查找方案：多 bit 前缀树

原理：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。对于 n bit 前缀树，每个结点有 2^n 个子树。

本次实验实现了 2 bit 前缀树和 4 bit 前缀树。

实现细节：对于 n bit 前缀树，每个匹配结点除了保存当前匹配的端口号，还保存从当前结点开始，多匹配 k bit（0<k<n）后对应的所有端口号，用于保存前缀长度不是 n 的倍数的项的端口信息。

与基本前缀树一样，仍采用静态链表。

具体实现：以 2 bit 前缀树为例

① 数据结构

```
typedef struct{
    char port;
    char odd[2];
} Node;
int trie[MAXN][4];
Node port[MAXN];
```

```
int cnt, cntip;
```

② 插入

```
int insert(unsigned ip, int mask, char p)
{
    int cur = 0, bit;
    if(MAXN<cnt+40) return 0;
    for(int i=1;i<mask;i+=2){
        bit = ip>>(31-i) & 3;
        if(trie[cur][bit]==0)
            trie[cur][bit] = ++cnt;
        cur = trie[cur][bit];
    }
    if(mask&1){
        bit = ip>>(32-mask)&1;
        if(port[cur].odd[bit]) return 0;
        else port[cur].odd[bit] = p+1;
    }
    else{
        if(port[cur].port) return 0;
        else port[cur].port = p+1;
    }
    return 1;
}
```

③ 查找

需要查找至匹配失败。每进入一个结点，需要查找 port 表查看当前是否有匹配当前前缀的端口，以及多匹配 1 位对应的端口。

```
char lookup(unsigned ip)
{
    int cur = 0;
    char p = 0;
    for(int i=1;i<32;i+=2) {
        int bit = ip>>(31-i) & 3;
        int bit1 = bit>>1;
        if(port[cur].odd[bit1])
            p = port[cur].odd[bit1];
        cur = trie[cur][bit];
        if(cur==0)
            break;
        if(port[cur].port)
            p = port[cur].port;
    }
    return p-1;
}
```

4、结果与讨论

（1）基本前缀树：

测试命令：

```
gcc -O3 trie.c -o trie
./trie
```

测试结果如图 1。

```
alphabet@ubuntu:~/netexp/10/10-lookup$ ./trie
Entries:697882 Nodes: 1646584
Time:31344 us Avg:44 ns
Mem:14MB Sum_val:2452234
alphabet@ubuntu:~/netexp/10/10-lookup$ ./trie
Entries:697882 Nodes: 1646584
Time:31836 us Avg:45 ns
Mem:14MB Sum_val:2452234
alphabet@ubuntu:~/netexp/10/10-lookup$ ./trie
Entries:697882 Nodes: 1646584
Time:32241 us Avg:46 ns
Mem:14MB Sum_val:2452234
```

图 1

测试结果显示，基本前缀树的平均查找时间为 45ns，对于约 70 万条前缀，占用内存空间为 14MB。并且 verify 函数没有报错，说明经验证，查找结果正确。

（2）2 bit 前缀树：

测试命令：

```
gcc -O3 trie_2bit.c -o trie_2bit
./trie_2bit
```

测试结果如图 2。

```
alphabet@ubuntu:~/netexp/10/10-lookup$ ./trie_2bit
Entries:697882 Nodes: 943995
Time:22871 us Avg:32 ns
Mem:17MB Sum_val:2452234
alphabet@ubuntu:~/netexp/10/10-lookup$ ./trie_2bit
Entries:697882 Nodes: 943995
Time:22146 us Avg:31 ns
Mem:17MB Sum_val:2452234
alphabet@ubuntu:~/netexp/10/10-lookup$ ./trie_2bit
Entries:697882 Nodes: 943995
Time:21459 us Avg:30 ns
Mem:17MB Sum_val:2452234
```

图 2

测试结果显示，2 bit 前缀树的平均查找时间为 31ns，对于约 70 万条前缀，占用内存空间为 17MB。返回的查找结果与基本前缀树相同，说明结果正确。相比基本 1 bit 前缀树，结点数更少，占用内存增大约 21.4%，平均查找时间减少约 31.1%。

（3）4 bit 前缀树

测试命令：

```
gcc -O3 trie_4bit.c -o trie_4bit
./trie_4bit
```

测试结果如图 3。

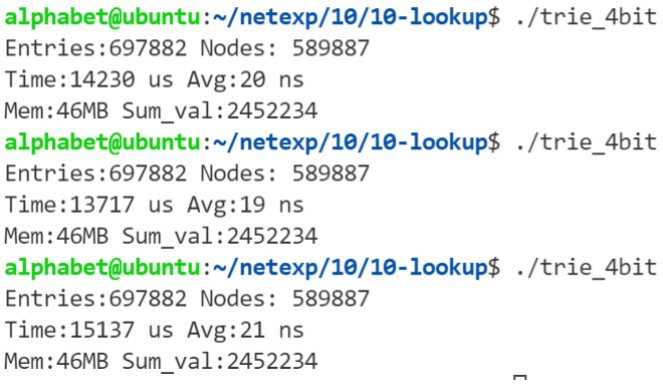

图 3

测试结果显示，4 bit 前缀树的平均查找时间为 20ns，对于约 70 万条前缀，占用内存空间为 46MB。返回的查找结果与基本前缀树相同，说明结果正确。相比基本 2 bit 前缀树，结点数更少，占用内存增大，平均查找时间减少约 35.5%。从 1 bit 前缀树到 4 bit 前缀树，查找时间减少 55.6%。