计算机网络实验 9 报告

学号　2017K8009929059
姓名　於修远

高效 IP 路由查找实验

一、实验内容

- 实现最基本的前缀树查找。
- 调研并实现某种 IP 前缀查找方案。
- 对比基本前缀树和所实现 IP 前缀查找的性能。

二、实验流程

（一）基本前缀树实现

1、插入操作

插入的第一阶段类似于查找。基本的前缀树构建基于这样的思想：首先搜索得到前缀匹配的最大路径，这一路径的终点无法找到匹配下一位的路径。在此之后，以这个终点为起点，每次开辟一个结点，直至完成一条可以代表目标条目的路径。树中每个结点的两个子结点分别代表下一位为 0 或 1 时的下一步路径。

```
u32 bit_idx = 0, bit_chk = 0x80000000;
node_t *current = root;
while (depth++ < mask) {
    bit_idx = (bit_chk & dest) >> (32 - depth);
    if (current->son[bit_idx] == NULL) {
        got = 0;
        break;
    }
    current = current->son[bit_idx];
    bit_chk = bit_chk >> 1;
}
if (got) {...}
```

随后的每一步都需要开辟新的节点。新增条目时，与之前的条目共享上半部分的结点，不存在的下半部分结点都需要自行申请和连接。

```
while (depth < mask) {
    current->son[bit_idx] = malloc_base();
    current = current->son[bit_idx];
    current->son[0] = current->son[1] = NULL;
    current->match = 0;
    bit_chk = bit_chk >> 1;
    depth++;
    bit_idx = (bit_chk & dest) >> (32 - depth);
}
```

随后的每一步都需要开辟新的节点。新增条目时，与之前的条目共享上半部分的结点，不存在的下半部分结点都需要自行申请和连接。路径构建完毕后在最后的叶子结点填入本条目的内容即可。

```
while (depth < mask) {
    current->son[bit_idx] = malloc_base();
    current = current->son[bit_idx];
    current->son[0] = current->son[1] = NULL;
    current->match = 0;
    bit_chk = bit_chk >> 1;
    depth++;
    bit_idx = (bit_chk & dest) >> (32 - depth);
}
```

2、查找操作

查找过程中的搜索方式与插入时类似，不赘述。值得注意的是，为了保证“最大匹配”，搜索应当尽可能深地进行。一旦在搜索路径上得到一个标注了` match `的结点，就以其中的记录更新查找结果，直至路径的尽头，返回的就是最后一个匹配的结果。

```
node_t *get_base(node_t* root, u32 dest) {
    int depth = 0;
    u32 bit_idx = 0, bit_chk = 0x80000000;
    node_t *current = root, *result = NULL;
    while (depth++ < 32) {
        bit_idx = (bit_chk & dest) >> (32 - depth);
        if (current->son[bit_idx] == NULL) break;
        current = current->son[bit_idx];
        if (current->match) result = current;
        bit_chk = bit_chk >> 1;
    }
    return result;
}
```

（二）Poptrie 实现

1、插入操作

Poptrie 方法的核心思想类似于多比特前缀树，目的在于减少内存需求。但是由于这会引入反复位移并确认等操作，所以不加优化的多比特前缀树的时间开销很可能会比较大；不合理的结构组织方式也无法实现节省内存的目的。针对前者，Poptrie 采用了` direct pointing `方法，通过少量的内存开销来进行一定程度的效率提高；针对后者，子结点合并等操作可以在很大程度上节约内存。

Poptrie 的插入操作中，第一步是定位到最末端的中间结点。与论文中相同，这里采用了每层 6 位的压缩程度，每次取出目的 IP 的高 6 位作为下一层的索引，同时目的 IP 左移 6 位，保证可以逐级取出索引值。在当前中间结点，分别用` vector `和` leafvec `指示子结点类型/结点计数。其中，若` vector `的第 k 位为 1，则代表一个中间子结点；反之代表一个叶子结点。相同且连续的叶子结点会被合并，` leafvec `中置 1 表示一个叶子结点。基于这些特性，随着对上述两个值进行移位操作，并统计其中为 1 的位的个数，就可以得到特定位所表示的结点类型和偏移量，如下图所示。

```
for (; mask_copy > K; mask_copy -= K) {
    key = (dest & SEC_MASK) >> (32 - K);
    dest = dest << K;
    idx[0] = idx[1] = 0;
    u64 vector = current->vector;
    u64 leafvec = current->leafvec;
    u16 bit, lbit;
    for (int i = 0; i < key; ++i) {
        bit = vector & 0x1;
        lbit = leafvec & 0x1;
        idx[bit] += (bit | lbit);
        vector = vector >> 1;
        leafvec = leafvec >> 1;
    }
    bit = vector & 0x1;
    lbit = leafvec & 0x1;
    idx[bit] += (bit | lbit);
    if (!bit) {...}
    current = current->child;
    current += idx[1] - bit;
    level++;
}
```

若子结点为中间节点（即 vector 中对应位为 1），直接索引即可；否则需要一系列创建新结点的工作，包括将 vector 中对应位置 1、申请新的中间结点空间等。值得注意的是，由于本质上是将一个叶结点扩展为了中间结点，所以新生成的中间结点应当默认为“子结点有且只有一个叶子结点”，且其中的记录与原叶子结点相同。

```
int one_bit = count_one_bit(current->vector);
u64 set_bit = 0x1L;
set_bit = set_bit << key;
current->vector |= set_bit;
current->leafvec &= (~set_bit);
u32 iface_tmp = (current->leaf + idx[0] - 1)->iface;
u32 mask_tmp = (current->leaf + idx[0] - 1)->mask;
if (lbit) {...}
internal_t *new_child = malloc_internal( size: one_bit + 1);
memcpy(new_child, current->child,  n: sizeof(internal_t) * idx[1]);
memcpy( dest: new_child + idx[1] + 1,  src: current->child + idx[1],
        n: sizeof(internal_t) * (one_bit - idx[1]));
internal_t *new_internal = new_child + idx[1];
new_internal->leafvec = 0x1;
new_internal->leaf = malloc_leaf( size: 1);
new_internal->leaf->iface = iface_tmp;
new_internal->leaf->mask = mask_tmp;
free(current->child);
if (level == 1) renew_pointing(head, current->vector, new_child);
current->child = new_child;
counter_leaf++;
```

事实上，扩展叶子结点时还有两种可能的情况：该位对应的` leafvec `标志为 0 或 1。若为 0，虽然在意义上会将叶子结点扩展，但不会涉及当前节点叶子节点序列的变化；反之则不然。此时，首先要判断后续是否有共享同一记录的叶子结点。若有，将下一个` vector `为 0 的位在` leafvec `中对应记录置 1 即可。否则，才意味着需要删除一项叶结点记录。新的叶结点序列为删除该记录后的前后两段原纪录拼接而成。具体实现如下图所示：

```
for (i = key + 1; i < 64; ++i) {
    vector = vector >> 1;
    leafvec = leafvec >> 1;
    tail_l = leafvec & 0x1;
    tail_v = vector & 0x1;
    if (!tail_v) break;
}
if (!tail_v && !tail_l) {
    set_bit = 0x1L;
    set_bit = set_bit << i;
    current->leafvec |= set_bit;
} else {
    int l_one_bit = count_one_bit(current->leafvec);
    leaf_t *new_leaf = malloc_leaf(l_one_bit);
    memcpy(new_leaf, current->leaf,  n: sizeof(leaf_t) * (idx[0] - 1));
    memcpy( dest: new_leaf + idx[0] - 1,  src: current->leaf + idx[0],
            n: sizeof(leaf_t) * (l_one_bit - idx[0] + 1));
    free(current->leaf);
    current->leaf = new_leaf;
    counter_leaf--;
}
```

至此，已经完成了“最末中间结点”的检索工作，下一步是重新组织叶子结点。由于这一步操作可能由于结点信息的更新与否而变得十分复杂，所以用一个专门的 cache 区来缓存组织新的叶子结点序列。根据设计，此时进行掩码操作后目的地址的剩余有效位数应当不大于 6 位，且至少有 1 位。设这一有效位数为 k，依照掩码的定义可知，在该叶子结点的 64 个指向中，将有 2^{6-k} 个指向归于本记录（因为低 $6-k$ 位会被掩码消除）。具体而言，设剩余有效位数中的索引值位 x，则索引值不小于 x，且小于 $x+2^{6-k}$ 的指向都应归于本记录。这也是新叶子结点处理复杂的原因：之前的操作最多删除一个叶子结点，这里可能会有多次操作并产生不连续的叶子结点序列。相关代码如下图所示。

值得一提的是，在记录有效范围内，如果出现的第一个叶子结点不开启一个新的记录（即` leafvec `中对应位为 0），也会在新的叶子结点组中衍生出新的记录。此外，此处的记录只有在原记录的掩码位数更小时才可以进行，否则无法满足最长匹配的要求。

```
for (int i = 0; i < key; ++i) {
    bit = vector & 0x1;
    lbit = leafvec & 0x1;
    if (!bit && lbit) {...}
    vector = vector >> 1;
    leafvec = leafvec >> 1;
}
int first = 1;
u64 bit_edit = 0x1L << key;
for (int i = 0; i < range; ++i) {
    bit = vector & 0x1;
    lbit = leafvec & 0x1;
    if (!bit && (first || lbit)) {...}
    vector = vector >> 1;
    leafvec = leafvec >> 1;
    bit_edit = bit_edit << 1;
}
```

最后，还需要遍历一次剩余的` vector `位，将其中的原叶子结点记录项移入 cache 中。在这里，同样要注意，如果上一步中有修改记录，第一次遇到的` vector `和` leafvec `均为 0 的位需要新开一条独立记录，因为这意味着之前共享的叶子结点被截断或修改，已不可复用。这些细节都完成后，可以将 cache 中存储的记录移出，替换为当前节点的新叶子结点，一次插入操作也就此完成。

```
for (i = key + range; i < 64; ++i) {
    lbit = leafvec & 0x1;
    bit = vector & 0x1;
    if (!bit) {...}
    vector = vector >> 1;
    leafvec = leafvec >> 1;
    bit_edit = bit_edit << 1;
}

leaf_t *new_leaf = malloc_leaf(cache_idx);
memcpy(new_leaf, cache,  n: sizeof(leaf_t) * cache_idx);
free(current->leaf);
current->leaf = new_leaf;
counter_leaf += (cache_idx - idx[0]);
```

2、查找操作

与基本的前缀树实现相似，查找操作可以看作不考虑新增结点情况的插入操作，搜索到叶结点时停止并返回即可。其计算子结点索引值、循环遍历等方式也与插入操作类似，在此不赘述。核心代码如下图所示。

```
while (mask > 0) {
    key = (dest & SEC_MASK) >> (32 - K);
    mask -= K;
    dest = dest << K;
    idx[0] = idx[1] = 0;
    u64 vector = current->vector;
    u64 leafvec = current->leafvec;
    u16 bit, lbit;
    for (int i = 0; i < key; ++i) {...}
    bit = vector & 0x1;
    lbit = leafvec & 0x1;
    idx[bit] += (bit | lbit);
    if (!bit) {
        result = current->leaf;
        result += idx[0] - 1;
        return result;
    } else {
        current = current->child;
        //if (!current) return NULL;
        current += idx[1] - 1;
    }
}
return result;
```

值得一提的是，本实验中还实现了论文中的` direct-pointing `预处理，即对目的地址的前 12 位建立一个直接索引来提高搜索效率。不论是插入时还是搜索时，都可以通过这一设计来减少 2 个层级的分析。考虑到 IP 地址为 32 位，每层关联 6 位时可以在 6 层内实现 Poptrie，这一设计可以减少近三分之一的耗时。这一直接索引的维护也很简单，每次在第 1 层（根为第 0 层）的节点上，向下一层开辟或更新中间子结点序列时，调用一下函数更新即可。合理性在于这些直接索引只会在这种情况下被修改，且不可能被消除。

```
static void renew_pointing(u32 head, u64 vec, internal_t *new_child) {
    u32 idx = 0;
    for (int i = 0; i < 64; ++i) {
        u32 bit = vec & 0x1;
        if (bit) {
            direct_pointing[head + i] = new_child + idx;
            idx++;
        }
        vec = vec >> 1;
    }
}
```

以在查找操作中的应用为例。搜索的起点从` direct-pointing `所索引的中间结点开始。若索引值为空，则意味着这一条目会在不超过 12 位的掩码内完成搜索，所以将搜索指针重定向至根节点；否则，视作已完成前 12 位（2 层）的搜索，并对后续循环中的相关变量做出对应修改即可。

```
internal_t *current = direct_pointing[dest >> 20];
if (!current || mask <= 12) current = root;
else {
    mask -= 12;
    dest = dest << 12;
}
```

（三）测试模板编写

1、文件读入

已插入操作为例。如下图所示，根据宏定义决定运行基本前缀树或 Poptrie 的测试。通过` sscanf `函数对每行的条目进行读取、拆解，再合并得到目的地址、掩码位数、端口号等信息，即可调用对应的插入函数。

```
#ifdef BASE
int add_from_txt(int num, node_t *root) {
#else
int add_from_txt(int num, internal_t *root) {
#endif
    char line[256];
    int count = 0;
    FILE *fp = fopen( filename: "../forwarding-table.txt",  modes: "rb");
    if (fp == NULL) {...}
    while (!feof(fp) && !ferror(fp)) {
        strcpy(line,  src: "\n");
        fgets(line, sizeof(line), fp);
        u32 part1, part2, part3, part4, mask, iface;
        sscanf(line,  format: "%u.%u.%u.%u %u %u", &part1, &part2, &part3, &part4, &mask, &iface);
        u32 dest = (part1 << 24) | (part2 << 16) | (part3 << 8) | part4;
        // printf("ip : %08x ; mask : %u ; iface : %u\n", dest, mask, iface);
#ifdef BASE
        put_base(root, iface, mask, dest);
#else
        put_poptrie(root, dest, mask, iface);
#endif
        if (++count >= num) break;
    }
    fclose(fp);
```

2、空间计算

基本前缀树的空间计算较为简单。由于不涉及节点删除，所以每次申请空间时令计数器自增 1 即可。最后乘每个结点的空间大小，即可得到总空间开销。

Poptrie 的空间开销应当分为三部分考虑：中间结点开销、叶子结点开销、direct-pointing 开销。其中的最后一项是静态空间，开销固定为 2^{12} 个指针的空间（事实上，相对动态开销，这一静态开销几乎可以忽略）。中间结点的开销也容易计算，因为中间结点不会减少，所以每次调用申请新中间结点函数时自增 1 即可。叶子结点的空间开销计算较为复杂，需要在每个相关操作处结合具体情况修改计数。本实验中仅打印了动态空间的开销，静态开销可在分析时手动加入。

三、结果分析

（一）运行开销测试

1、基本前缀树的性能

如下图所示，使用基本前缀树存储近 70 万条 IP 路由条目，需要约 50MB 的空间，查找耗时约 0.596 秒。

```
Space cost: 52690720 Byte.
Items: 697883
Time cost: 596420 us.
```

2、Poptrie 的性能

如下图所示，使用 Poptrie 存储近 70 万条 IP 路由条目，需要约 8.5MB 的动态空间，加上静态数组开销后共计约 8.53MB，查找耗时约 0.608 秒。

```
Space cost: 8909632 Byte
Items: 697883
Time cost: 608316 us.
```

（二）结果对比与分析

通过上述比对可以看到，Poptrie 确实可以在很大程度上减小内存空间的占用，尤其是在数据量增大时，表现得尤其明显。相应的，维护树结构等操作使之增加了工作量，但在 direct-pointing 这一直接索引帮助下，又通过不大的内存开销换来了性能的可观提升，最终与基本前缀树表现相近。

事实上，本实现中的 Poptrie 至少有两个可以优化的要点。第一是大块内存的频繁申请和释放，势必对运行效率有着极大影响。如果可以使用类似 C++中的动态数组方法进行维护，或是预先进行测试，并开辟一定长度的中间结点数组空间和叶结点数组空间，后续在这两个空间中进行组装分配，效率的增长都将相当明显。第二是运行过程中大量的位移操作，如果能在硬件层面进行针对性的优化，性能的提升同样值得期待。