网络实验报告

高效路由查找

2016K8009909005 雷正宇

实验内容

实现网络路由的前缀树查找

我的具体实现为：多比特前缀树 + 多层优化

实验流程

测试集的编写

本次试验我使用了 Java。需求是实现一个可以进行路由查找的前缀树结构，它通过配置一个由子网段、前缀长度和端口构成的表来完成整棵树的构建，然后可以通过对一个 ip 地址进行匹配来得到对应的端口号或者返回打印"没有找到"的信息，如下：

```
interface Trie {
  public void put(String ip, int mask, int port);
  public int get(String ip);
}
```

具体实现中没有声明这个接口，直接将 Trie 作为了类名。Test 类提供用于配置 Trie 和测试效率的静态方法，这个方法不能用来检测正确性（正确性检测在实验结果与分析部分会提及）：

```
public class Test {
    private static final int testNumber = 398765;

    public static void main(String[] args) {
        String testFile = "./test-list.txt";
        String config = "./forwarding-table.txt";
        Trie trie = new Trie();
        try {
            makeTestFile(config, testFile);
            makeTrieConfig(trie, config);
            long time = lookUpTest(trie, testFile);

            System.out.println("average time for one lookup: " + time * 1000000 / testNumber + " ns");
        } catch (IOException e) {
            System.out.println("wtf? It is impossible");
        }
```

```
    }
    ...
}
```

其中，makeTestFile 用来创建测试信息：

```
private static void makeTestFile(String config, String filename) throws
IOException {
    PrintWriter pw = new PrintWriter(new FileWriter(filename));
    BufferedReader br = new BufferedReader(new FileReader(config));

    for (int i = 0; i < testNumber; i++) {
        String[] strings = br.readLine().split(" ");
        int ip = Trie.parseIpString(strings[0]);
        int mask = Integer.parseInt(strings[1]);
        pw.println(Trie.parseIpNumber(makeRandomIp(ip, mask)));
    }

    br.close();
    pw.close();
}
```

Test 定义了一个常量 testNumber. 实验中给出的 forwarding-table 的总量为 697882 行，实验中可以取一个子集用来进行测试，以获得各方面效率和测试集大小的关系。代码中的 makeRandomIp 是一个根据网段和前缀长度随机生成网段中合法 IP 的函数：

```
public class Test {
  ...
    private static int makeRandomIp(int ip, int mask) {
        Random random = new Random();
        int filled;
        if (mask == 0) {
            filled = -1;
        } else if (mask == 32) {
            filled = 0;
        } else {
            filled = (1 << (31 - mask)) - 1;
        }
        return ip | (random.nextInt() & filled);
    }
  ...
}
```

makeTrieConfig 则用来根据具体文件配置 Trie:

```
public class Test {
  ...
```

```
    private static void makeTrieConfig(Trie trie, String filename) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader(filename));
        for (int i = 0; i < testNumber; i++) {
            String[] strings = br.readLine().split(" ");
            int ip = Trie.parseIpString(strings[0]);
            int mask = Integer.parseInt(strings[1]);
            int port = Integer.parseInt(strings[2]);
            trie.put(ip, mask, port);
        }
        br.close();
    }
  ...
}
```

lookUpTest 测试指定文件中所有的 IP，并把获得的端口号输出到控制台：

```
private static long lookUpTest(Trie trie, String testFile) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(testFile));
    int[] ip = new int[testNumber];
    int[] port = new int[testNumber];
    for (int i = 0; i < testNumber; i++) {
        ip[i] = Trie.parseIpString(br.readLine());
    }

    long start = System.currentTimeMillis();
    for (int i = 0; i < testNumber; i++) {
        port[i] = trie.get(ip[i]);
    }
    long end = System.currentTimeMillis();

    for (int i = 0; i < testNumber; i++) {
        System.out.println(port[i]);
    }
    return end - start;
}
```

通过控制台打印的内容，对比配置表的端口号，可以初步检查正确性。

一个初步的实现

第一个实现是简单的 1bit 前缀树，只考虑结果的正确性，不考虑效率。查找方式用递归算法。以下代码中删去了 setter/getter 和几个构造方法的具体实现：

```
class BinTrieNode {
    private Subnet body;
    private boolean valid = false;
```

```
    private int port;
    private BinTrieNode subNode0 = null;
    private BinTrieNode subNode1 = null;

    private void addPort(int port) {
        valid = true;
        this.port = port;
    }

    void put(Subnet subnet, int port) {
        if (body.getMask() == subnet.getMask()) {
            if (valid) {
                System.out.println("bad things happened");
            } else {
                addPort(port);
            }
        } else {
            int index = 31 - body.getMask();
            String newIp = Subnet.asString(Subnet.asNumber(body.getIp()) +
(1 << index));
            if (subnet.getByIndex(index)) {
                if (subNode1 == null) {
                    subNode1 = new BinTrieNode(newIp, body.getMask() + 1);
                }
                subNode1.put(subnet, port);
            } else {
                if (subNode0 == null) {
                    subNode0 = new BinTrieNode(body.getIp(),
body.getMask() + 1);
                }
                subNode0.put(subnet, port);
            }
        }
    }
}

public class BinTrie {
    private BinTrieNode root;

    public BinTrie() {
        this.root = new BinTrieNode("0.0.0.0", 0);
    }

    public void put(String ip, int mask, int port) {
        root.put(new Subnet(ip, mask), port);
    }

    public int get(String ip) {
        int port = 0;
```

```
        int ipNumber = Subnet.asNumber(ip);
        boolean found = false;
        BinTrieNode node = root;
        while (node != null) {
            int mask = node.getBody().getMask();
            if (node.isValid() && node.getBody().matchIp(ip)) {
                found = true;
                port = node.getPort();
            }
            if ((ipNumber & (1 << (31 - mask))) != 0) {
                node = node.getSubNode1();
            } else {
                node = node.getSubNode0();
            }
        }
        if (!found) {
            System.out.println(ip + " not found");
        }
        return port;
    }
}
```

其中 Subnet 是一个字段为网段和前缀长度的类，提供了用于 IP 字符串和整数相互转化的静态函数。以下代码省略了较多无关紧要的内容，例如各种 setter/getter：

```
public class Subnet {
    private String ip;
    private int mask;

    public static int asNumber(String net) {
        Integer[] parts = Arrays.stream(net.split("\\.")).
                map((s) -> Integer.parseUnsignedInt(s)).
                toArray(Integer[]::new);
        return parts[3] + (parts[2] << 8) + (parts[1] << 16) + (parts[0] << 24);
    }

    public static String asString(int ip) {
        String[] nets = new String[4];
        for (int i = 0; i < 4; i++) {
            nets[3 - i] = Integer.toString((ip >> (i * 8)) & 0xff);
        }
        return String.join(".", nets);
    }

    public boolean getByIndex(int index) {
        return (asNumber() & (1 << index)) != 0;
    }

```

```
    public boolean matchIp(String target) {
        int ipa = asNumber();
        int ipb = asNumber(target);
        return ipa >> (32 - mask) == ipb >> (32 - mask);
    }
}
```

IP 地址是32位的，Java 中没有 uint32 这样的类型，所以数字类型都是有符号的。int 类型为32位，可以用来存储完整的 IP 地址，但一定要注意用于运算时符号带来的影响。

优化策略

用 int 而不是 String 来保存数据

在初步实现中，只考虑了功能的正确性，忽视了空间消耗。所以，对于和 IP 以及网段相关的内容，在运用于 Trie 对象之前可以先转换为32位整数类型。

用多 bit 而不是 1bit 的前缀树

多 bit 前缀树会稍微牺牲一点空间，但可以在查找上换取成倍的速率。而且，在 JVM 和机器层面而言，当数据访问以 8bit 即一字节的整数倍为单位时，可以得到更好的 Cache 命中率并且在数据索引时省略一些编译器自动补充的移位操作。

叶推消除冗余节点

如果一个节点的所有子节点都已经有表项了（在我的实现中体现为 valid），就可以把这个节点删去，将所有子节点进行层次提升。如果一个节点的孩子节点存在但是无效，则把这个节点的表项下推给孩子节点。

在下面的实现中，没有删除子节点都为 valid 的节点。因为最终我发现 8bit trie 的性能让人满意，而 8bit trie 的一个节点有 256 个子节点，全部提升意味着相当繁重的内存存取。

用栈做循环而不是递归查找

递归的好处在于代码清晰简单，但效率的确不如手动用栈实现。不过本例看似是一个递归算法，其实用不着栈，因为是尾递归，优化起来很方便。

优化后的版本

根据上述策略，Trie 几乎被重写了。重写后的版本代码量也少了许多（依旧删去了一些无关紧要的部分）：

```
public class Trie {
    public static final int BITS = 8;
    public static final int FILLED;
    public static final int POW;

    static {
        POW = 1 << BITS;
        FILLED = POW - 1;
```

```
        if (32 % BITS != 0) {
            System.out.println("Bits is illegal");
        }
    }

    public static int parseIpString(String ipString) {
        Integer[] parts = Arrays.stream(ipString.split("\\.")).
                map((s) -> Integer.parseUnsignedInt(s)).
                toArray(Integer[]::new);
        return parts[3] | (parts[2] << 8) | (parts[1] << 16) | (parts[0] << 24);
    }

    public static String parseIpNumber(int ip) {
        String[] nets = new String[4];
        for (int i = 0; i < 4; i++) {
            nets[3 - i] = Integer.toString((ip >> (i * 8)) & 0xff);
        }
        return String.join(".", nets);
    }

    private TrieNode root = new TrieNode(0, 0);

    public static int getBinsByIndex(int i, int index) {
        return (i >> index) & FILLED;
    }

    public void put(String ip, int mask, int port) {
        put(parseIpString(ip), mask, port);
    }

    public void put(int ip, int mask, int port) {
        root.put(ip, mask, port);
    }

    public int get(String ip) {
        return get(parseIpString(ip));
    }

    public int get(int ip) {
        TrieNode find = root.get(ip);
        if (find == null) {
            System.out.println(parseIpNumber(ip) + " not found");
            return 0;
        } else {
            return find.getPort();
        }
    }
}
```

```
57
58  class TrieNode {
59      private int ip;
60      private int mask;
61      private boolean valid = false;
62      private int validMask = 0;
63
64      private int port = 0;
65      private TrieNode[] sub = new TrieNode[Trie.POW];
66
67      private boolean match(int ip) {
68          if (this.mask == 0) {
69              return true;
70          } else if (this.valid) {
71              return this.ip >> (32 - this.validMask) == ip >> (32 -
    this.validMask);
72          } else {
73              return this.ip >> (32 - this.mask) == ip >> (32 - this.mask);
74          }
75      }
76
77      private TrieNode getSub(int i, boolean autoCreate) {
78          if (i > Trie.POW) {
79              System.out.println("illegal access to sub");
80              return null;
81          }
82          if (i >= Trie.POW || i < 0) {
83              System.out.println("getSub a wrong number: " + i);
84              return null;
85          }
86          if (autoCreate && sub[i] == null) {
87              int newMask = this.mask + Trie.BITS;
88              sub[i] = new TrieNode(this.ip + (i << (32 - newMask)),
    newMask);
89          }
90          return sub[i];
91      }
92
93      private TrieNode getSub(int i) {
94          return getSub(i, true);
95      }
96
97      private void addPort(int validMask, int port) {
98          if (validMask <= 0 || validMask > 32) {
99              System.out.println("add port illegally");
100         }
101         valid = true;
102         this.port = port;
103         this.validMask = validMask;
```

```
    }

    void put(int ip, int mask, int port) {
        int from = Trie.getBinsByIndex(ip, 32 - this.mask - Trie.BITS);
        int to = from + (1 << (Trie.BITS - mask + this.mask));
        if (mask > this.mask) {
            if (mask >= this.mask + Trie.BITS) {
                getSub(from).put(ip, mask, port);
            } else {
                for (int i = from; i < to; i++) {
                    getSub(i).put(ip, mask, port);
                }
            }
        } else if (!valid || mask > this.validMask){
            addPort(mask, port);
        }
    }

    TrieNode get(int ip) {
        if (!match(ip)) {
            return null;
        }

        int index = Trie.getBinsByIndex(ip, 32 - this.mask - Trie.BITS);
        if (sub[index] != null && sub[index].get(ip) != null) {
            return sub[index].get(ip);
        } else if (this.valid) {
            return this;
        } else {
            return null;
        }
    }
}
```

所有的方法都是有弹性的：当 BITS 常量发生改变时，Trie 的结构会发生变化，但行为正确性不变。后面的测试中我将会展示不同 BITS 值下的效率差别。

以上代码没有包括完全消除尾递归的优化，我曾经写了如下去递归版本：

```
    public int getFaster(int ip) {
        TrieNode tn = root;
        while (tn != null) {
            if (!tn.match(ip)) {
                tn = null;
            }

            int index = getBinsByIndex(ip, 32 - tn.getMask() - BITS);
            if (tn.getSub(index, false) != null && tn.getSub(index, true)
!= null) {
```

```
                tn = tn.getSub(index, true).get(ip);
            } else if (tn.isValid()) {
                return tn.getPort();
            } else {
                tn = null;
            }
        }
        System.out.println(parseIpNumber(ip) + "not found");
        return 0;
    }

    public int getFaster(String ip) {
        return getFaster(parseIpString(ip));
    }
```

然后发现这个版本几乎没有带来任何效率上的提升，根本不"faster". 所以在最终提交的版本中将它删去了。我的猜想是，编译器已经在尾递归上做了足够的优化（现在似乎很少有不支持尾递归优化的编译器了）。

实验结果与分析

这里直接测试优化后的版本，首先是正确性分析：

```
Trie trie = new Trie();
trie.put("192.168.255.0", 24, 61666);
trie.put("192.168.0.0", 16, 12345);
trie.put("192.0.0.0", 8, 54321);
trie.testIp("192.168.100.0");
```

这个样例用来测试最长匹配规则，结果如下（我似乎难以证明这是用上面的代码跑出来的结果，如果老师有兴趣可以用我的代码进行测试）：

```
/Library/Java/JavaVirtualMachines/jdk-10.0.2.jdk/Con
12345

Process finished with exit code 0
```

对于更多的数据测试，可以查看在"实验流程#测试集的编写"部分运行结果的端口号，它的顺序和 forwarding-table 中端口号的顺序一致。

接下来分别是优化后版本的 2bit, 4bit, 8bit 的测试结果（ 测试集大小 testNumber 为398765）：

```
/Library/Java/JavaVirtualMachines/jdk-10.0.2.
average time for one lookup: 31898 ns

Process finished with exit code 0
```

```
/Library/Java/JavaVirtualMachines/jdk-10.0.2.jdk/Content
average time for one lookup: 692 ns

Process finished with exit code 0
```

```
/Library/Java/JavaVirtualMachines/jdk-10.0
average time for one lookup: 250 ns

Process finished with exit code 0
```

可以看出，2bit 的平均查找时间非常恐怖，达到了 30ms 以上，而 4bit 和 8bit 就控制在 1ms 以内了（数据波动不大，4bit 版本有时会达到 800ns 以上，8bit 版本则稳定在 250ns 左右）。

接下来是 8bit 版本在 testNumber 分别为 765, 8765, 98765, 398765 时的平均查找时间：

```
/Library/Java/JavaVirtualMachines/jdk-10.0.2.jdk/Content
average time for one lookup: 3921 ns

Process finished with exit code 0
```

```
/Library/Java/JavaVirtualMachines/jdk-10.0.2.
average time for one lookup: 684 ns

Process finished with exit code 0
```

```
/Library/Java/JavaVirtualMachines/jdk-10.0.2.
average time for one lookup: 415 ns

Process finished with exit code 0
```

```
/Library/Java/JavaVirtualMachines/jdk-10.0.2.jc
average time for one lookup: 240 ns

Process finished with exit code 0
```

测试集数量越大，查找速度越快对于这个结果，我也分析不出具体原因。时间获取函数自身占用的时间应该不到 1ms，不会造成很大影响。这个问题先保留着。

对于内存的使用，我为 JVM 设置的内存上限是 725 M，在 2bit、4bit Trie 版本下，即使将 forwarding-table 中的所有条目全部配置进 Trie 对象也不会发生内存溢出，但是 8bit 前缀树会在配置到第 600 000 行左右的数据时发生堆溢出。具体获得堆的使用情况有点麻烦，需要先使用 jsp 命令获得进程号，然后用 jmap 命令 dump 出堆的二进制信息，这个文件非常庞大，有 5GB。最后使用 jhat 命令进行解析，用浏览器访问http://localhost:7000/获得具体内存使用情况。

最终页面呈现如上图。这里罗列的应该是所有的 TrieNode 对象，每个占 41字节，但很难确定到底有多少个。我的电脑目前启动 jhat 后会变得相当卡顿。但是我可以通过在 TrieNode 类中增加对象计数器来统计对象个数。最终计算出的 testNumber 为 398765 时，2bit, 4bit, 8bit 的 TrieNode 所占用的总空间分别为：

```
total space: 27650 KB
```

```
total space: 35317 KB
```

```
total space: 99409 KB
```

即不到 100MB，而且测试集越大，8bit Trie 占用的额外空间比例就越小（虽然还是很耗费空间）。