手撕字典树和并查集

今日目标：

1：说出`Trie`树的构建过程

2：完成`Trie`树相关的面试题

3：说出并查集如何合并与查找

4：完成并查集的相关面试题

1、Trie树

业务场景：网站的搜索提示，返回前缀最相似的可能。

解决方案分析：如何设计存储的数据结构，能满足快速高效的前缀匹配？

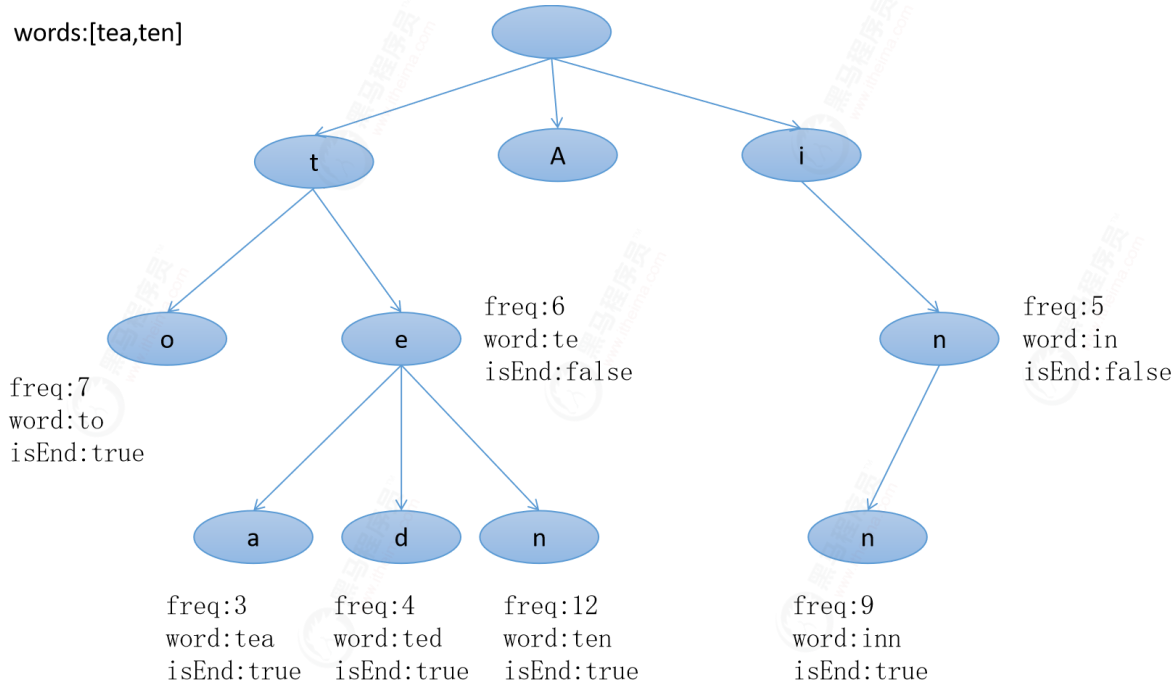
> 问题：你有什么样的思路？
>
> 1、二叉搜索树？
>
> 2、散列表？
>
> 3、堆？

接下来所讲的`Trie`树就是基于这种情况而产生的。

1.1、Trie树的定义及特性

字典树，即 `Trie` **树**，又称单词查找树或键树，也是一种树形结构，典型应用是统计和排序大量的字符串（但也不仅限于字符串），因此经常被搜索引擎系统用于文本词频统计。

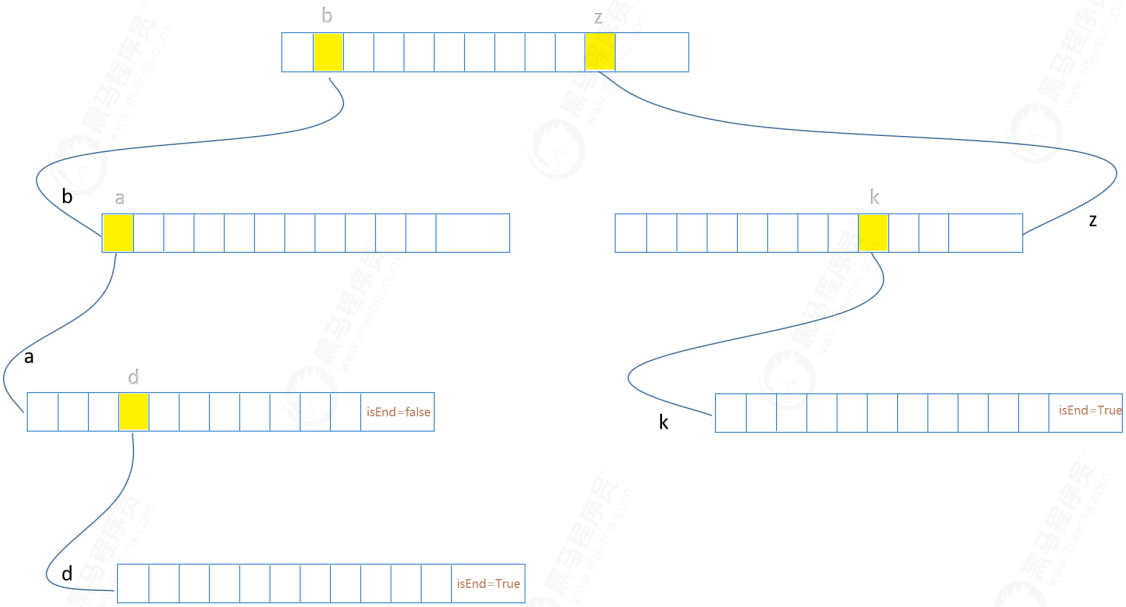
字典树如下图：



字典树的**特性**有如下几点：

- 1、节点本身不存完整单词字符串，（跟以前的树不太一样，以前树中的节点存储的都是完整的数据）
- 2、根节点不存任何数据，除根节点外，其他节点每个节点可存单词字符串中的一个字符（也可不存），当然也可根据需要存储其他的信息，比如词条出现频率等等。
- 3、从根节点到某一节点，路径上经过的字符连接起来，才为该节点对应的单词字符串。
- 4、每个节点的子节点路径代表的字符都不一样。
- 5、在 `Trie` 树中查找字符串的时间复杂度是 O(k)，k 表示要查找的字符串的长度

> **总结：**
>
> 1、trie树是一个多叉树
>
> 2、Trie 树的本质，就是利用字符串之间的公共前缀，将重复的前缀合并在一起
>
> 3、节点内部的结构如下图



应用场景：

208题的官方题解中例举了几个常见的应用场景：https://leetcode-cn.com/problems/implement-trie-prefix-tree/solution/shi-xian-trie-qian-zhui-shu-by-leetcode/

1.2、面试实战

208. 实现 Trie (前缀树)

208. 实现 Trie (前缀树)

> 可参考不同的题解进行实现，我们基于官方题解来实现
>
> https://leetcode-cn.com/problems/implement-trie-prefix-tree/solution/shi-xian-trie-qian-zhui-shu-by-leetcode/

1、节点的定义

```
class TrieNode {
    /*
        当前节点所代表的字母,当然也可以不存
        当前节点还可以存储其他所需要的信息，比如从访问频率等
    */
    // private char w;

    //指向子节点的指针
    private TrieNode[] links;

    /*
        最多R个指向子结点的链接，其中每个链接对应字母表数据集中的一个字母。
        本题R定位26，为小写拉丁字母的数量
    */
    private final int R = 26;

    //指定节点是对应单词的结尾，还是说只是单词前缀
    private boolean isEnd;

```

```
    public TrieNode() {
        links = new TrieNode[R];
    }

    /*
        判断当前节点后是否有节点ch
        即当前字母后是否有字母`ch`
    */
    public boolean containsKey(char ch) {
        return links[ch -'a'] != null;
    }
    /*
        从当前节点获取存储`ch`的节点
    */
    public TrieNode get(char ch) {
        return links[ch -'a'];
    }
    /*
        存储`ch`的映射关系，存储后表明当前字母后面多了一个字母`ch`
    */
    public void put(char ch, TrieNode node) {
        links[ch -'a'] = node;
    }

    /*
        设置当前节点为单词结尾，即这个单词到此节点就结束了
    */
    public void setEnd() {
        isEnd = true;
    }
    /*
        获取是否结束的标识
    */
    public boolean isEnd() {
        return isEnd;
    }
}
```

2、完成根节点的创建

```
//Trie树根节点
TrieNode root;

/** Initialize your data structure here. */
public Trie() {
    root = new TrieNode();
}
```

3、向`Trie`树中插入单词，

操作过程如下图所示：

判断当前节点后面是否有当前字母的节点：

- 如果没有，则在当前节点后创建代表当前字母的节点，然后进入下一节点继续判断
- 如果有，进入下一节点继续判断

```
/** Inserts a word into the trie. */
public void insert(String word) {
    if (word==null || word.length() ==0) {
        return;
    }
    char[] words = word.toCharArray();
    TrieNode node = root;
    for (int i=0;i<words.length;i++) {
        char currentChar = words[i];
        if (!node.containsKey(currentChar)) {
            node.put(currentChar,new TrieNode());
        }
        node = node.get(currentChar);
    }
    //尾节点要设置标识
    node.setEnd();
}
```

> 时间复杂度：O(m)，其中 m 为单词长度。在算法的每次迭代中，我们要么检查要么创建一个节点，直到到达键尾。只需要 m次操作。
>
> 空间复杂度：O(m)。最坏的情况下，新插入的键和 Trie 树中已有的键没有公共前缀。此时需要添加 m 个结点，使用 O(m)空间。

4、在`Trie`树中查找单词

查找的思路跟插入的思路类似

```
//找到以word为前缀的节点
public TrieNode searchPrefix(String word) {
    char[] words = word.toCharArray();
```

```
    TrieNode node = root;
    for (int i=0;i<words.length;i++) {
        char currentChar = words[i];
        if (!node.containsKey(currentChar)) {
            return null;
        }
        node = node.get(currentChar);
    }
    return node;
}



/** Returns if the word is in the trie. */
public boolean search(String word) {
    if (word==null || word.length() ==0) {
        return false;
    }
    TrieNode node =  searchPrefix(word);
    //查找word是否在trie树中
    return node!=null && node.isEnd();//isEnd=true才表明是一个完整的单词，否则表明只是前缀
}

/** Returns if there is any word in the trie that starts with the given prefix. */
public boolean startsWith(String prefix) {
    if (prefix==null || prefix.length() ==0) {
        return false;
    }
    TrieNode node =  searchPrefix(prefix);
    return node!=null;
}
```

> 时间复杂度 : O(m)。
>
> 空间复杂度 : O(1)。

212. 单词搜索 II

212. 单词搜索 II

算法分析：

- 暴力解法，在二维网格board中搜索出所有可能的单词，然后跟`words`中的单词去匹配，复杂度高，不可取。
- 使用`Trie`树的回溯搜索，配合剪枝

1、用所给的单词`words`构建`Trie`树，并且在树节点中，如果有一个节点代表了一个完整单词的结束，则将该完整的单词存储起来。

2、在二维`board`中按照每个坐标顶点去`Trie`树回溯搜索，如果搜索路径能在`Trie`中找到匹配的单词前缀或完整单词，则继续往下搜索，如果无匹配的前缀则直接返回；然后继续从下一个坐标顶点开始搜索。

这部分中，需要对`Trie`树加以改造，代码实现如下：

```
class WordTrie {

    public TrieNode root = new TrieNode();

    //根据单词构建字典树
    public void insert(String word) {
        TrieNode node = root;
        for (char c:word.toCharArray()) {
            if (!node.containsKey(c)) {
                node.put(c,new TrieNode());
            }
            node = node.get(c);
        }
        node.setEnd();
        //当前节点如果代表了一个完整单词，则将该单词存储起来
        node.setWord(word);
    }
}

//字典树节点对象
class TrieNode{
    //如果该节点代表了一个单词的结尾,则将该完整单词存储到该节点的word属性中
    public String word;
    public TrieNode[] children;
    public final int R = 26;
    public boolean isEnd;

    public TrieNode(){
        this.children = new TrieNode[R];
    }

    public boolean containsKey (char key) {
        return this.children[key-'a'] !=null;
    }

    public void put (char key,TrieNode node) {
        this.children[key-'a'] = node;
    }

    public TrieNode get (char key) {
        return this.children[key -'a'];
    }

    public void setEnd() {
        this.isEnd = true;
    }

    public boolean isEnd () {
        return this.isEnd;
    }

    public void setWord(String word) {
        this.word = word;
```

```
    }

    public String getWord () {
        return this.word;
    }
}
```

实现逻辑如下：

```
class Solution {

    int rows;
    int cols;

    public List<String> findWords(char[][] board, String[] words) {
        //根据words构建字典树
        WordTrie wordTrie = new WordTrie();
        for (String word:words) {
            wordTrie.insert(word);
        }

        //准备进行搜索board
        rows = board.length;
        cols = board[0].length;

        //定义board相关顶点是否被访问过
        boolean[][] visited = new boolean[rows][cols];
        //用来存储中间结果,用Set去重,因为从不同坐标顶点开始可能会得到相同的单词
        Set<String> result = new HashSet();
        //搜索二维数组
        TrieNode root = wordTrie.root;
        for (int i=0;i<rows;i++) {
            for (int j=0;j<cols;j++) {
                if (root.containsKey(board[i][j]) ) {
                    dfs(board,i,j,visited,root,result);
                }
            }
        }
        return new ArrayList(result);
    }



    /*
        (row,col) board中的坐标
    */
    public void dfs (char[][] board,int row,int col,boolean[][]
visited,TrieNode currentNode,Set<String> result) {
        //如果(row,col)不在二维board内 或者 该顶点已被访问过则返回
        if (!inBoard(row,col) || visited[row][col]) {
            return;
        }
        /*
            查找currentNode后是否存在字母board[row][col]
            我们不能先在board中去搜索所有的单词后再去trie树中查找前缀匹配的
            正确的做法是在dfs搜索board的过程中trie树也跟着逐层搜索，看currentNode节
点后是否存在当前(row,col)这个字母
```

```
        */
        //如果当前节点后不存在该(i,j)字母,无需下探了直接返回
        if (!currentNode.containsKey(board[row][col])) {
            return;
        }
        /*
            如果当前节点后存在该(i,j)字母,
            1、判断到当(i,j)为止是否是一个完整的单词,如果是则表明我们已经找到了在board和单词列表中同时存在的一个了，加入到结果集中
            2、继续下探且标注该(i,j)已访问过了 找到单词后不能回退，因为可能是“ad”“addd”这样的单词得继续回溯
        */
        visited[row][col] = true;
        currentNode = currentNode.get(board[row][col]);

        if (currentNode.isEnd()) {
            result.add(currentNode.getWord());
        }


        //继续下探有4个方向可以走 上下左右
        // int[] rowOffset = {-1, 0, 1, 0};
        // int[] colOffset = {0, 1, 0, -1};
        // for (int i=0;i<4;i++) {
        //     int newI = row + rowOffset[i];
        //     int newJ = col + colOffset[i];
        //     if (inBoard(newI,newJ) && !visited[newI][newJ]) {
        //         dfs(board,newI,newJ,visited,currentNode,result);
        //     }
        // }
        dfs(board,row+1,col,visited,currentNode,result);//往下
        dfs(board,row,col+1,visited,currentNode,result);//往右
        dfs(board,row-1,col,visited,currentNode,result);//往上
        dfs(board,row,col-1,visited,currentNode,result);//往左
        //最后要回退，因为下一个起点可能会用到上一个起点的字符
        visited[row][col] = false;
    }

    //判断坐标(i,j)是否在二维board中
    public boolean inBoard (int i,int j) {
        return i>=0 && i< rows && j>=0 && j<cols;
    }

}
```

2、并查集

2.1、并查集的基本实现和特性

并查集（Union Find/Disjoint Set），简单理解它体现出了两个操作：合并，查找。即可以将两个元素所在的集合合并以及查找元素所在的集合。

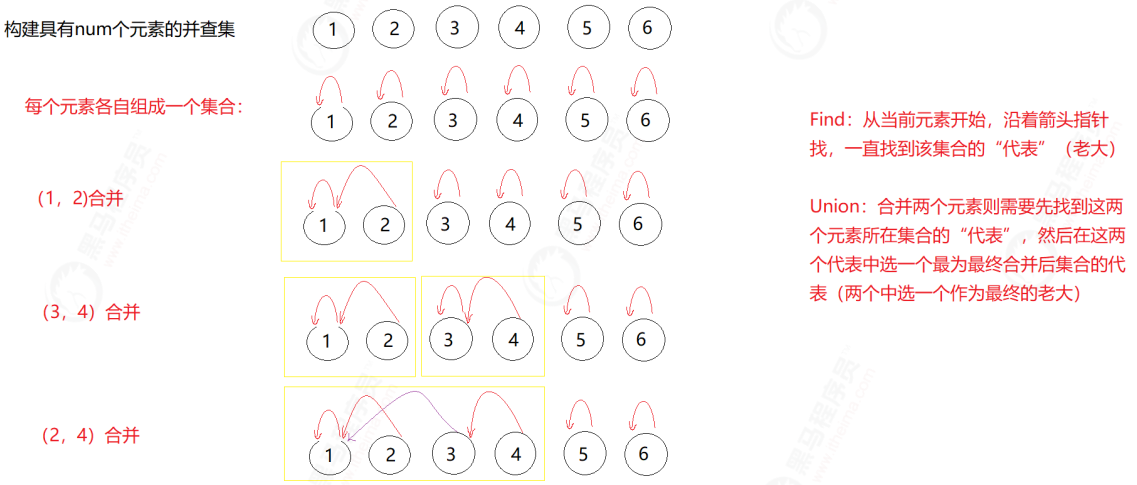
并查集一般包含以下三个操作：

`新建：make`：给定`num`个元素，建立包含`num`个元素的集合。

合并：union(x,y)：将元素x所在的集合与元素y所在的集合合并，当然如果x和y本身就在一个集合的话无需操作

查找:find(x)：查找元素x所在的集合，返回的是能代表该集合的一个代表，当然也可以用来判断两个元素是否在同一个集合，只需要判断两个元素所在集合的代表是否是同一个即可。

并查集具体的构建，合并，查找过程如下图



代码实现如下：创建：com.itheima.unionfind.UnionFind

提供四个操作：

1、初始化构建

2、查找

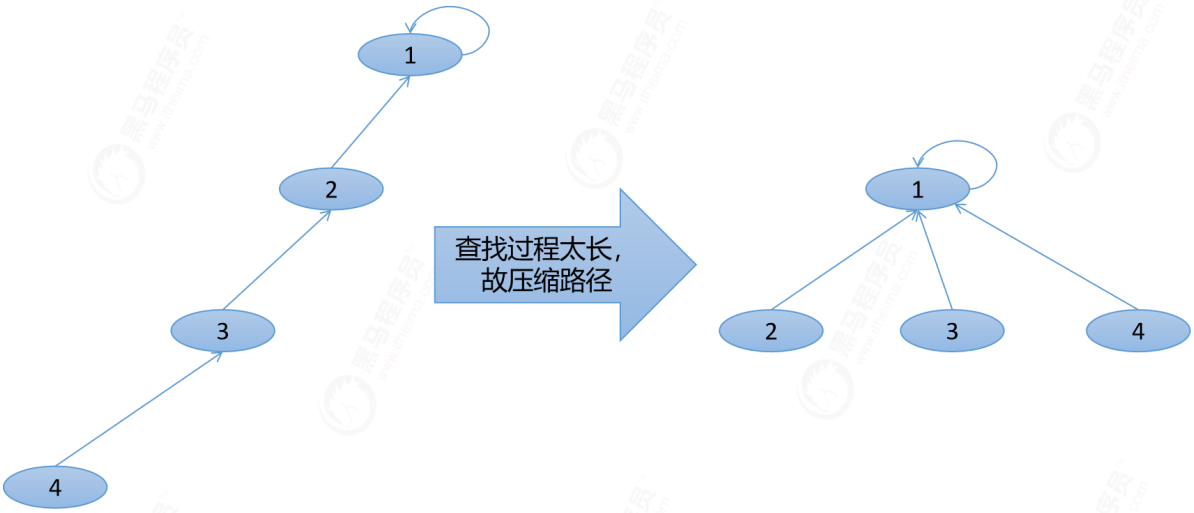
3、合并

4、获取集合个数

```java
package com.itheima.unionfind;

/**
 * 并查集
 *
 */
public class UnionFind {

    int count = 0 ;//代表该并查集中集合的个数

    int[] father; //指向父节点的箭头

    public UnionFind (int num) { // num代表元素的个数,即初始时有num个单独的集合
        //初始条件每个元素的parent都指向自己 parent[i] = i;
        father = new int[num];
        for (int i=0;i<num;i++) {
            father[i] = i;
        }
        //初始集合个数即元素个数
        count = num;
    }
```

```
22
23      /**
24       * 查找
25       * @param n
26       * @return
27       */
28      public int find (int n) { //查找n所在的集合
29          //循环版本
30          /*while (father[n] != n) {
31              n = father[n];
32          }
33          return n;*/
34
35          //递归版本
36          if (father[n] == n) {
37              return n;
38          }
39          return find(father[n]);
40      }
41
42      /**
43       * 合并
44       * @param p
45       * @param q
46       */
47      public void union (int p,int q) {
48          //先查找p,q各自所在的集合
49          int p_root = find(p);
50          int q_root = find(q);
51          //如果p,q在同一个集合无需合并
52          if (p_root == q_root) {
53              return;
54          }
55          //让p_root的父节点为q_root;
56          father[p_root] = q_root;
57
58          //集合个数减一
59          count--;
60      }
61
62      //获取集合个数
63      public int getCount(){
64          return this.count;
65      }
66
67  }
```

当然在这个过程中，还可以进行路径压缩：



路径压缩，主要影响的是查找操作，代码实现如下：

```
/**
     * 查找
     * @param n
     * @return
     */
public int find (int n) { //查找n所在的集合
    //循环版本
    /*while (father[n] != n) {

            father[n] = father[father[n]];
            n = father[n];
        }
        return n;*/

    //递归版本
    if (father[n] == n) {
        return n;
    }
    // find(father[n])查找就是当前节 父亲的父亲即爷爷
    father[n] = find(father[n]);//路径压缩版本
    return father[n];
}
```

2.2、面试实战

547. 朋友圈

字节，腾讯最近面试，547. 朋友圈

算法分析：

- 直接使用DFS,BFS，类似岛屿
- 使用并查集

可以直接套用并查集的模板代码

```
class Solution {
    public int findCircleNum(int[][] M) {
        //找到N
```

```
        int N = M.length;//也表明最多有N个朋友圈，并查集的初始元素个数可为N
        UnionFind uf = new UnionFind(N);

        //扫描矩阵，如果是朋友就合并
        for (int i=0;i<N;i++) {
            for (int j= i+1;j<N;j++) { //对称矩阵只需遍历上三角即可
                if (M[i][j] == 1) {
                    uf.union(i,j);
                }
            }
        }
        return uf.getCount();

    }
}

class UnionFind {

    int count = 0 ;//代表该并查集中集合的个数

    int[] father; //指向父节点的箭头

    public UnionFind (int num) { // num代表元素的个数,即初始时有num个单独的集合
        //初始条件每个元素的parent都指向自己 parent[i] = i;
        father = new int[num];
        for (int i=0;i<num;i++) {
            father[i] = i;
        }
        //初始集合个数即元素个数
        count = num;
    }

    /**
     * 查找
     * @param n
     * @return
     */
    public int find (int n) { //查找n所在的集合
        //循环版本
        /*while (father[n] != n) {
            father[n] = father[father[n]];
            n = father[n];
        }
        return n;*/

        //递归版本
        if (father[n] == n) {
            return n;
        }
        // find(father[n])查找就是当前节 父亲的父亲即爷爷
        father[n] = find(father[n]);//路径压缩版本
        return father[n];
    }

    /**
     * 合并
     * @param p
     * @param q
```

```
     */
    public void union (int p,int q) {
        //先查找p,q各自所在的集合
        int p_root = find(p);
        int q_root = find(q);
        //如果p,q在同一个集合无需合并
        if (p_root == q_root) {
            return;
        }
        //让p_root的父节点为q_root;
        father[p_root] = q_root;

        //集合个数减一
        count--;
    }

    //获取集合个数
    public int getCount(){
        return this.count;
    }
}
```

其他题目：

130. 被围绕的区域

200. 岛屿数量