Trie树，并查集

笔记本：	程序设计与算法		
创建时间：	2020/10/16 1:01	**更新时间：**	2020/10/20 0:41
作者：	Jie Zhong		

Trie树

基本结构
字典树，即Trie树，又称单词查找树或键树，是一种树形结构。典型应用是用于统计和排序大量字符串，经常被搜索引擎系统用于文本统计。
优点：最大限度减少无谓的字符串比较，查询效率比哈希表高。(哈希表无法进行范围查找)

三个基本性质

1. 结点本身不存完整单词
2. 从根结点到某一结点，路径上经过的字符连接起来，为该结点对应的字符串
3. 每个结点的所有子结点路径代表的字符都不相同

Trie树的核心思想是空间换时间。

Trie树模板

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

    /** Inserts a word into the trie. */
    public void insert(String word) {
        TrieNode node = root;
        for (int i = 0; i < word.length(); i++) {
            char ch = word.charAt(i);
            if (!node.containsKey(ch)) {
                node.put(ch, new TrieNode());
            }
            node = node.get(ch);
        }
        node.setEnd();
    }

    //Search a prefix or whole key in trie and
    //returns the node where search ends;
    private TrieNode searchPrefix (String word) {
        TrieNode node = root;
        for (int i = 0; i < word.length(); i++) {
            char ch = word.charAt(i);
            if (node.containsKey(ch)) {
                node = node.get(ch);
            } else {
                return null;
            }
        }
        return node;
    }


    /** Returns if the word is in the trie. */
    public boolean search(String word) {
        TrieNode node = searchPrefix(word);
```

```
        return node != null && node.isEnd();
    }

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

class TrieNode {
    //R links to node children
    private TrieNode[] links;

    private final int R = 26;

    private boolean isEnd;

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

    public boolean containsKey(char ch) {
        return links[ch - 'a'] != null;
    }

    public TrieNode get(char ch) {
        return links[ch - 'a'];
    }

    public void put(char ch, TrieNode node) {
        links[ch - 'a'] = node;
    }

    public void setEnd() {
        isEnd = true;
    }

    public boolean isEnd() {
        return isEnd;
    }
}
```

单词搜索II的时间复杂度

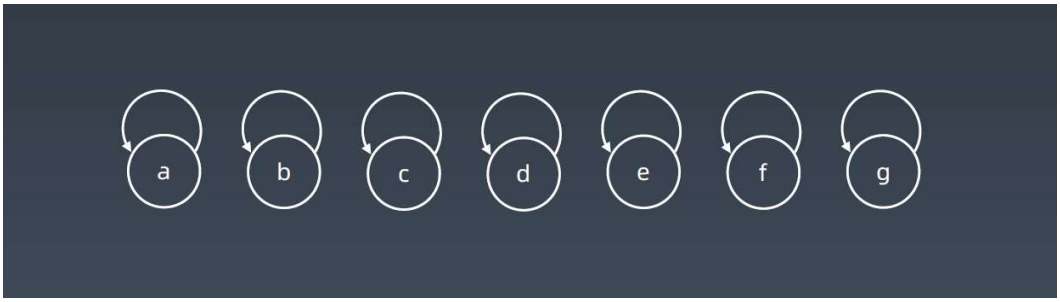
- 方法一：对每一个单词DFS， O(N*m*m*4^k)
    - N - 单词数，m * m矩阵，k - 单词平均长度
- 方法二：将所有单词放入Trie树，O(N*k+m*m) N*k - 将单词放入Trie的时间，m*m DFS棋盘的时间

并查集

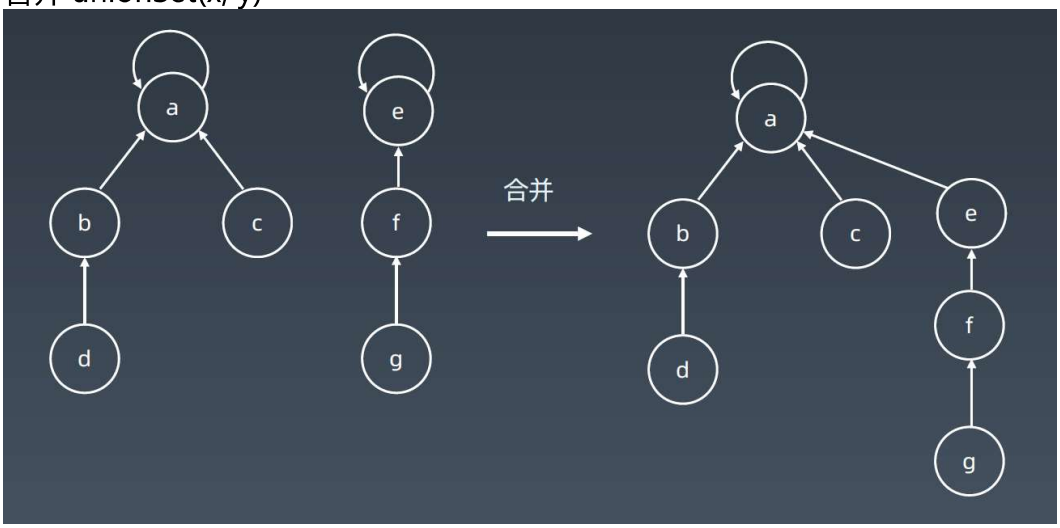
基本操作

- makeSet(s): 建立一个新的并查集，其中包括s个单元素集合
- unionSet(x, y): 把元素x和元素y所在的集合合并，要求x和y所在的集合不相交，如果相交则不合并
- find(x): 找到元素x所在的集合的代表，该操作也可以用于判断两个元素是否位于同一个集合，只要将它们各自的代表比较一下就可以了。

初始化 makeSet(s)



合并 unionSet(x, y)



查询 find(x)

- 在查询时，进行路径压缩


```
class MyUnionFind {
    private int count = 0;
    private int[] parent;
    private int[] rank;


    public MyUnionFind(char[][] grid) {
        int m = grid.length;
        int n = grid[0].length;


        parent = new int[m * n];
        rank = new int[m * n];
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
```

```
                if (grid[i][j] == '1') {
                    parent[i * n + j] = i * n + j;
                    ++count;
                }
                rank[i * n + j] = 0;
            }
        }
    }


    public int find(int p) {
        int root = p;
        while (root != parent[root]) {
            root = parent[root];
        }
        while (p != parent[p]) {
            int x = p;
            p = parent[p];
            parent[x] = root;
        }
        return root;
    }


    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP != rootQ) {
            if (rank[rootP] > rank[rootQ]) {
                parent[rootQ] = rootP;
            } else if (rank[rootP] < rank[rootQ]) {
                parent[rootP] = rootQ;
            } else {
                parent[rootQ] = rootP;
                rank[rootP]++;
            }
            count--;
        }
    }


    public int getCount() {
        return count;
    }
}
```