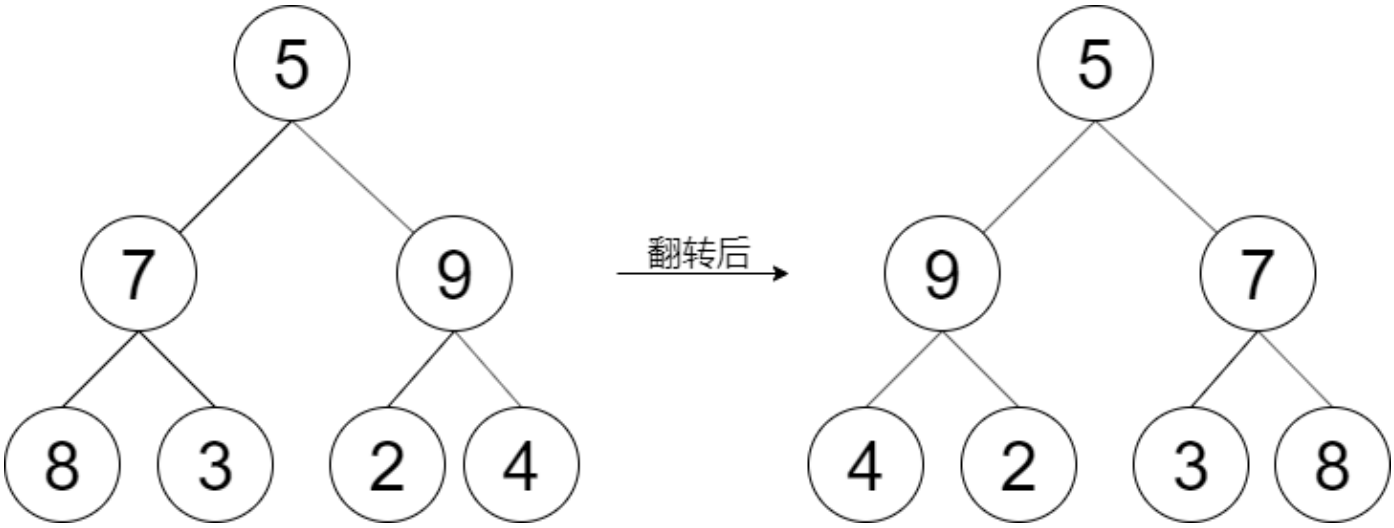
剑指offer(11)——简单+中等

题目144:

给定一棵二叉树的根节点 `root`，请左右翻转这棵二叉树，并返回其根节点。

示例 1：




```
输入：root = [5,7,9,8,3,2,4]
输出：[5,9,7,4,2,3,8]
```

提示：

- 树中节点数目范围在 `[0, 100]` 内
- `-100 <= Node.val <= 100`

我的代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* mirrorTree(TreeNode* root) {
        TreeNode* node = new TreeNode(0);
        if(root == nullptr){
            return root;
        }
        node->val = root->val;
```

```
        if(root->right != nullptr){
            node->left = mirrorTree(root->right);
        }
        if(root->left != nullptr){
            node->right = mirrorTree(root->left);
        }
        return node;
    }
};
```

思路简单的总结一下就是利用递归进行思考

解题方法1:更简单的递归方法

```
class Solution {
public:
    TreeNode* mirrorTree(TreeNode* root) {
        if (root == nullptr) return nullptr;
        TreeNode* tmp = root->left;
        root->left = mirrorTree(root->right);
        root->right = mirrorTree(tmp);
        return root;
    }
};
```

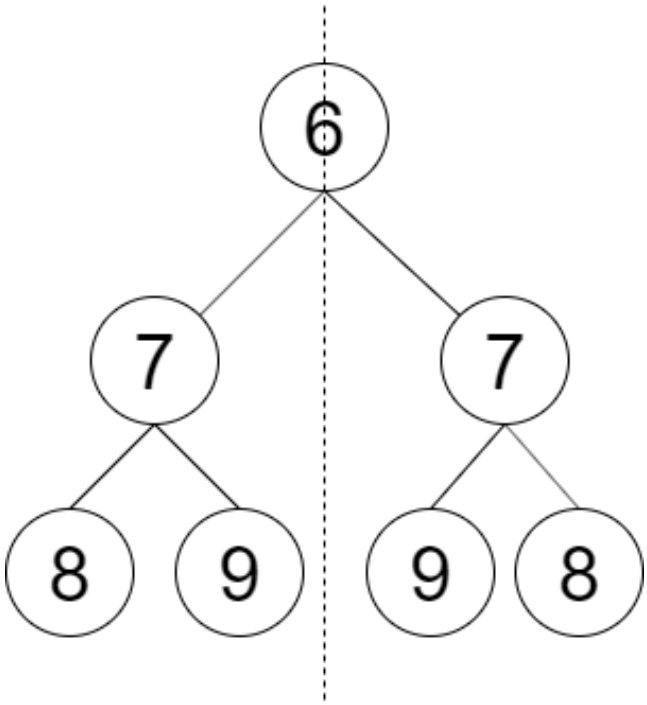
解题方法2:辅助栈或队列

```
class Solution {
public:
    TreeNode* mirrorTree(TreeNode* root) {
        if(root == nullptr) return nullptr;
        stack<TreeNode*> stack;
        stack.push(root);
        while (!stack.empty())
        {
            TreeNode* node = stack.top();
            stack.pop();
            if (node->left != nullptr) stack.push(node->left);
            if (node->right != nullptr) stack.push(node->right);
            TreeNode* tmp = node->left;
            node->left = node->right;
            node->right = tmp;
        }
        return root;
    }
};
```

题目145:

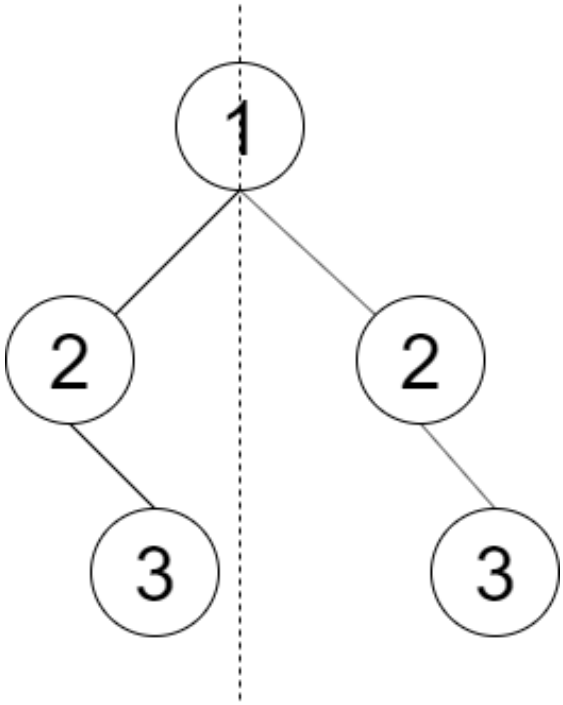
请设计一个函数判断一棵二叉树是否 **轴对称** 。

示例 1：




```
输入：root = [6,7,7,8,9,9,8]
输出：true
解释：从图中可看出树是轴对称的。
```

示例 2：




```
输入：root = [1,2,2,null,3,null,3]
输出：false
解释：从图中可看出最后一层的节点不对称。
```

提示：

```
0 <= 节点个数 <= 1000
```

注意：本题与主站 101 题相同：https://leetcode-cn.com/problems/symmetric-tree/

我的代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right)
{}
 * };
 */
class Solution {
public:
    bool checkSymmetricTree(TreeNode* root) {
        TreeNode* reverse = mirrorTree(root);
        if(root == nullptr){
            return true;
        }
        if(isSame(root, reverse)){
            return true;
        }
        return false;
    }
    TreeNode* mirrorTree(TreeNode* root) {
        TreeNode* node = new TreeNode(0);
        if(root == nullptr){
            return root;
        }
        node->val = root->val;
        if(root->right != nullptr){
            node->left = mirrorTree(root->right);
        }
        if(root->left != nullptr){
            node->right = mirrorTree(root->left);
        }
        return node;
    }
    bool isSame(TreeNode * a, TreeNode * b){
        if(a == nullptr && b == nullptr){
            return true;
        }
        if(a == nullptr) return false;
```

```
        if(b == nullptr) return true;
        if(a->val == b->val){
            return isSame(a->left,b->left) && isSame(a->right, b->right);
        }
        return false;
    }
};
```

我的方法比较愚蠢，就是将这个树翻转一下，然后看前后的两棵树是不是相同的。

解题方法1:递归

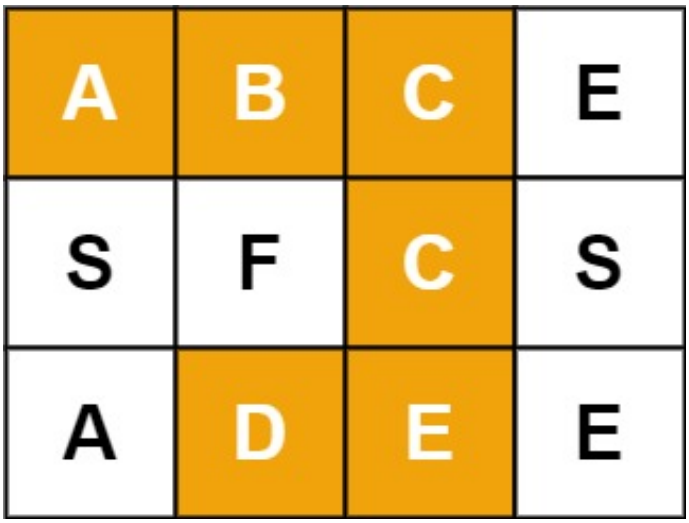
```
class Solution {
public:
    bool check(TreeNode *p, TreeNode *q) {
        if (!p && !q) return true;
        if (!p || !q) return false;
        return p->val == q->val && check(p->left, q->right) && check(p->right, q->left);
    }

    bool checkSymmetricTree(TreeNode* root) {
        return check(root, root);
    }
};
```

！递归还是需要学习的，对递归的理解还不够深入

题目129:

字母迷宫游戏初始界面记作 `m x n` 二维字符串数组 `grid`，请判断玩家是否能在 `grid` 中找到目标单词 `target`。
注意：寻找单词时 **必须** 按照字母顺序，通过水平或垂直方向相邻的单元格内的字母构成，同时，同一个单元格内的字母 **不允许被重复使用** 。



示例 1：

```
输入：grid = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], target = "ABCCED"
输出：true
```

示例 2：

```
输入：grid = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], target = "SEE"
输出：true
```

示例 3：

```
输入：grid = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], target = "ABCB"
输出：false
```

提示：

- `m == grid.length`
- `n = grid[i].length`
- `1 <= m, n <= 6`
- `1 <= target.length <= 15`
- `grid` 和 `target` 仅由大小写英文字母组成

注意：本题与主站 79 题相同：https://leetcode-cn.com/problems/word-search/

解题方法1:回溯

设函数 $\text{wordPuzzle}(i, j, k)$ 表示判断以网格的 (i, j) 位置出发，能否搜索到单词 $word[k..]$，其中 $word[k..]$ 表示字符串 $word$ 从第 k 个字符开始的后缀子串。如果能搜索到，则返回 `true`，反之返回 `false`。函数 $\text{wordPuzzle}(i, j, k)$ 的执行步骤如下：

- 如果 $grid[i][j] \neq s[k]$，当前字符不匹配，直接返回 `false`。
- 如果当前已经访问到字符串的末尾，且对应字符依然匹配，此时直接返回 `true`。
- 否则，遍历当前位置的所有相邻位置。如果从某个相邻位置出发，能够搜索到子串 $word[k+1..]$，则返回 `true`，否则返回 `false`。

这样，我们对每一个位置 (i, j) 都调用函数 $\text{wordPuzzle}(i, j, 0)$ 进行检查：只要有一处返回 `true`，就说明网格中能够找到相应的单词，否则说明不能找到。

为了防止重复遍历相同的位置，需要额外维护一个与 $grid$ 等大的 $visited$ 数组，用于标识每个位置是否被访问过。每次遍历相邻位置时，需要跳过已经被访问的位置。

```
class Solution {
public:
    bool exist(vector<vector<char>>& grid, vector<vector<int>>& visited, int i, int j,
string& s, int k) {
        if (grid[i][j] != s[k]) {
```

```
            return false;
        } else if (k == s.length() - 1) {
            return true;
        }
        visited[i][j] = true;
        vector<pair<int, int>> directions{{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
        bool result = false;
        for (const auto& dir: directions) {
            int newi = i + dir.first, newj = j + dir.second;
          // 这边我觉得很厉害，我们在使用时可以直接限定一下得到有效的位置
            if (newi >= 0 && newi < grid.size() && newj >= 0 && newj < grid[0].size()) {
                if (!visited[newi][newj]) {
                    bool flag = exist(grid, visited, newi, newj, s, k + 1);
                    if (flag) {
                        result = true;
                        break;
                    }
                }
            }
        }
        visited[i][j] = false;
        return result;
    }

    bool wordPuzzle(vector<vector<char>>& grid, string word) {
        int h = grid.size(), w = grid[0].size();
        vector<vector<int>> visited(h, vector<int>(w)); // 这边初始化的方式可以学习一下
        for (int i = 0; i < h; i++) {
            for (int j = 0; j < w; j++) {
                bool flag = exist(grid, visited, i, j, word, 0);
                if (flag) {
                    return true;
                }
            }
        }
        return false;
    }
};
```