秒杀所有岛屿题目（DFS）

200. 岛屿数量

1254. 统计封闭岛屿的数目

695. 岛屿的最大面积

1905. 统计子岛屿

130. 被围绕的区域

417. 太平洋大西洋水流问题

定义框架

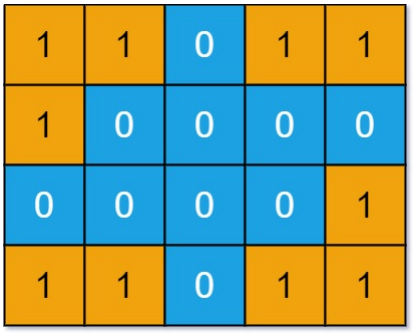
本质其实就是 DFS，**关于 DFS 详细内容可见 回溯（DFS）算法框架**，下面直接给出框架

```
// 递归：「当前节点」「该做什么」「什么时候做」
// FloodFill：如果当前位置是岛屿，则填充为海水
//    - 充当了 visited[] 的作用
private void dfs(int[][] grid, int i, int j) {
    int m = grid.length;
    int n = grid[0].length;
    // 越界检查
    if (i < 0 || i ≥ m || j < 0 || j ≥ n) return ;
    // 如果是海水
    if (grid[i][j] == 0) return ;
    // 否则：1 → 0
    grid[i][j] = 0;
    // 递归处理上下左右
    dfs(grid, i - 1, j); // 上
    dfs(grid, i + 1, j); // 下
    dfs(grid, i, j - 1); // 左
    dfs(grid, i, j + 1); // 右
}
```

岛屿数量

题目详情可见 岛屿数量

思路：把与 1 相连的区域进行 FloodFill




```
public int numIslands(char[][] grid) {
    // 记录数量
    int res = 0;
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[0].length; j++) {
            // 如果当前位置是岛屿
            // 利用 dfs 将与之相连的所有位置均值为海水
            if (grid[i][j] == '1') {
                res++;
                dfs(grid, i, j);
            }
        }
    }
    return res;
}
private void dfs(int[][] grid, int i, int j) {
    // ...
}
```

统计封闭岛屿的数目

题目详情可见 统计封闭岛屿的数目

前提：与岸边相连的岛屿不算封闭岛屿

思路：首选去除与岸边相连的岛屿，然后按照「岛屿数量」的思路计算

相似题目：飞地的数量

1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	0
1	0	1	0	1	1	1	0
1	0	0	0	0	1	0	1
1	1	1	1	1	1	1	0

```
public int closedIsland(int[][] grid) {
```

```
    // 除去上下两边
    for (int i = 0; i < grid.length; i++) {
        if (grid[i][0] == 0) dfs(grid, i, 0);
        if (grid[i][grid[0].length - 1] == 0) dfs(grid, i, grid[0].length - 1);
    }
    // 除去左右两边
    for (int j = 0; j < grid[0].length; j++) {
        if (grid[0][j] == 0) dfs(grid, 0, j);
        if (grid[grid.length - 1][j] == 0) dfs(grid, grid.length - 1, j);
    }
    // 正常计算
    int res = 0;
    for (int i = 1; i < grid.length - 1; i++) {
        for (int j = 1; j < grid[0].length - 1; j++) {
            if (grid[i][j] == 0) {
                res++;
                dfs(grid, i, j);
            }
        }
    }
    return res;
}
```

岛屿的最大面积

题目详情可见 **岛屿的最大面积**

思路： 新增一个变量记录每个岛屿的面积，然后选择出最大面积

```
private int sum = 0;
public int maxAreaOfIsland(int[][] grid) {
    int res = 0;
    int m = grid.length;
    int n = grid[0].length;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 1) {
                sum = 0;
                dfs(grid, i, j);
                // 取最值
                res = Math.max(res, sum);
            }
        }
    }
    return res;
}
private void dfs(int[][] grid, int i, int j) {
    int m = grid.length;
    int n = grid[0].length;
```

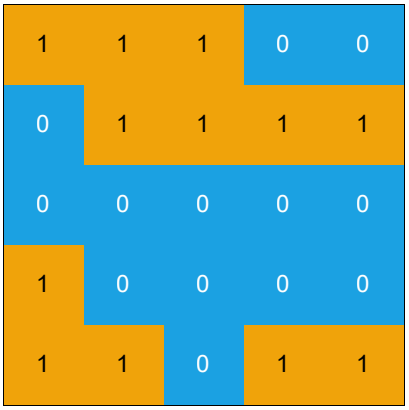
```
    if (i < 0 || i ⩾ m || j < 0 || j ⩾ n) return ;
    if (grid[i][j] == 0) return ;
    grid[i][j] = 0;
    sum ++;
    dfs(grid, i + 1, j);
    dfs(grid, i - 1, j);
    dfs(grid, i, j + 1);
    dfs(grid, i, j - 1);
}
```

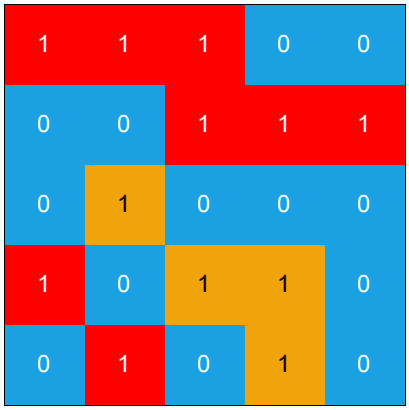
统计子岛屿

题目详情可见 **统计子岛屿**

子岛屿概念：如果 B 某一位置是岛屿，A 相同位置也必须是岛屿

思路：如果 B 是岛屿，但 A 不是岛屿，则对 B 进行 FloodFill






```
public int countSubIslands(int[][] grid1, int[][] grid2) {
    int m = grid2.length;
    int n = grid2[0].length;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid2[i][j] == 1 && grid1[i][j] ≠ 1) {
                dfs(grid2, i, j);
            }
        }
    }
    int res = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid2[i][j] == 1) {
                res++;
                dfs(grid2, i, j);
            }
        }
    }
    return res;
```

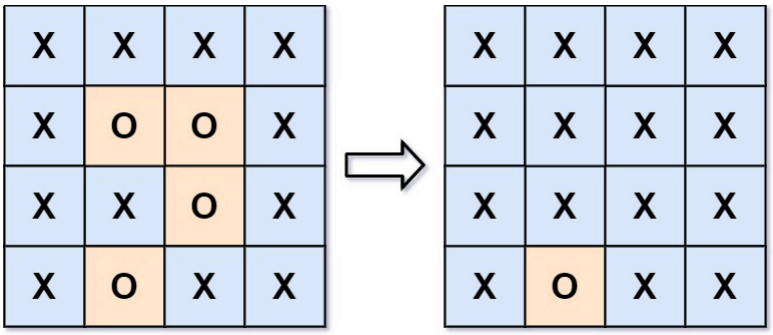
```
}
private void dfs(int[][] grid, int i, int j) {
    // ...
}
```

被围绕的区域

题目详情可见 被围绕的区域

这个题目可以用「并查集」，也可以使用「DFS」去解决，这篇文章给出 DFS 的解决方法。想要了解「并查集」的方法，**可见** 并查集 (Union-Find)

思路： 首选选择出与岸边相连的岛屿并标记为 `F` ，然后把内部封闭的岛屿全部置为 `X` ，最后把 `F` 置为 `O`



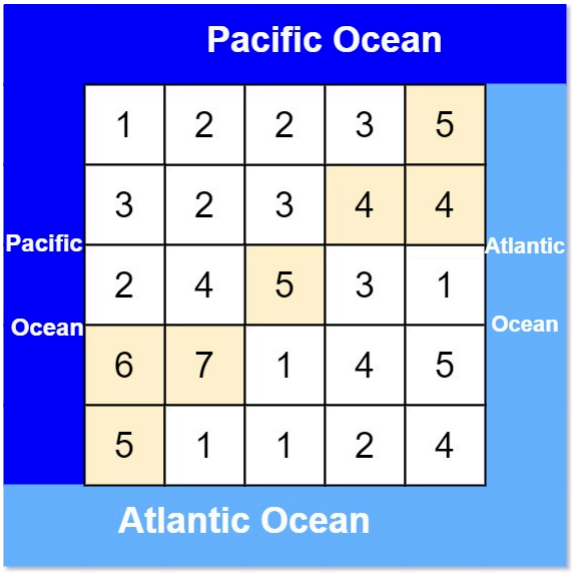

```
public void solve(char[][] board) {
    int m = board.length;
    int n = board[0].length;
    // 选择出与岸边相连的岛屿并标记为 F
    for (int i = 0; i < m; i++) {
        if (board[i][0] == 'O') dfs(board, i, 0);
        if (board[i][n - 1] == 'O') dfs(board, i, n - 1);
    }
    for (int j = 0; j < n; j++) {
        if (board[0][j] == 'O') dfs(board, 0, j);
        if (board[m - 1][j] == 'O') dfs(board, m - 1, j);
    }
    // 把内部封闭的岛屿全部置为 X，把 F 置为 O
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (board[i][j] == 'F') board[i][j] = 'O';
            else if (board[i][j] == 'O') board[i][j] = 'X';
        }
    }
}
private void dfs(char[][] board, int i, int j) {
    int m = board.length;
    int n = board[0].length;
    if (i < 0 || i ⩾ m || j < 0 || j ⩾ n) return ;
    if (board[i][j] ≠ 'O') return ;
    board[i][j] = 'F';
```

```
    dfs(board, i + 1, j);
    dfs(board, i - 1, j);
    dfs(board, i, j + 1);
    dfs(board, i, j - 1);
}
```

太平洋大西洋水流问题

题目详情可见 **太平洋大西洋水流问题**

这个题目很有意思，有意思到一开始题目都没看懂！！！麻了！！！



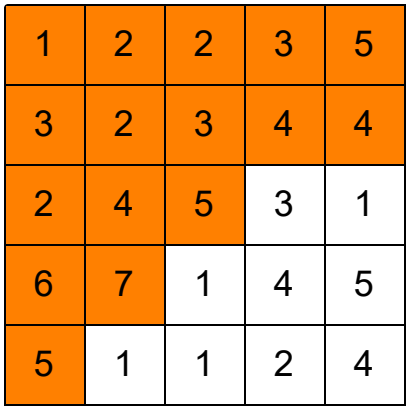
先简单解释一下这个题目的意思：与 `Ocean` 直接相连的格子中的水可以直接流入 `Ocean` 中；非直接相连的格子中的水只能流到相邻且比自己矮（相等也可以）的格子中

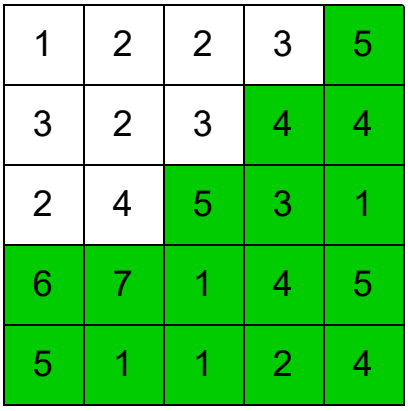
这个题目的问题是：求出既可以流入 `Pacific Ocean`，也可以流入 `Atlantic Ocean` 中的格子

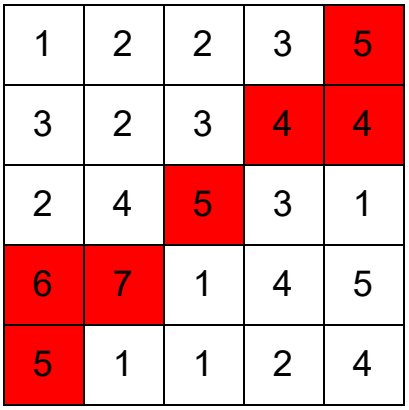
思路：

- 这个题目可以借鉴「被围绕的区域」的思路，从四个边入手。「被围绕的区域」中判断与岸边相连是通过相连格子是否等于 `0`，而我们判断与边相连需要通过高度，只有当前格子比前一格子高时才可以保证水可以流入 `Ocean`
- 还需要注意的一个点，「被围绕的区域」中直接给访问过的格子赋了新的值，可以通过新值判断是否已经访问过；但是本题中是没有改变格子原有的值，所以需要 `visited[][]` 来记录访问情况
- 我们分成两部分，首先求出可以流入 `Pacific Ocean` 的格子，然后求出可以流入 `Atlantic Ocean` 的格子，这两部分格子的重叠部分就是题目的答案

如下图所示：







```java
public List<List<Integer>> pacificAtlantic(int[][] heights) {
    int m = heights.length;
    int n = heights[0].length;
    // 记录可以流入 Pacific Ocean 的格子
    boolean[][] po = new boolean[m][n];
    // 记录可以流入 Atlantic Ocean 的格子
    boolean[][] ao = new boolean[m][n];
    // 处理第 1 行和最后 1 行
    for (int i = 0; i < n; i++) {
        dfs(heights, 0, i, po, heights[0][i]);
        dfs(heights, m - 1, i, ao, heights[m - 1][i]);
    }
    // 处理第 1 列和最后 1 列
    for (int j = 0; j < m; j++) {
        dfs(heights, j, 0, po, heights[j][0]);
        dfs(heights, j, n - 1, ao, heights[j][n - 1]);
    }
    List<List<Integer>> res = new ArrayList<>();
    // 取交集
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (po[i][j] == true && ao[i][j] == true) {
                res.add(Arrays.asList(i, j));
            }
        }
```

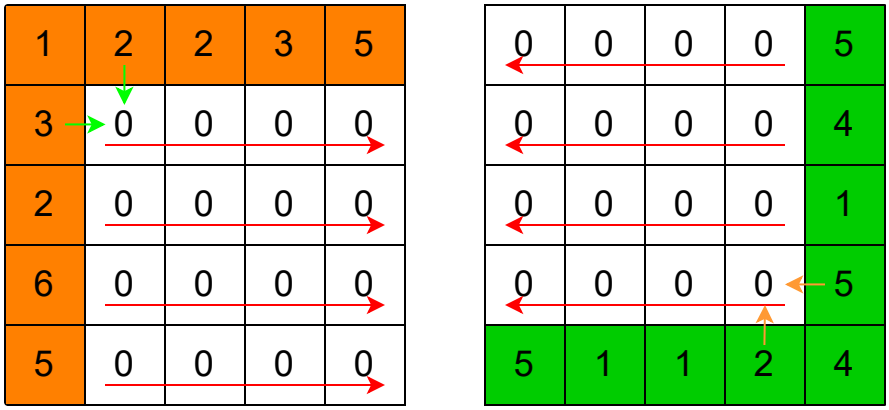
```
    }
    return res;
}
private void dfs(int[][] heights, int i, int j, boolean[][] visited, int pre) {
    int m = heights.length;
    int n = heights[0].length;
    if (i < 0 || i ⩾ m || j < 0 || j ⩾ n || visited[i][j] || pre > heights[i][j]) return ;

    visited[i][j] = true;

    dfs(heights, i + 1, j, visited, heights[i][j]);
    dfs(heights, i - 1, j, visited, heights[i][j]);
    dfs(heights, i, j + 1, visited, heights[i][j]);
    dfs(heights, i, j - 1, visited, heights[i][j]);
}
```

好了，大功告成！！！

但是还没有完，这里讲一下本人的一个错误思路！！由于最近学了「动态规划」，看到这个题目，就感觉很像，就尝试用 DP 去写了一下。**关于动态规划的详细内容可见** **动态规划解题套路框架**

我的思路，就是用两个 `dp[][]` 数组记录可以流入两个 Ocean 的格子，然后取交集就行了

关于状态转移，可以看下图：



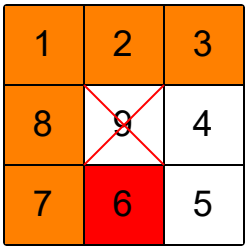
通过如上图所标注的方向来填满 `dp[][]` 数组。下面给出详细代码：

```
public List<List<Integer>> pacificAtlantic(int[][] heights) {
    int m = heights.length;
    int n = heights[0].length;
    List<List<Integer>> res = new ArrayList<>();
    int[][] podp = new int[m][n];
    int[][] aodp = new int[m][n];
    for (int i = 0; i < n; i++) {
        podp[0][i] = heights[0][i];
        aodp[m - 1][i] = heights[m - 1][i];
    }
    for (int j = 0; j < m; j++) {
        podp[j][0] = heights[j][0];
        aodp[j][n - 1] = heights[j][n - 1];
```

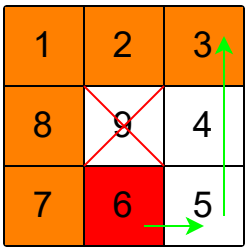
```
        }
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                int min = Math.min(podp[i - 1][j], podp[i][j - 1]);
                podp[i][j] = min > heights[i][j] ? Integer.MAX_VALUE : heights[i][j];
            }
        }
        for (int i = m - 2; i >= 0; i--) {
            for (int j = n - 2; j >= 0; j--) {
                int min = Math.min(aodp[i + 1][j], aodp[i][j + 1]);
                aodp[i][j] = min > heights[i][j] ? Integer.MAX_VALUE : heights[i][j];
            }
        }
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (podp[i][j] != Integer.MAX_VALUE && aodp[i][j] != Integer.MAX_VALUE) {
                    res.add(Arrays.asList(i, j));
                }
            }
        }
        return res;
    }
```

本来以为可以一发就过的，但是事与愿违。事实证明「动态规划」是错误的！！！

下面给出一个反例就明白了！！！



当我们处理 `(2, 1)` 的时候，根据上方和左方的状态判断显然是不可以流入 Pacific Ocean 的。但其实是错误的，我们看这样一条路径：



很明显是可以的，所以我们的状态转移方程是错误的！！！

最后这个题目貌似 DP 是写不出来的！！！也算是一个错误反例练习了一下 DP。。。