



一文秒杀所有岛屿题目

Stars 108k B站 @labuladong 配套PDF和插件 下载 打卡挑战 报名 精品课程 查看



微信搜一搜

labuladong 公众号

通知： 数据结构精品课 **V1.7** 持续更新中；第九期打卡挑战 **开始报名**；B 站可查看 **核心算法框架系列视频**。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

牛客	LeetCode	力扣	难度
-	1020. Number of Enclaves	1020. 飞地的数量	●
-	1254. Number of Closed Islands	1254. 统计封闭岛屿的数目	●
-	1905. Count Sub Islands	1905. 统计子岛屿	●
-	200. Number of Islands	200. 岛屿数量	●
-	694. Number of Distinct Islands 🔒	694. 不同岛屿的数量 🔒	●
-	695. Max Area of Island	695. 岛屿的最大面积	●
-	-	剑指 Offer II 105. 岛屿的最大面积	●

岛屿系列算法问题是经典的面试高频题，虽然基本的问题并不难，但是这类问题有一些有意思的扩展，比如求子岛屿数量，求形状不同的岛屿数量等等，本文就来把这些问题一网打尽。

岛屿系列题目的核心考点就是用 DFS/BFS 算法遍历二维数组。



的，无非就是把 DFS 改写成 BFS 而已。

那么如何在二维矩阵中使用 DFS 搜索呢？如果你把二维矩阵中的每一个位置看做一个节点，这个节点的上下左右四个位置就是相邻节点，那么整个矩阵就可以抽象成一幅网状的「图」结构。

根据 [学习数据结构和算法的框架思维](#)，完全可以根据二叉树的遍历框架改写出二维矩阵的 DFS 代码框架：

```
// 二叉树遍历框架
void traverse(TreeNode root) {
    traverse(root.left);
    traverse(root.right);
}

// 二维矩阵遍历框架
void dfs(int[][] grid, int i, int j, boolean[][] visited) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n) {
        // 超出索引边界
        return;
    }
    if (visited[i][j]) {
        // 已遍历过 (i, j)
        return;
    }
    // 进入节点 (i, j)
    visited[i][j] = true;
    dfs(grid, i - 1, j, visited); // 上
    dfs(grid, i + 1, j, visited); // 下
    dfs(grid, i, j - 1, visited); // 左
    dfs(grid, i, j + 1, visited); // 右
}
```

因为二维矩阵本质上是一幅「图」，所以遍历的过程中需要一个 `visited` 布尔数组防止走回头路，如果你能理解上面这段代码，那么搞定所有岛屿系列题目都很简单。

这里额外说一个处理二维数组的常用小技巧，你有时会看到使用「方向数组」来处理上下左右的遍历，和前文 [图遍历框架](#) 的代码很类似：



```
// 方向数组，分别代表上、下、左、右
int[][] dirs = new int[][]{{-1,0}, {1,0}, {0,-1}, {0,1}};

void dfs(int[][] grid, int i, int j, boolean[][] visited) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n) {
        // 超出索引边界
        return;
    }
    if (visited[i][j]) {
        // 已遍历过 (i, j)
        return;
    }

    // 进入节点 (i, j)
    visited[i][j] = true;
    // 递归遍历上下左右的节点
    for (int[] d : dirs) {
        int next_i = i + d[0];
        int next_j = j + d[1];
        dfs(grid, next_i, next_j, visited);
    }
    // 离开节点 (i, j)
}
```

这种写法无非就是用 for 循环处理上下左右的遍历罢了，你可以按照个人喜好选择写法。

岛屿数量

这是力扣第 200 题「[岛屿数量](#)」，最简单也是最经典的一道问题，题目会输入一个二维数组 `grid`，其中只包含 `0` 或者 `1`，`0` 代表海水，`1` 代表陆地，且假设该矩阵四周都是被海水包围着的。

我们说连成片的陆地形成岛屿，那么请你写一个算法，计算这个矩阵 `grid` 中岛屿的个数，函数签名如下：

```
int numIslands(char[][] grid);
```



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

思路很简单，关键在于如何寻找并标记「岛屿」，这就要 DFS 算法发挥作用了，我们直接看解法代码：

```
// 主函数，计算岛屿数量
int numIslands(char[][] grid) {
    int res = 0;
    int m = grid.length, n = grid[0].length;
    // 遍历 grid
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == '1') {
                // 每发现一个岛屿，岛屿数量加一
                res++;
                // 然后使用 DFS 将岛屿淹了
                dfs(grid, i, j);
            }
        }
    }
    return res;
}
```

```
// 从 (i, j) 开始，将与之相邻的陆地都变成海水
```



```

if (i < 0 || j < 0 || i >= m || j >= n) {
    // 超出索引边界
    return;
}
if (grid[i][j] == '0') {
    // 已经是海水了
    return;
}
// 将 (i, j) 变成海水
grid[i][j] = '0';
// 淹没上下左右的陆地
dfs(grid, i + 1, j);
dfs(grid, i, j + 1);
dfs(grid, i - 1, j);
dfs(grid, i, j - 1);
}

```

为什么每次遇到岛屿，都要用 DFS 算法把岛屿「淹了」呢？主要是为了省事，避免维护 `visited` 数组。

因为 `dfs` 函数遍历到值为 `0` 的位置会直接返回，所以只要把经过的位置都设置为 `0`，就可以起到不走回头路的作用。

PS：这类 DFS 算法还有个别名叫做 `FloodFill 算法`，现在有没有觉得 FloodFill 这个名字还挺贴切的~

这个最最基本的算法问题就说到这，我们来看看后面的题目有什么花样。

封闭岛屿的数量

上一题说二维矩阵四周可以认为也是被海水包围的，所以靠边的陆地也算作岛屿。

力扣第 1254 题「统计封闭岛屿的数目」和上一题有两点不同：

- 1、用 `0` 表示陆地，用 `1` 表示海水。
- 2、让你计算「封闭岛屿」的数目。所谓「封闭岛屿」就是上下左右全部被 `1` 包围的 `0`，也就是说靠边的陆地不算作「封闭岛屿」。

函数签名如下：



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

比如题目给你输入如下这个二维矩阵：

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

算法返回 2，只有图中灰色部分的 0 是四周全都被海水包围着的「封闭岛屿」。

那么如何判断「封闭岛屿」呢？其实很简单，把上一题中那些靠边的岛屿排除掉，剩下的不就是「封闭岛屿」了吗？

有了这个思路，就可以直接看代码了，注意这题规定 0 表示陆地，用 1 表示海水：

```
// 主函数：计算封闭岛屿的数量
int closedIsland(int[][] grid) {
    int m = grid.length, n = grid[0].length;
    for (int j = 0; j < n; j++) {
        // 把靠上边的岛屿淹掉
        dfs(grid, 0, j);
        // 把靠下边的岛屿淹掉
        dfs(grid, m - 1, j);
    }
    for (int i = 0; i < m; i++) {
        // 把靠左边的岛屿淹掉
        dfs(grid, i, 0);
        // 把靠右边的岛屿淹掉
        dfs(grid, i, n - 1);
    }
}
```



```

// 遍历 grid，剩下的岛屿都是封闭岛屿
int res = 0;
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (grid[i][j] == 0) {
            res++;
            dfs(grid, i, j);
        }
    }
}
return res;
}

// 从 (i, j) 开始，将与之相邻的陆地都变成海水
void dfs(int[][] grid, int i, int j) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n) {
        return;
    }
    if (grid[i][j] == 1) {
        // 已经是海水了
        return;
    }
    // 将 (i, j) 变成海水
    grid[i][j] = 1;
    // 淹没上下左右的陆地
    dfs(grid, i + 1, j);
    dfs(grid, i, j + 1);
    dfs(grid, i - 1, j);
    dfs(grid, i, j - 1);
}

```

只要提前把靠边的陆地都淹掉，然后算出来的就是封闭岛屿了。

PS：处理这类岛屿题目除了 DFS/BFS 算法之外，Union Find 并查集算法也是一种可选的方法，前文 [Union Find 算法运用](#) 就用 Union Find 算法解决了一道类似的问题。

这道岛屿题目的解法稍微改改就可以解决力扣第 1020 题「[飞地的数量](#)」，这题不让你求封闭岛屿的数量，而是求封闭岛屿的面积总和。

其实思路都是一样的，先把靠边的陆地淹掉，然后去数剩下的陆地数量就行了，注意第 1020 题中

1 代表陆地，**0** 代表海水：



```

int numEnclaves(int[][] grid) {
    int m = grid.length, n = grid[0].length;
    // 淹掉靠边的陆地
    for (int i = 0; i < m; i++) {
        dfs(grid, i, 0);
        dfs(grid, i, n - 1);
    }
    for (int j = 0; j < n; j++) {
        dfs(grid, 0, j);
        dfs(grid, m - 1, j);
    }

    // 数一数剩下的陆地
    int res = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 1) {
                res += 1;
            }
        }
    }

    return res;
}

// 和之前的实现类似
void dfs(int[][] grid, int i, int j) {
    // ...
}

```

篇幅所限，具体代码我就不写了，我们继续看其他的岛屿题目。

岛屿的最大面积

这是力扣第 695 题「[岛屿的最大面积](#)」，`0` 表示海水，`1` 表示陆地，现在不让你计算岛屿的个数了，而是让你计算最大的那个岛屿的面积，函数签名如下：

```
int maxAreaOfIsland(int[][] grid)
```




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

其中面积最大的是橘红色的岛屿，算法返回它的面积 6。

这题的大体思路和之前完全一样，只不过 `dfs` 函数淹没岛屿的同时，还应该想办法记录这个岛屿的面积。

我们可以给 `dfs` 函数设置返回值，记录每次淹没的陆地的个数，直接看解法吧：

```
int maxAreaOfIsland(int[][] grid) {  
    // 记录岛屿的最大面积  
    int res = 0;  
    int m = grid.length, n = grid[0].length;  
    for (int i = 0; i < m; i++) {  
        for (int j = 0; j < n; j++) {  
            if (grid[i][j] == 1) {  
                // 淹没岛屿，并更新最大岛屿面积  
                res = Math.max(res, dfs(grid, i, j));  
            }  
        }  
    }  
    return res;  
}
```

```
// 淹没与 (i, j) 相邻的陆地，并返回淹没的陆地面积
```



```

if (i < 0 || j < 0 || i >= m || j >= n) {
    // 超出索引边界
    return 0;
}
if (grid[i][j] == 0) {
    // 已经是海水了
    return 0;
}
// 将 (i, j) 变成海水
grid[i][j] = 0;

return dfs(grid, i + 1, j)
    + dfs(grid, i, j + 1)
    + dfs(grid, i - 1, j)
    + dfs(grid, i, j - 1) + 1;
}

```

解法和之前相比差不多，我也不多说了，接下来的两道岛屿题目是比较有技巧性的，我们重点来看一下。

子岛屿数量

如果说前面的题目都是模板题，那么力扣第 1905 题「统计子岛屿」可能得动动脑子了：

1905. 统计子岛屿

难度 中等

👍 22

☆ 收藏

🔗 分享

🌐 切换为英文

🔔 接收动态

📝 反馈

给你两个 $m \times n$ 的二进制矩阵 `grid1` 和 `grid2`，它们只包含 0（表示水域）和 1（表示陆地）。一个 **岛屿** 是由 **四个方向**（水平或者竖直）上相邻的 1 组成的区域。任何矩阵以外的区域都视为水域。

如果 `grid2` 的一个岛屿，被 `grid1` 的一个岛屿 **完全** 包含，也就是说 `grid2` 中该岛屿的每一个格子都被 `grid1` 中 **同一个** 岛屿完全包含，那么我们称 `grid2` 中的这个岛屿为 **子岛屿**。

请你返回 `grid2` 中 **子岛屿** 的数目。

示例 1：

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

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



输入: grid1 = [[1,1,1,0,0],[0,1,1,1,1],[0,0,0,0,0],[1,0,0,0,0],
[1,1,0,1,1]], grid2 = [[1,1,1,0,0],[0,0,1,1,1],[0,1,0,0,0],
[1,0,1,1,0],[0,1,0,1,0]]

输出: 3

解释: 如上图所示, 左边为 grid1, 右边为 grid2。
grid2 中标红的 1 区域是子岛屿, 总共有 3 个子岛屿。

这道题的关键在于, 如何快速判断子岛屿? 肯定可以借助 [Union Find 并查集算法](#) 来判断, 不过本文重点在 DFS 算法, 就不展开并查集算法了。

什么情况下 `grid2` 中的一个岛屿 B 是 `grid1` 中的一个岛屿 A 的子岛?

当岛屿 B 中所有陆地在岛屿 A 中也是陆地的时候, 岛屿 B 是岛屿 A 的子岛。

反过来说, 如果岛屿 B 中存在一片陆地, 在岛屿 A 的对应位置是海水, 那么岛屿 B 就不是岛屿 A 的子岛。

那么, 我们只要遍历 `grid2` 中的所有岛屿, 把那些不可能是子岛的岛屿排除掉, 剩下的就是子岛。

依据这个思路, 可以直接写出下面的代码:

```
int countSubIslands(int[][] grid1, int[][] grid2) {
    int m = grid1.length, n = grid1[0].length;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid1[i][j] == 0 && grid2[i][j] == 1) {
                // 这个岛屿肯定不是子岛, 淹掉
                dfs(grid2, i, j);
            }
        }
    }
    // 现在 grid2 中剩下的岛屿都是子岛, 计算岛屿数量
    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;
    }

    // 从 (i, j) 开始, 将与之相邻的陆地都变成海水
    void dfs(int[][] grid, int i, int j) {
        int m = grid.length, n = grid[0].length;
        if (i < 0 || j < 0 || i >= m || j >= n) {
            return;
        }
        if (grid[i][j] == 0) {
            return;
        }

        grid[i][j] = 0;
        dfs(grid, i + 1, j);
        dfs(grid, i, j + 1);
        dfs(grid, i - 1, j);
        dfs(grid, i, j - 1);
    }
}
```

这道题的思路和计算「封闭岛屿」数量的思路有些类似, 只不过后者排除那些靠边的岛屿, 前者排除那些不可能是子岛的岛屿。

不同的岛屿数量

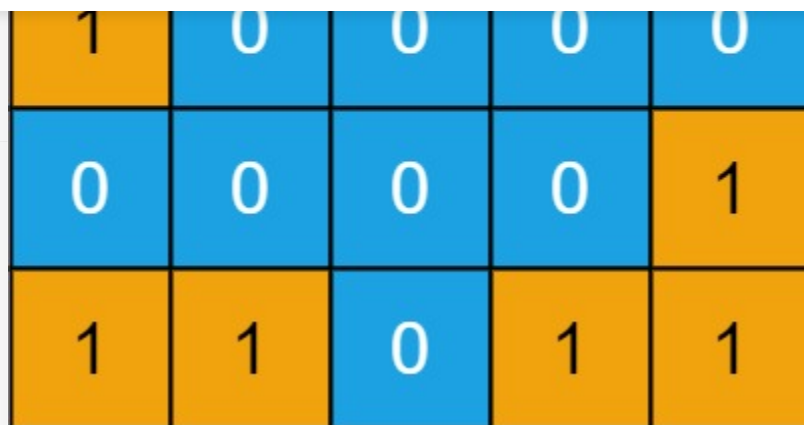
这是本文的最后一道岛屿题目, 作为压轴题, 当然是最有意思的。

力扣第 694 题「[不同的岛屿数量](#)」, 题目还是输入一个二维矩阵, `0` 表示海水, `1` 表示陆地, 这次让你计算 **不同的 (distinct)** 岛屿数量, 函数签名如下:

```
int numDistinctIslands(int[][] grid)
```

比如题目输入下面这个二维矩阵:

1	1	0	1	1
---	---	---	---	---



其中有四个岛屿，但是左下角和右上角的岛屿形状相同，所以不同的岛屿共有三个，算法返回 3。

很显然我们得想办法把二维矩阵中的「岛屿」进行转化，变成比如字符串这样的类型，然后利用 HashSet 这样的数据结构去重，最终得到不同的岛屿的个数。

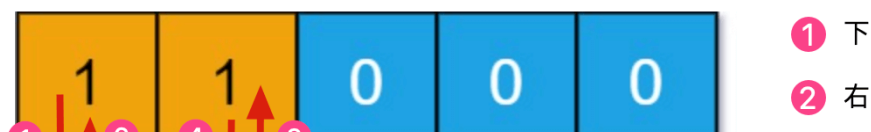
如果要把岛屿转化成字符串，说白了就是序列化，序列化说白了就是遍历嘛，前文 [二叉树的序列化和反序列化](#) 讲了二叉树和字符串互转，这里也是类似的。

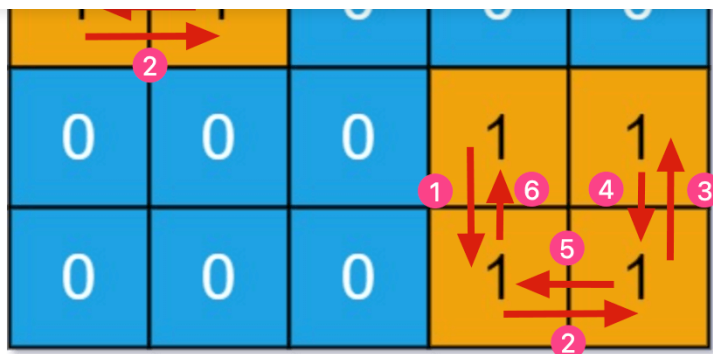
首先，对于形状相同的岛屿，如果从同一起点出发，dfs 函数遍历的顺序肯定是一样的。

因为遍历顺序是写死在你的递归函数里面的，不会动态改变：

```
void dfs(int[][] grid, int i, int j) {
    // 递归顺序：
    dfs(grid, i - 1, j); // 上
    dfs(grid, i + 1, j); // 下
    dfs(grid, i, j - 1); // 左
    dfs(grid, i, j + 1); // 右
}
```

所以，遍历顺序从某种意义上说就可以用来描述岛屿的形状，比如下图这两个岛屿：





5 撤销右

6 撤销下

假设它们的遍历顺序是：

下，右，上，撤销上，撤销右，撤销下

如果我用分别用 1, 2, 3, 4 代表上下左右，用 -1, -2, -3, -4 代表上下左右的撤销，那么可以这样表示它们的遍历顺序：

2, 4, 1, -1, -4, -2

你看，这就相当于是岛屿序列化的结果，只要每次使用 dfs 遍历岛屿的时候生成这串数字进行比较，就可以计算到底有多少个不同的岛屿了。

PS：细心的读者问到，为什么记录「撤销」操作才能唯一表示遍历顺序呢？不记录撤销操作好像也可以？实际上不是的。

比方说「下，右，撤销右，撤销下」和「下，撤销下，右，撤销右」显然是两个不同的遍历顺序，但如果不记录撤销操作，那么它俩都是「下，右」，成了相同的遍历顺序，显然是不对的。

所以我们需要稍微改造 dfs 函数，添加一些函数参数以便记录遍历顺序：

```
void dfs(int[][] grid, int i, int j, StringBuilder sb, int dir) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n
        || grid[i][j] == 0) {
        return;
    }
```



```

grid[i][j] = 0;
sb.append(dir).append(',');

dfs(grid, i - 1, j, sb, 1); // 上
dfs(grid, i + 1, j, sb, 2); // 下
dfs(grid, i, j - 1, sb, 3); // 左
dfs(grid, i, j + 1, sb, 4); // 右

// 后序遍历位置: 离开 (i, j)
sb.append(-dir).append(',');
}

```

`dir` 记录方向, `dfs` 函数递归结束后, `sb` 记录着整个遍历顺序。有了这个 `dfs` 函数就好办了, 我们可以直接写出最后的解法代码:

```

int numDistinctIslands(int[][] grid) {
    int m = grid.length, n = grid[0].length;
    // 记录所有岛屿的序列化结果
    HashSet<String> islands = new HashSet<>();
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 1) {
                // 淹没这个岛屿, 同时存储岛屿的序列化结果
                StringBuilder sb = new StringBuilder();
                // 初始的方向可以随便写, 不影响正确性
                dfs(grid, i, j, sb, 666);
                islands.add(sb.toString());
            }
        }
    }
    // 不相同的岛屿数量
    return islands.size();
}

```

这样, 这道题就解决了, 至于为什么初始调用 `dfs` 函数时的 `dir` 参数可以随意写, 这里涉及 DFS 和回溯算法的一个细微差别, 前文 [图算法基础](#) 有写, 这里就不展开了。

以上就是全部岛屿系列题目的解题思路, 也许前面的题目大部分人会做, 但是最后两题还是比较巧妙的, 希望本文对你有帮助。



► 引用本文的文章

《labuladong 的算法小抄》已经出版，关注公众号查看详情；后台回复关键词「进群」可加入算法群；回复「PDF」可获取精华文章 PDF：



微信搜一搜

Q labuladong 公众号

共同维护高质量学习环境，评论礼仪[见这里](#)，违者直接拉黑不解释



















