LeetCode 例题精讲 | 12 岛屿问题：网格结构中的 DFS

原创 nettee 面向大象编程 4月15日

来自专辑

LeetCode 例题精讲

本期例题为 LeetCode「岛屿问题」系列：

- **LeetCode 463. Island Perimeter** 岛屿的周长（Easy）
- **LeetCode 695. Max Area of Island** 岛屿的最大面积（Medium）
- **LeetCode 827. Making A Large Island** 填海造陆（Hard）

我们所熟悉的 DFS（深度优先搜索）问题通常是在树或者图结构上进行的。而我们今天要讨论的 DFS 问题，是在一种「网格」结构中进行的。岛屿问题是这类网格 DFS 问题的典型代表。网格结构遍历起来要比二叉树复杂一些，如果没有掌握一定的方法，DFS 代码容易写得冗长繁杂。

本文将以岛屿问题为例，展示网格类问题 DFS 通用思路，以及如何让代码变得简洁。主要内容包括：

- 网格类问题的基本性质
- 在网格中进行 DFS 遍历的方法与技巧
- 三个岛屿问题的解法
- 相关题目

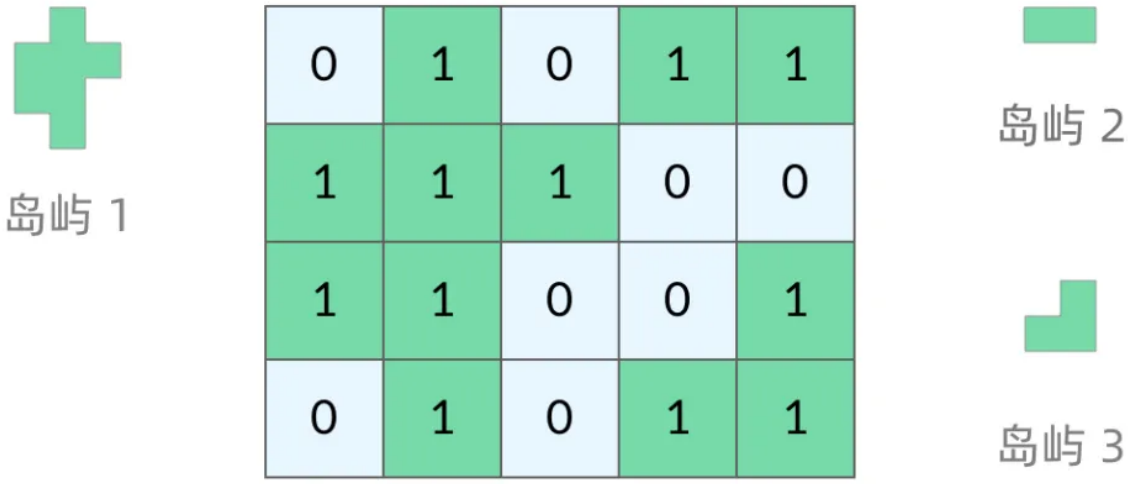
网格类问题的 DFS 遍历方法

网格问题的基本概念

我们首先明确一下岛屿问题中的网格结构是如何定义的，以方便我们后面的讨论。

网格问题是由 $m \times n$ 个小方格组成一个网格，每个小方格与其上下左右四个方格认为是相邻的，要在这样的网格上进行某种搜索。

岛屿问题是一类典型的网格问题。每个格子中的数字可能是 0 或者 1。我们把数字为 0 的格子看成海洋格子，数字为 1 的格子看成陆地格子，这样相邻的陆地格子就连接成一个岛屿。



岛屿问题示例

在这样一个设定下，就出现了各种岛屿问题的变种，包括岛屿的数量、面积、周长等。不过这些问题，基本都可以用 DFS 遍历来解决。

DFS 的基本结构

网格结构要比二叉树结构稍微复杂一些，它其实是一种简化版的图结构。要写好网格上的 DFS 遍历，我们首先要理解二叉树上的 DFS 遍历方法，再类比写出网格结构上的 DFS 遍历。我们写的二叉树 DFS 遍历一般是这样的：

```
void traverse(TreeNode root) {
    // 判断 base case
    if (root == null) {
        return;
    }
    // 访问两个相邻结点：左子结点、右子结点
    traverse(root.left);
    traverse(root.right);
}
```

可以看到，二叉树的 DFS 有两个要素：「**访问相邻结点**」和「**判断 base case**」。

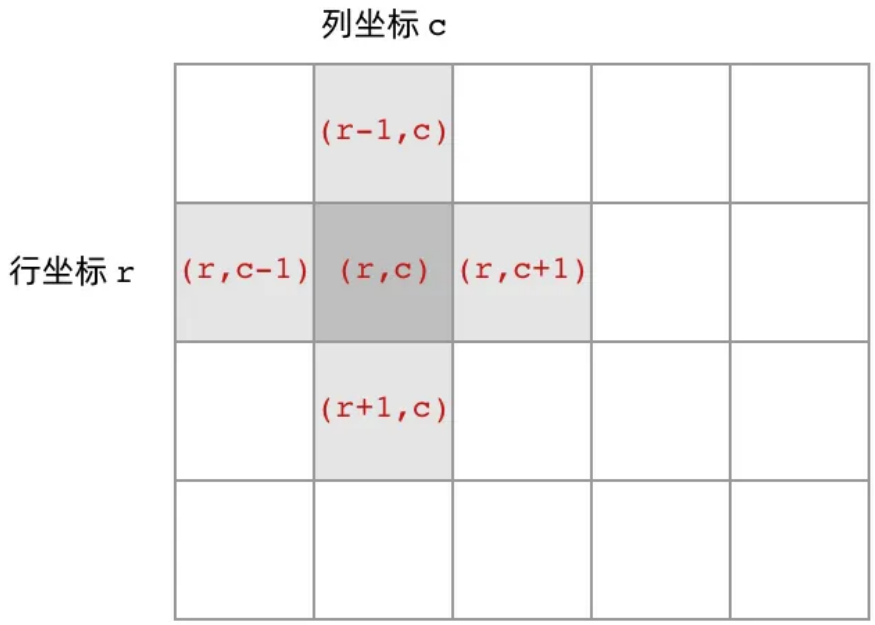
第一个要素是**访问相邻结点**。二叉树的相邻结点非常简单，只有左子结点和右子结点两个。二叉树本身就是一个递归定义的结构：一棵二叉树，它的左子树和右子树也是一棵二叉树。那么我们的 DFS 遍历只需要递归调用左子树和右子树即可。

第二个要素是 **判断 base case**。一般来说，二叉树遍历的 base case 是 `root == null` 。这样一个条件判断其实有两个含义：一方面，这表示 `root` 指向的子树为空，不需要再往下遍历了。

另一方面，在 `root == null` 的时候及时返回，可以让后面的 `root.left` 和 `root.right` 操作不会出现空指针异常。

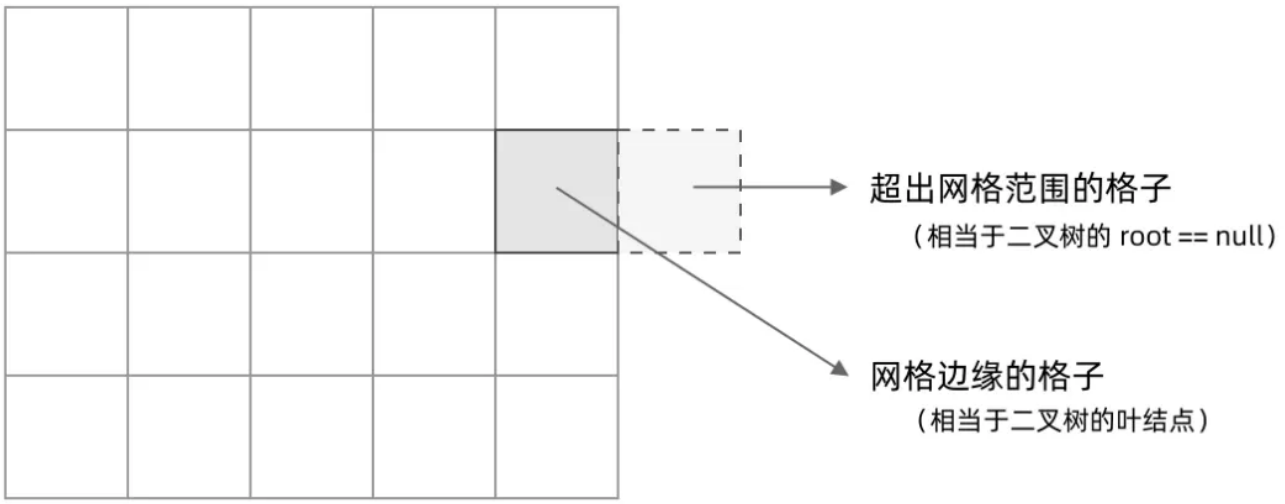
对于网格上的 DFS，我们完全可以参考二叉树的 DFS，写出网格 DFS 的两个要素：

首先，网格结构中的格子有多少相邻结点？答案是上下左右四个。对于格子 `(r, c)` 来说（r 和 c 分别代表行坐标和列坐标），四个相邻的格子分别是 `(r-1, c)`、`(r+1, c)`、`(r, c-1)`、`(r, c+1)`。换句话说，网格结构是「四叉」的。



网格结构中四个相邻的格子

其次，网格 DFS 中的 base case 是什么？从二叉树的 base case 对应过来，应该是网格中不需要继续遍历、`grid[r][c]` 会出现数组下标越界异常的格子，也就是那些超出网格范围的格子。



网格 DFS 的 base case

这一点稍微有些反直觉，坐标竟然可以临时超出网格的范围？这种方法我称为「先污染后治理」—— 甭管当前是在哪个格子，先往四个方向走一步再说，如果发现走出了网格范围再赶紧

返回。这跟二叉树的遍历方法是一样的，先递归调用，发现 `root == null` 再返回。

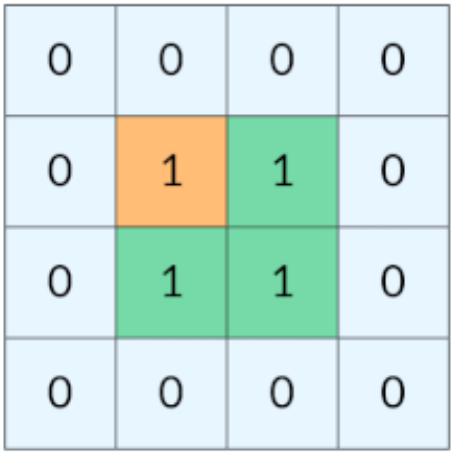
这样，我们得到了网格 DFS 遍历的框架代码：

```
void dfs(int[][] grid, int r, int c) {
    // 判断 base case
    // 如果坐标 (r, c) 超出了网格范围，直接返回
    if (!inArea(grid, r, c)) {
        return;
    }
    // 访问上、下、左、右四个相邻结点
    dfs(grid, r - 1, c);
    dfs(grid, r + 1, c);
    dfs(grid, r, c - 1);
    dfs(grid, r, c + 1);
}

// 判断坐标 (r, c) 是否在网格中
boolean inArea(int[][] grid, int r, int c) {
    return 0 <= r && r < grid.length
         && 0 <= c && c < grid[0].length;
}
```

如何避免重复遍历

网格结构的 DFS 与二叉树的 DFS 最大的不同之处在于，遍历中可能遇到遍历过的结点。这是因为，网格结构本质上是一个「图」，我们可以把每个格子看成图中的结点，每个结点有向上下左右的四条边。在图中遍历时，自然可能遇到重复遍历结点。

这时候，DFS 可能会不停地「兜圈子」，永远停不下来，如下图所示：



DFS 遍历可能会兜圈子（动图）

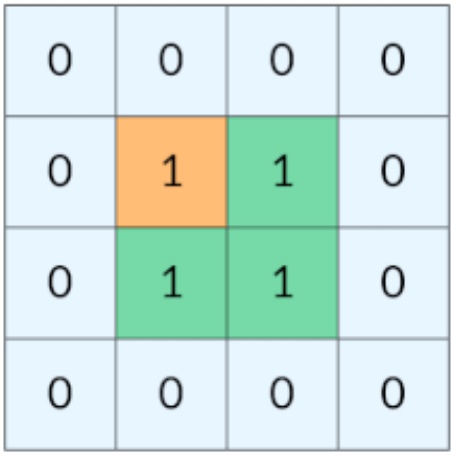
如何避免这样的重复遍历呢？答案是标记已经遍历过的格子。以岛屿问题为例，我们需要在所有值为 1 的陆地格子上做 DFS 遍历。每走过一个陆地格子，就把格子的值改为 2，这样当我们遇到 2 的时候，就知道这是遍历过的格子了。也就是说，每个格子可能取三个值：

- 0 —— 海洋格子
- 1 —— 陆地格子（未遍历过）
- 2 —— 陆地格子（已遍历过）

我们在框架代码中加入避免重复遍历的语句：

```
void dfs(int[][] grid, int r, int c) {
    // 判断 base case
    if (!inArea(grid, r, c)) {
        return;
    }
    // 如果这个格子不是岛屿，直接返回
    if (grid[r][c] != 1) {
        return;
    }
    grid[r][c] = 2; // 将格子标记为「已遍历过」

    // 访问上、下、左、右四个相邻结点
    dfs(grid, r - 1, c);
    dfs(grid, r + 1, c);
    dfs(grid, r, c - 1);
    dfs(grid, r, c + 1);
}

// 判断坐标 (r, c) 是否在网格中
```

```
boolean inArea(int[][] grid, int r, int c) {
    return 0 <= r && r < grid.length
         && 0 <= c && c < grid[0].length;
}
```



标记已遍历的格子

这样，我们就得到了一个岛屿问题、乃至各种网格问题的通用 DFS 遍历方法。以下所讲的几个例题，其实都只需要在 DFS 遍历框架上稍加修改而已。

> 小贴士：
>
> 在一些题解中，可能会把「已遍历过的陆地格子」标记为和海洋格子一样的 0，美其名曰「陆地沉没方法」，即遍历完一个陆地格子就让陆地「沉没」为海洋。这种方法看似很巧妙，但实际上有很大隐患，因为这样我们就无法区分「海洋格子」和「已遍历过的陆地格子」了。如果题目更复杂一点，这很容易出 bug。

岛屿问题的解法

理解了网格结构的 DFS 遍历方法以后，岛屿问题就不难解决了。下面我们分别看看三个题目该如何用 DFS 遍历来求解。

例题 1：岛屿的最大面积

LeetCode 695. Max Area of Island （Medium）

> 给定一个包含了一些 0 和 1 的非空二维数组 `grid` ，一个岛屿是一组相邻的 1（代表陆地），这里的「相邻」要求两个 1 必须在水平或者竖直方向上相邻。你可以假设 `grid` 的四个边缘都被 0（代表海洋）包围着。
>
> 找到给定的二维数组中最大的岛屿面积。如果没有岛屿，则返回面积为 0 。

这道题目只需要对每个岛屿做 DFS 遍历，求出每个岛屿的面积就可以了。求岛屿面积的方法也很简单，代码如下，每遍历到一个格子，就把面积加一。

```
int area(int[][] grid, int r, int c) {
    return 1
        + area(grid, r - 1, c)
        + area(grid, r + 1, c)
        + area(grid, r, c - 1)
        + area(grid, r, c + 1);
}
```

最终我们得到的完整题解代码如下：

```
public int maxAreaOfIsland(int[][] grid) {
    int res = 0;
    for (int r = 0; r < grid.length; r++) {
        for (int c = 0; c < grid[0].length; c++) {
            if (grid[r][c] == 1) {
                int a = area(grid, r, c);
                res = Math.max(res, a);
            }
        }
    }
    return res;
}

int area(int[][] grid, int r, int c) {
    if (!inArea(grid, r, c)) {
        return 0;
    }
    if (grid[r][c] != 1) {
        return 0;
    }
    grid[r][c] = 2;
```

```
        return 1
            + area(grid, r - 1, c)
            + area(grid, r + 1, c)
            + area(grid, r, c - 1)
            + area(grid, r, c + 1);
    }

    boolean inArea(int[][] grid, int r, int c) {
        return 0 <= r && r < grid.length
             && 0 <= c && c < grid[0].length;
    }
```

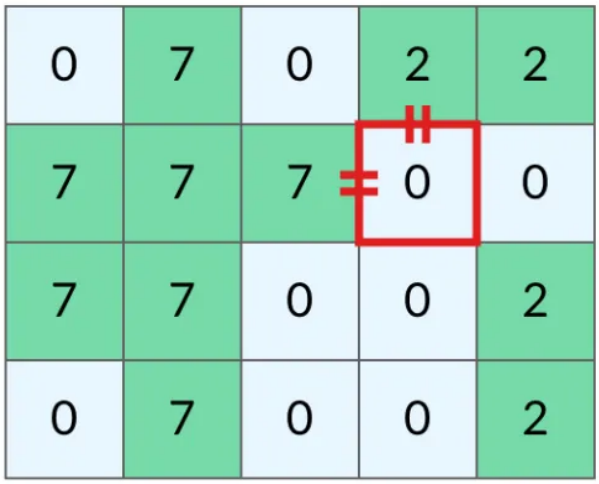
例题 2：填海造陆问题

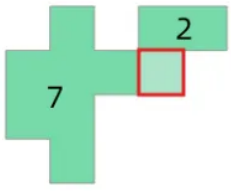
LeetCode 827. Making A Large Island （Hard）

> 在二维地图上， 0 代表海洋，1代表陆地，我们最多只能将一格 0 （海洋）变成 1 （陆地）。进行填海之后，地图上最大的岛屿面积是多少？

这道题是岛屿最大面积问题的升级版。现在我们有填海造陆的能力，可以把一个海洋格子变成陆地格子，进而让两块岛屿连成一块。那么填海造陆之后，最大可能构造出多大的岛屿呢？

大致的思路我们不难想到，我们先计算出所有岛屿的面积，在所有的格子上标记出岛屿的面积。然后搜索哪个海洋格子相邻的两个岛屿面积最大。例如下图中红色方框内的海洋格子，上边、左边都与岛屿相邻，我们可以计算出它变成陆地之后可以连接成的岛屿面积为 $7 + 1 + 2 = 10$。

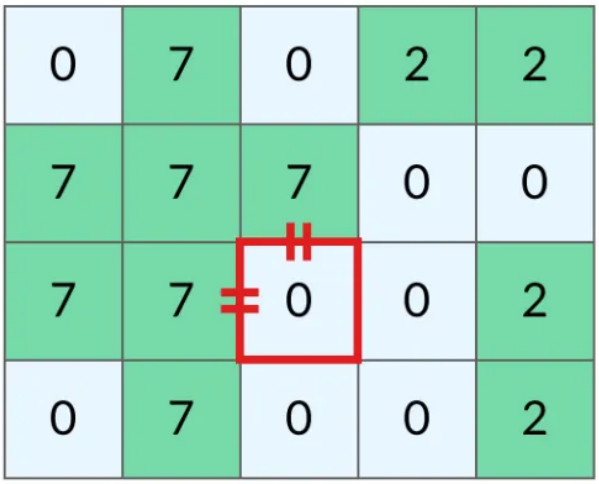


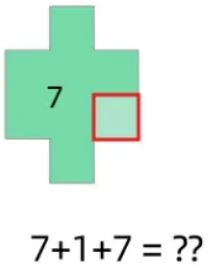


7+1+2 = 10

一个海洋格子连接起两个岛屿

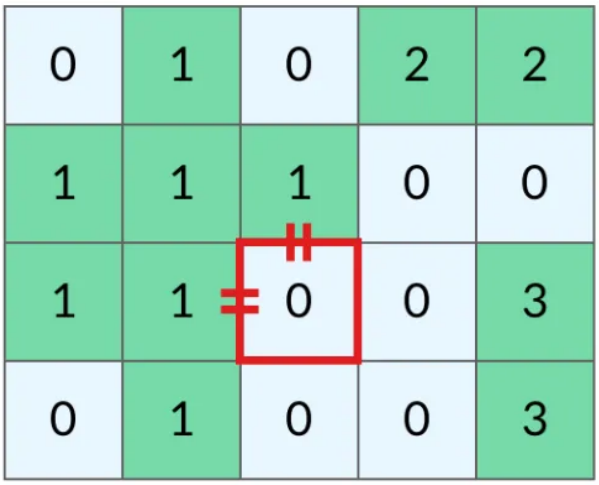
然而，这种做法可能遇到一个问题。如下图中红色方框内的海洋格子，它的上边、左边都与岛屿相邻，这时候连接成的岛屿面积难道是 $7+1+7$？显然不是。这两个 7 来自同一个岛屿，所以填海造陆之后得到的岛屿面积应该只有 $7+1=8$。





一个海洋格子与同一个岛屿有两个边相邻

可以看到，要让算法正确，我们得能区分一个海洋格子相邻的两个 7 是不是来自同一个岛屿。那么，我们不能在方格中标记岛屿的面积，而应该标记岛屿的索引（下标），另外用一个数组记录每个岛屿的面积，如下图所示。这样我们就可以发现红色方框内的海洋格子，它的「两个」相邻的岛屿实际上是同一个。



标记每个岛屿的索引（下标）

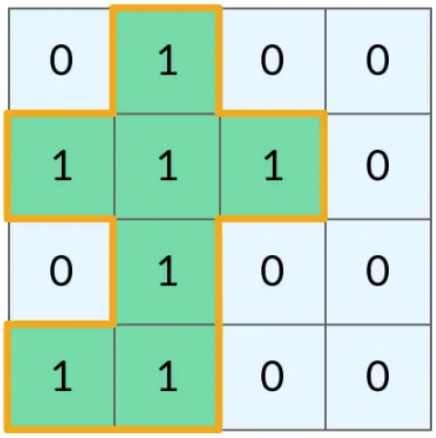
可以看到，这道题实际上是对网格做了两遍 DFS：第一遍 DFS 遍历陆地格子，计算每个岛屿的面积并标记岛屿；第二遍 DFS 遍历海洋格子，观察每个海洋格子相邻的陆地格子。

这道题的基本思路就是这样，具体的代码还有一些需要注意的细节，但和本文的主题已经联系不大。各位可以自己思考一下如何把上述思路转化为代码。

例题 3：岛屿的周长

LeetCode 463. Island Perimeter（Easy）

> 给定一个包含 0 和 1 的二维网格地图，其中 1 表示陆地，0 表示海洋。网格中的格子水平和垂直方向相连（对角线方向不相连）。整个网格被水完全包围，但其中恰好有一个岛屿（一个或多个表示陆地的格子相连组成岛屿）。
>
> 岛屿中没有“湖”（“湖” 指水域在岛屿内部且不和岛屿周围的水相连）。格子是边长为 1 的正方形。计算这个岛屿的周长。



题目示例

实话说，这道题用 DFS 来解并不是最优的方法。对于岛屿，直接用数学的方法求周长会更容易。不过这道题是一个很好的理解 DFS 遍历过程的例题，不信你跟着我往下看。

我们再回顾一下 网格 DFS 遍历的基本框架：

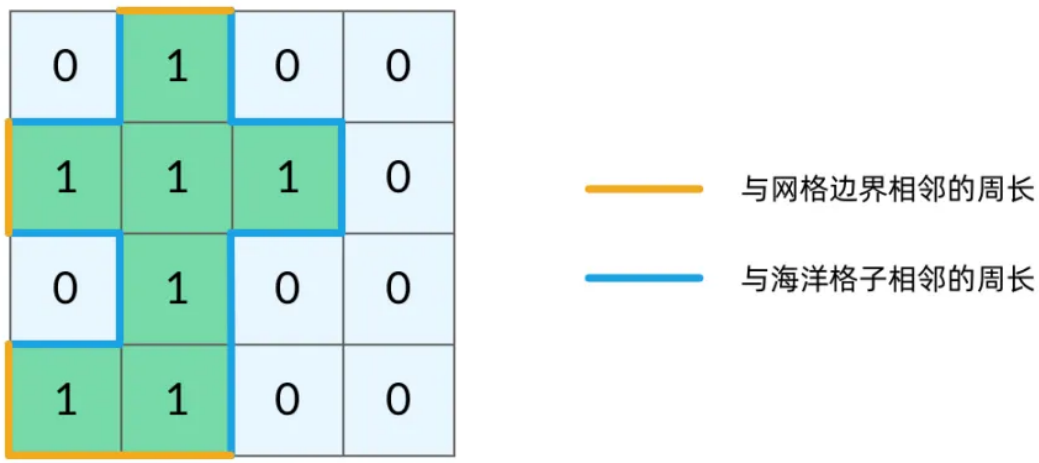
```
void dfs(int[][] grid, int r, int c) {
    // 判断 base case
    if (!inArea(grid, r, c)) {
        return;
    }
    // 如果这个格子不是岛屿，直接返回
    if (grid[r][c] != 1) {
        return;
```

```
        }
        grid[r][c] = 2; // 将格子标记为「已遍历过」

        // 访问上、下、左、右四个相邻结点
        dfs(grid, r - 1, c);
        dfs(grid, r + 1, c);
        dfs(grid, r, c - 1);
        dfs(grid, r, c + 1);
    }

    // 判断坐标 (r, c) 是否在网格中
    boolean inArea(int[][] grid, int r, int c) {
        return 0 <= r && r < grid.length
                && 0 <= c && c < grid[0].length;
    }
```

可以看到， `dfs` 函数直接返回有这几种情况：

- `!inArea(grid, r, c)` ，即坐标 `(r, c)` 超出了网格的范围，也就是我所说的「先污染后治理」的情况
- `grid[r][c] != 1` ，即当前格子不是岛屿格子，这又分为两种情况：
  - `grid[r][c] == 0` ，当前格子是海洋格子
  - `grid[r][c] == 2` ，当前格子是已遍历的陆地格子

那么这些和我们岛屿的周长有什么关系呢？实际上，岛屿的周长是计算岛屿全部的「边缘」，而这些边缘就是我们在 DFS 遍历中， `dfs` 函数返回的位置。观察题目示例，我们可以将岛屿的周长中的边分为两类，如下图所示。黄色的边是与网格边界相邻的周长，而蓝色的边是与海洋格子相邻的周长。



将岛屿周长中的边分为两类

当我们的 `dfs` 函数因为「坐标 `(r, c)` 超出网格范围」返回的时候，实际上就经过了一条黄色的边；而当函数因为「当前格子是海洋格子」返回的时候，实际上就经过了一条蓝色的边。这样，我们就把岛屿的周长跟 DFS 遍历联系起来了，我们的题解代码也呼之欲出：

```
public int islandPerimeter(int[][] grid) {
    for (int r = 0; r < grid.length; r++) {
        for (int c = 0; c < grid[0].length; c++) {
            if (grid[r][c] == 1) {
                // 题目限制只有一个岛屿，计算一个即可
                return dfs(grid, r, c);
            }
        }
    }
    return 0;
}

int dfs(int[][] grid, int r, int c) {
    // 函数因为「坐标 (r, c) 超出网格范围」返回，对应一条黄色的边
    if (!inArea(grid, r, c)) {
        return 1;
    }
    // 函数因为「当前格子是海洋格子」返回，对应一条蓝色的边
    if (grid[r][c] == 0) {
        return 1;
    }
    // 函数因为「当前格子是已遍历的陆地格子」返回，和周长没关系
    if (grid[r][c] != 1) {
        return 0;
    }
    grid[r][c] = 2;
    return dfs(grid, r - 1, c)
        + dfs(grid, r + 1, c)
        + dfs(grid, r, c - 1)
        + dfs(grid, r, c + 1);
}

// 判断坐标 (r, c) 是否在网格中
boolean inArea(int[][] grid, int r, int c) {
    return 0 <= r && r < grid.length
         && 0 <= c && c < grid[0].length;
}
```

总结

对比完三个例题的题解代码，你会发现网格问题的代码真的都非常相似。其实这一类问题属于「会了不难」类型。了解树、图的基本遍历方法，再学会一点小技巧，掌握网格 DFS 遍历就一点也不难了。

岛屿类问题是网格类问题中的典型代表，做了几道岛屿类问题，再看其他的问题其实本质上和岛屿问题是一样的，例如 **LeetCode 130. Surrounded Regions** 这道题，将岛屿的 1 和 0 转换为字母 O 和 X，但本质上是完全一样的。

不过，一些网格类问题用 DFS 做不出来，需要用到 BFS 遍历。下一篇文章中，会介绍 BFS 的应用场景以及代码技巧，敬请期待！

往期文章

- **LeetCode 例题精讲 | 11 二叉树转化为链表：二叉树遍历中的相邻结点**
- **LeetCode 例题精讲 | 10 二叉树直径：二叉树遍历中的全局变量**
- **LeetCode 例题精讲 | 04 用双指针解 Two Sum：缩减搜索空间**