BFS 的使用场景总结：层序遍历、最短路径问题

leetcode-cn.com/problems/binary-tree-level-order-traversal/solution/bfs-de-shi-yong-chang-jing-zong-jie-ceng-xu-bian-l

解题思路

本文将会讲解为什么这道题适合用广度优先搜索（BFS），以及 BFS 适用于什么样的场景。

DFS（深度优先搜索）和 BFS（广度优先搜索）就像孪生兄弟，提到一个总是想起另一个。然而在实际使用中，我们用 DFS 的时候远远多于 BFS。那么，是不是 BFS 就没有什么用呢？

如果我们使用 DFS/BFS 只是为了遍历一棵树、一张图上的所有结点的话，那么 DFS 和 BFS 的能力没什么差别，我们当然更倾向于更方便写、空间复杂度更低的 DFS 遍历。不过，某些使用场景是 DFS 做不到的，只能使用 BFS 遍历。这就是本文要介绍的两个场景：「层序遍历」、「最短路径」。

本文包括以下内容：

- DFS 与 BFS 的特点比较
- BFS 的适用场景
- 如何用 BFS 进行层序遍历
- 如何用 BFS 求解最短路径问题

DFS 与 BFS

让我们先看看在二叉树上进行 DFS 遍历和 BFS 遍历的代码比较。

DFS 遍历使用**递归**：

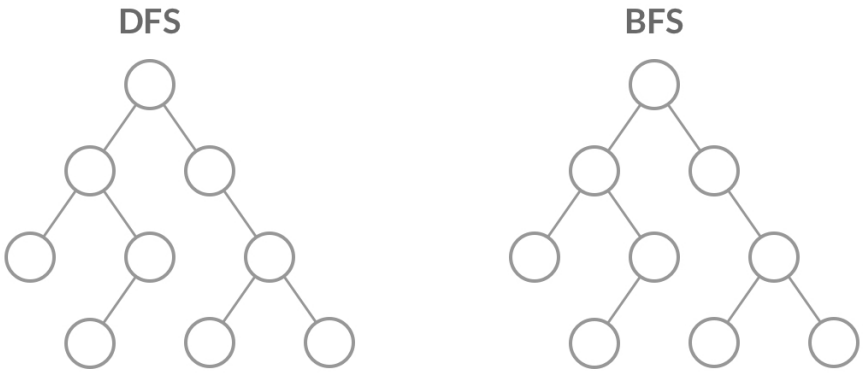
```
void dfs(TreeNode root) {
    if (root == null) {
        return;
    }
    dfs(root.left);
    dfs(root.right);
}
```

BFS 遍历使用**队列**数据结构：

```
void bfs(TreeNode root) {
    Queue<TreeNode> queue = new ArrayDeque<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        TreeNode node = queue.poll();
        if (node.left != null) {
            queue.add(node.left);
        }
        if (node.right != null) {
            queue.add(node.right);
        }
    }
}
```

只是比较两段代码的话，最直观的感受就是：DFS 遍历的代码比 BFS 简洁太多了！这是因为递归的方式隐含地使用了系统的 **栈**，我们不需要自己维护一个数据结构。如果只是简单地将二叉树遍历一遍，那么 DFS 显然是更方便的选择。

虽然 DFS 与 BFS 都是将二叉树的所有结点遍历了一遍，但它们遍历结点的顺序不同。



这个遍历顺序也是 BFS 能够用来解「层序遍历」、「最短路径」问题的根本原因。下面，我们结合几道例题来讲讲 BFS 是如何求解层序遍历和最短路径问题的。

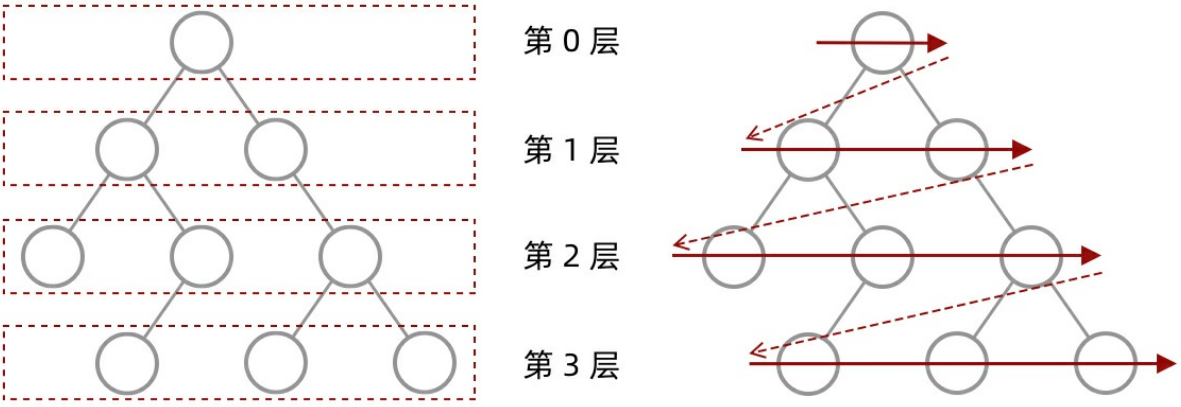
BFS 的应用一：层序遍历

BFS 的层序遍历应用就是本题了：

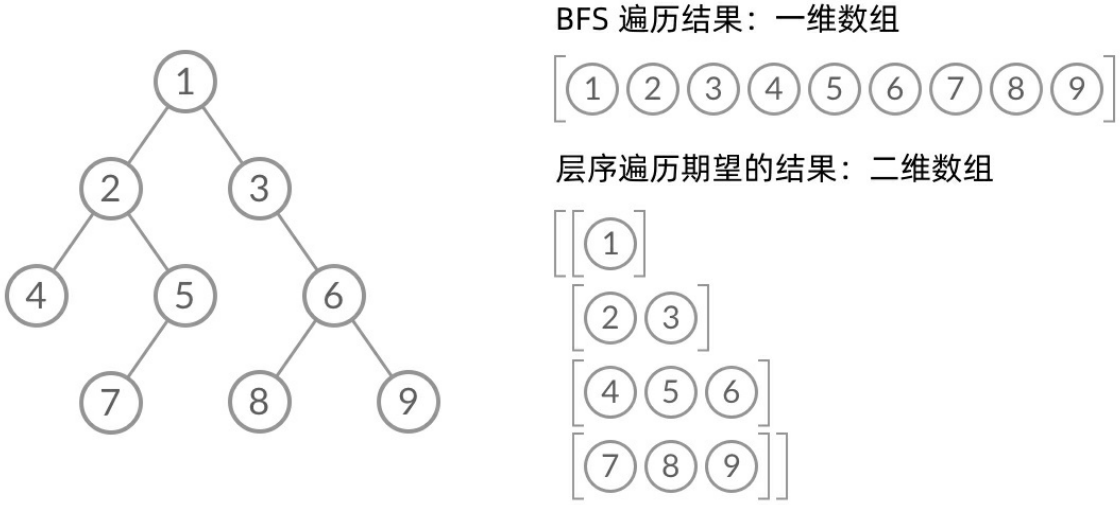
LeetCode 102. Binary Tree Level Order Traversal 二叉树的层序遍历（Medium）

> 给定一个二叉树，返回其按层序遍历得到的节点值。 层序遍历即逐层地、从左到右访问所有结点。

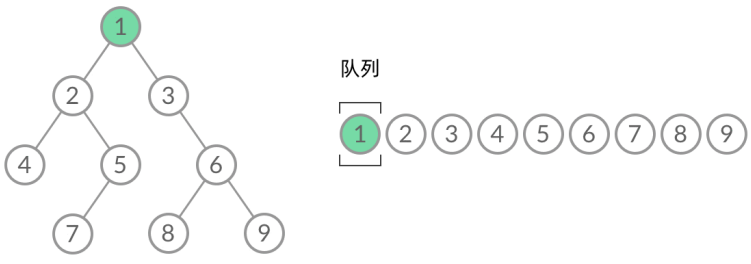
什么是层序遍历呢？简单来说，层序遍历就是把二叉树分层，然后每一层从左到右遍历：



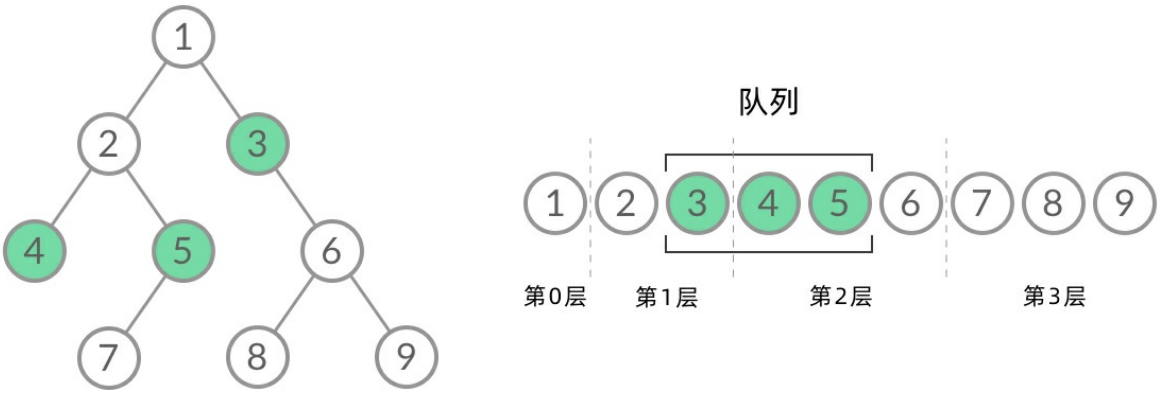
乍一看来，这个遍历顺序和 BFS 是一样的，我们可以直接用 BFS 得出层序遍历结果。然而，层序遍历要求的输入结果和 BFS 是不同的。层序遍历要求我们区分每一层，也就是返回一个二维数组。而 BFS 的遍历结果是一个一维数组，无法区分每一层。



那么，怎么给 BFS 遍历的结果分层呢？我们首先来观察一下 BFS 遍历的过程中，结点进队列和出队列的过程：



截取 BFS 遍历过程中的某个时刻：

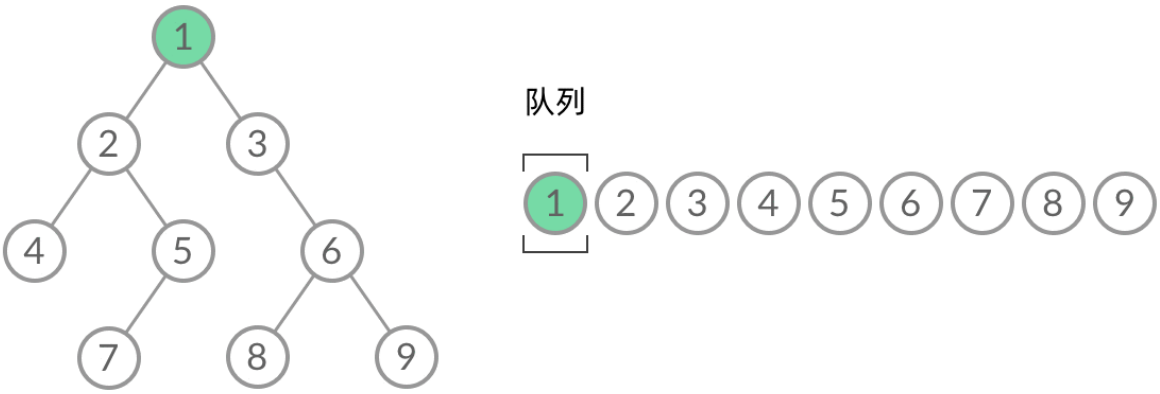


可以看到，此时队列中的结点是 3、4、5，分别来自第 1 层和第 2 层。这个时候，第 1 层的结点还没出完，第 2 层的结点就进来了，而且两层的结点在队列中紧挨在一起，我们**无法区分队列中的结点来自哪一层**。

因此，我们需要稍微修改一下代码，在每一层遍历开始前，先记录队列中的结点数量 nnn（也就是这一层的结点数量），然后一口气处理完这一层的 nnn 个结点。

```
void bfs(TreeNode root) {
    Queue<TreeNode> queue = new ArrayDeque<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        int n = queue.size();
        for (int i = 0; i < n; i++) {

            TreeNode node = queue.poll();
            if (node.left != null) {
                queue.add(node.left);
            }
            if (node.right != null) {
                queue.add(node.right);
            }
        }
    }
}
```

这样，我们就将 BFS 遍历改造成了层序遍历。在遍历的过程中，结点进队列和出队列的过程为：



可以看到，在 while 循环的每一轮中，都是将当前层的所有结点出队列，再将下一层的所有结点入队列，这样就实现了层序遍历。

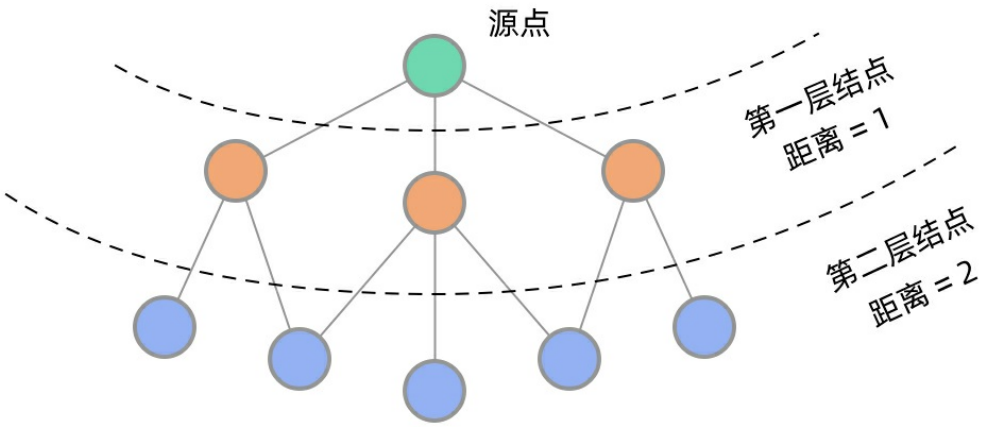
最终我们得到的题解代码为：

```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<>();

    Queue<TreeNode> queue = new ArrayDeque<>();
    if (root != null) {
        queue.add(root);
    }
    while (!queue.isEmpty()) {
        int n = queue.size();
        List<Integer> level = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            TreeNode node = queue.poll();
            level.add(node.val);
            if (node.left != null) {
                queue.add(node.left);
            }
            if (node.right != null) {
                queue.add(node.right);
            }
        }
        res.add(level);
    }

    return res;
}
```

BFS 的应用二：最短路径

在一棵树中，一个结点到另一个结点的路径是唯一的，但在图中，结点之间可能有多条路径，其中哪条路最近呢？这一类问题称为**最短路径问题**。最短路径问题也是 BFS 的典型应用，而且其方法与层序遍历关系密切。

在二叉树中，BFS 可以实现一层一层的遍历。在图中同样如此。从源点出发，BFS 首先遍历到第一层结点，到源点的距离为 1，然后遍历到第二层结点，到源点的距离为 2.....可以看到，用 BFS 的话，距离源点更近的点会先被遍历到，这样就能找到到某个点的最短路径了。



> **小贴士：**
>
> 很多同学一看到「最短路径」，就条件反射地想到「Dijkstra 算法」。为什么 BFS 遍历也能找到最短路径呢？
>
> 这是因为，Dijkstra 算法解决的是**带权最短路径问题**，而我们这里关注的是**无权最短路径问题**。也可以看成每条边的权重都是 1。这样的最短路径问题，用 BFS 求解就行了。
>
> 在面试中，你可能更希望写 BFS 而不是 Dijkstra。毕竟，敢保证自己能写对 Dijkstra 算法的人不多。

最短路径问题属于图算法。由于图的表示和描述比较复杂，本文用比较简单的网格结构代替。网格结构是一种特殊的图，它的表示和遍历都比较简单，适合作为练习题。在 LeetCode 中，最短路径问题也以网格结构为主。

最短路径例题讲解

LeetCode 1162. As Far from Land as Possible 离开陆地的最远距离（Medium）

> 你现在手里有一份大小为 n×nn \times nn×n 的地图网格 `grid` ，上面的每个单元格都标记为 0 或者 1，其中 0 代表海洋，1 代表陆地，请你找出一个海洋区域，这个海洋区域到离它最近的陆地区域的距离是最大的。
>
> 我们这里说的距离是「曼哈顿距离」。(x0,y0)(x_0, y_0)(x0,y0) 和 (x1,y1)(x_1, y_1)(x1,y1) 这两个区域之间的距离是 ∣x0−x1∣+∣y0−y1∣|x_0 - x_1| + |y_0 - y_1|∣x0−x1∣+∣y0−y1∣ 。
>
> 如果我们的地图上只有陆地或者海洋，请返回 -1。

这道题就是一个在网格结构中求最短路径的问题。同时，它也是一个「岛屿问题」，即用网格中的 1 和 0 表示陆地和海洋，模拟出若干个岛屿。

在上一篇文章中，我们介绍了网格结构的基本概念，以及网格结构中的 DFS 遍历。其中一些概念和技巧也可以用在 BFS 遍历中：

- 格子 `(r, c)` 的相邻四个格子为：`(r-1, c)`、`(r+1, c)`、`(r, c-1)` 和 `(r, c+1)`；
- 使用函数 `inArea` 判断当前格子的坐标是否在网格范围内；
- 将遍历过的格子标记为 2，避免重复遍历。

对于网格结构的性质、网格结构的 DFS 遍历技巧不是很了解的同学，可以复习一下上一篇文章：LeetCode 例题精讲 | 12 岛屿问题：网格结构中的 DFS。

上一篇文章讲过了网格结构 DFS 遍历，这篇文章正好讲解一下网格结构的 BFS 遍历。要解最短路径问题，我们首先要写出层序遍历的代码，仿照上面的二叉树层序遍历代码，类似地可以写出网格层序遍历：

```
void bfs(int[][] grid, int i, int j) {
    Queue<int[]> queue = new ArrayDeque<>();
    queue.add(new int[]{r, c});
    while (!queue.isEmpty()) {
        int n = queue.size();
        for (int i = 0; i < n; i++) {
            int[] node = queue.poll();
            int r = node[0];
            int c = node[1];
            if (r-1 >= 0 && grid[r-1][c] == 0) {
                grid[r-1][c] = 2;
                queue.add(new int[]{r-1, c});
            }
            if (r+1 < N && grid[r+1][c] == 0) {
                grid[r+1][c] = 2;
                queue.add(new int[]{r+1, c});
            }
            if (c-1 >= 0 && grid[r][c-1] == 0) {
                grid[r][c-1] = 2;
                queue.add(new int[]{r, c-1});
            }
            if (c+1 < N && grid[r][c+1] == 0) {
                grid[r][c+1] = 2;
                queue.add(new int[]{r, c+1});
            }
        }
    }
}
```

以上的层序遍历代码有几个注意点：

- 队列中的元素类型是 `int[]` 数组，每个数组的长度为 2，包含格子的行坐标和列坐标。

- 为了避免重复遍历，这里使用到了和 DFS 遍历一样的技巧：把已遍历的格子标记为 2。注意：我们在将格子放入队列之前就将其标记为 2。想一想，这是为什么？
- 在将格子放入队列之前就检查其坐标是否在网格范围内，避免将「不存在」的格子放入队列。

这段网格遍历代码还有一些可以优化的地方。由于一个格子有四个相邻的格子，代码中判断了四遍格子坐标的合法性，代码稍微有点啰嗦。我们可以用一个 `moves` 数组存储相邻格子的四个方向：

```
int[][] moves = {
    {-1, 0}, {1, 0}, {0, -1}, {0, 1},
};
```

然后把四个 if 判断变成一个循环：

```
for (int[][] move : moves) {
    int r2 = r + move[0];
    int c2 = c + move[1];
    if (inArea(grid, r2, c2) && grid[r2][c2] == 0) {
        grid[r2][c2] = 2;
        queue.add(new int[]{r2, c2});
    }
}
```

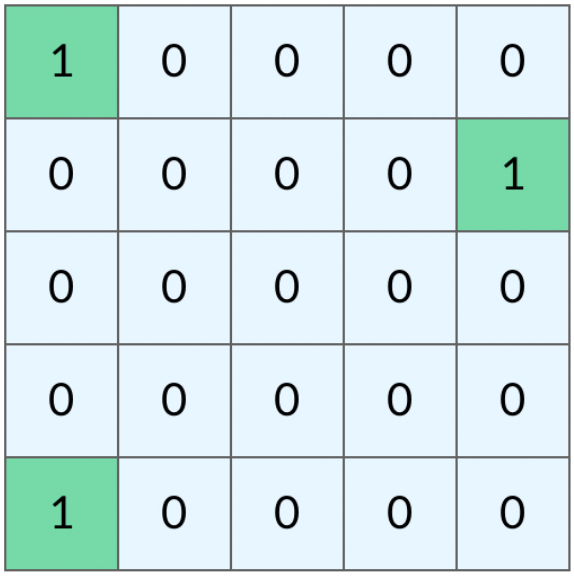
写好了层序遍历的代码，接下来我们看看如何来解决本题中的最短路径问题。

这道题要找的是距离陆地最远的海洋格子。假设网格中只有一个陆地格子，我们可以从这个陆地格子出发做层序遍历，直到所有格子都遍历完。最终遍历了几层，海洋格子的最远距离就是几。

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

那么有多个陆地格子的时候怎么办呢？一种方法是将每个陆地格子都作为起点做一次层序遍历，但是这样的时间开销太大。

BFS 完全可以以多个格子同时作为起点。我们可以把所有的陆地格子同时放入初始队列，然后开始层序遍历，这样遍历的效果如下图所示：



这种遍历方法实际上叫做「**多源 BFS**」。多源 BFS 的定义不是今天讨论的重点，你只需要记住多源 BFS 很方便，只需要把多个源点同时放入初始队列即可。

需要注意的是，虽然上面的图示用 1、2、3、4 表示层序遍历的层数，但是在代码中，我们不需要给每个遍历到的格子标记层数，只需要用一个 `distance` 变量记录当前的遍历的层数（也就是到陆地格子的距离）即可。

最终，我们得到的题解代码为：

```
public int maxDistance(int[][] grid) {
    int N = grid.length;

    Queue<int[]> queue = new ArrayDeque<>();

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (grid[i][j] == 1) {
                queue.add(new int[]{i, j});
            }
        }
    }


    if (queue.isEmpty() || queue.size() == N * N) {
        return -1;
    }

    int[][] moves = {
        {-1, 0}, {1, 0}, {0, -1}, {0, 1},
    };

    int distance = -1;
    while (!queue.isEmpty()) {
        distance++;
        int n = queue.size();
        for (int i = 0; i < n; i++) {
            int[] node = queue.poll();
            int r = node[0];
            int c = node[1];
            for (int[] move : moves) {
                int r2 = r + move[0];
                int c2 = c + move[1];
                if (inArea(grid, r2, c2) && grid[r2][c2] == 0) {
                    grid[r2][c2] = 2;
                    queue.add(new int[]{r2, c2});
                }
            }
        }
    }

    return distance;
}


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

总结

可以看到，「BFS 遍历」、「层序遍历」、「最短路径」实际上是递进的关系。在 BFS 遍历的基础上区分遍历的每一层，就得到了层序遍历。在层序遍历的基础上记录层数，就得到了最短路径。

BFS 遍历是一类很值得反复体会和练习的题目。一方面，BFS 遍历是一个经典的基础算法，需要重点掌握。另一方面，我们需要能根据题意分析出题目是要求最短路径，知道是要做 BFS 遍历。

本文讲解的只是两道非常典型的例题。LeetCode 中还有许多层序遍历和最短路径的题目

层序遍历的一些变种题目：

对于最短路径问题，还有两道题目也是求网格结构中的最短路径，和我们讲解的距离岛屿的最远距离非常类似：

还有一道在真正的图结构中求最短路径的问题：

> LeetCode 310. Minimum Height Trees

经过了本文的讲解，相信解决这些题目也不是难事。

本文作者 nettee，致力于写各种 LeetCode 解题套路，让你在解题之外举一反三。欢迎阅读原文（排版更好），或者在力扣关注我，我会不定期更新题解文章。