# 二叉树遍历 one pager

MLE 算法指北

2025 年 2 月 8 日

## 1 引言

二叉树是计算机科学中常见的数据结构，其遍历方式包括前序遍历（Preorder Traversal）、中序遍历（Inorder Traversal）、后序遍历（Postorder Traversal）和层序遍历（Level Order Traversal）。不同遍历方式适用于不同问题，合理选择合适的遍历方式可以大大提高算法的效率和可读性。

## 2 二叉树遍历框架

### 2.1 递归框架

二叉树的递归遍历可以通过如下的统一框架实现：

```
def traverse(root):
    if not root:
        return
    # 1. 前序遍历位置（Preorder）
    traverse(root.left)
    # 2. 中序遍历位置（Inorder）
    traverse(root.right)
    # 3. 后序遍历位置（Postorder）
```

### 2.2 迭代框架

递归方式虽然直观，但在数据规模较大时可能导致栈溢出，因此可以使用迭代方式来遍历二叉树。例如：

**前序遍历（Preorder）**

```
def preorder_traversal(root):
    if not root:
        return []
    stack, result = [root], []
    while stack:
        node = stack.pop()
        result.append(node.val)
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)
    return result
```

**层序遍历（Level Order）**

```
from collections import deque

def level_order_traversal(root):
    if not root:
        return []
    queue, result = deque([root]), []
    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
```

```
                queue.append(node.right)
            result.append(level)
        return result
```

# 3 遍历方法论与经典题目

## 3.1 前序遍历（Preorder）

### 3.1.1 适用场景

- 需要先处理当前节点，再处理子节点，例如：
  - **树的构造问题**：例如根据前序遍历和中序遍历重建二叉树。
  - **树的序列化与反序列化**：先访问根节点可以用于将树编码成字符串。

### 3.1.2 经典题目

- **Leetcode 105**：从前序遍历和中序遍历构造二叉树
- **Leetcode 297**：二叉树的序列化与反序列化
- **Leetcode 144**：二叉树的前序遍历
- **Leetcode 589**：N 叉树的前序遍历
- **Leetcode 1008**：前序遍历构造二叉搜索树

### 3.1.3 题目解析：Leetcode 105 - 从前序遍历和中序遍历构造二叉树

**题目描述**：给定二叉树的前序遍历和中序遍历结果，重建该二叉树。
**解法**：

- 前序遍历的第一个元素是 ** 根节点 **。
- 在中序遍历中，根节点左侧是 ** 左子树 **，右侧是 ** 右子树 **。
- 递归构造左右子树。

```
class Solution:
    def buildTree(self, preorder, inorder):
        if not preorder or not inorder:
            return None
        root = TreeNode(preorder[0])
        index = inorder.index(preorder[0])
        root.left = self.buildTree(preorder[1:index+1], inorder[:index])
        root.right = self.buildTree(preorder[index+1:], inorder[index+1:])
        return root
```

## 3.2 中序遍历（Inorder）

### 3.2.1 适用场景

- 适用于 ** 二叉搜索树（BST）** 相关问题：
  - **BST 的顺序处理**：中序遍历 BST 时，节点值是递增顺序。
  - **Kth Smallest 元素查找**：可以使用中序遍历找到第 k 小的元素。

### 3.2.2 经典题目

- **Leetcode 94**：二叉树的中序遍历
- **Leetcode 230**：二叉搜索树中第 k 小的元素
- **Leetcode 99**：恢复二叉搜索树
- **Leetcode 173**：二叉搜索树迭代器
- **Leetcode 897**：递增顺序搜索树

#### 3.2.3 题目解析：Leetcode 230 - BST 中第 k 小的元素

**题目描述**：在二叉搜索树（BST）中找到第 $k$ 小的元素。
**解法**：

- 中序遍历 BST 得到递增序列。
- 依次遍历，当遍历到第 $k$ 个节点时返回该值。

```
class Solution:
    def kthSmallest(self, root, k):
        stack, count = [], 0
        while stack or root:
            while root:
                stack.append(root)
                root = root.left
            root = stack.pop()
            count += 1
            if count == k:
                return root.val
            root = root.right
```

### 3.3 后序遍历（Postorder）

#### 3.3.1 适用场景

- 适用于 **从子树向上计算**的问题：
  - **计算树的某些属性**（高度、路径和）。
  - **递归删除节点**（后序遍历保证子节点被处理后再删除父节点）。

#### 3.3.2 经典题目

- **Leetcode 145**：二叉树的后序遍历
- **Leetcode 124**：二叉树的最大路径和
- **Leetcode 543**：二叉树的直径
- **Leetcode 337**：打家劫舍 III（树形 DP）
- **Leetcode 236**：二叉树的最近公共祖先

#### 3.3.3 题目解析：Leetcode 124 - 二叉树的最大路径和

**题目描述**：求二叉树中 ** 任意两节点 ** 之间的最大路径和。
**解法**：

- 采用 ** 后序遍历 **，计算 ** 包含当前节点的最大路径和 **。
- 递归返回 ** 以当前节点为起点的最大贡献值 **。

```
class Solution:
    def maxPathSum(self, root):
        self.max_sum = float('-inf')

        def dfs(node):
            if not node:
                return 0
            left = max(dfs(node.left), 0)
            right = max(dfs(node.right), 0)
            self.max_sum = max(self.max_sum, left + right + node.val)
            return max(left, right) + node.val

        dfs(root)
        return self.max_sum
```

## 3.4 层序遍历（Level Order）

### 3.4.1 适用场景

- 适用于 ** 逐层处理 ** 的问题：
  - **寻找最短路径**：如求最小深度。
  - **按层级返回结果**：如返回每一层的节点列表。
  - **计算每一层的某些属性**：如每层的最大值。

### 3.4.2 经典题目

- **Leetcode 102**：二叉树的层序遍历
- **Leetcode 107**：二叉树的层序遍历 II（从底部向上）
- **Leetcode 199**：二叉树的右视图
- **Leetcode 637**：二叉树每层的平均值
- **Leetcode 111**：二叉树的最小深度

### 3.4.3 题目解析：Leetcode 199 - 二叉树的右视图

**题目描述**：返回二叉树从 ** 右侧看见的所有节点 **。
**解法**：

- 采用 ** 层序遍历 **，记录每一层最右侧的节点。

```
from collections import deque

class Solution:
    def rightSideView(self, root):
        if not root:
            return []
        queue, result = deque([root]), []
        while queue:
            right_most = None
            for _ in range(len(queue)):
                node = queue.popleft()
                right_most = node
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            result.append(right_most.val)
        return result
```

## 3.5 通用方法论总结

- **前序遍历（Preorder）**适用于 **需要在递归时保留父节点信息**的问题，如构造二叉树、序列化/反序列化树。
- **中序遍历（Inorder）**适用于 **二叉搜索树（BST）相关问题**，例如查找第 k 小的元素、恢复 BST 等。
- **后序遍历（Postorder）**适用于 **从子树向上累积信息**的问题，如计算树的最大路径和、直径等。
- **层序遍历（Level Order）**适用于 **逐层处理或最短路径计算**的问题，如求最小深度、二叉树右视图等。

**复杂度分析**：

- **时间复杂度**：$O(N)$，每个节点访问一次。
- **空间复杂度**：$O(H)$，递归栈的深度（最坏情况下 $O(N)$，最好情况 $O(\log N)$）。