# 1 二叉树

## 1.1 二叉树的框架思维

所有回溯、动态规划、分治算法，其实都是树的问题。而树的问题就永远逃不开树的递归遍历框架这几行破代码：

```Java

/\* 二叉树遍历框架 \*/

void traverse(TreeNode root) {

    // 前序遍历

    traverse(root.left)

    // 中序遍历

    traverse(root.right)

    // 后序遍历

}

```

【递归解法应该是最简单，最容易理解的才对，行云流水地写递归代码是学好算法的基本功，而二叉树相关的题目就是最练习递归基本功，最练习框架思维的。】

【快速排序就是个二叉树的前序遍历，归并排序就是个二叉树的后序遍历】

快速排序的逻辑是，若要对 nums[lo..hi] 进行排序，我们先找一个分界点 p，通过交换元素使得 nums[lo..p-1] 都小于等于 nums[p]，且 nums[p+1..hi] 都大于 nums[p]，然后递归地去 nums[lo..p-1] 和 nums[p+1..hi] 中寻找新的分界点，最后整个数组就被排序了。

快速排序的代码框架如下：

```java

void sort(int[] nums, int lo, int hi) {

    /\*\*\*\*\*\* 前序遍历位置 \*\*\*\*\*\*/

    // 通过交换元素构建分界点 p

    int p = partition(nums, lo, hi);

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    sort(nums, lo, p - 1);

    sort(nums, p + 1, hi);

}

```

归并排序的逻辑，若要对 nums[lo..hi] 进行排序，我们先对 nums[lo..mid] 排序，再对 nums[mid+1..hi] 排序，最后把这两个有序的子数组合并，整个数组就排好序了。

归并排序的代码框架如下：

```java

void sort(int[] nums, int lo, int hi) {

    int mid = (lo + hi) / 2;

    sort(nums, lo, mid);

    sort(nums, mid + 1, hi);

    /\*\*\*\*\*\* 后序遍历位置 \*\*\*\*\*\*/

    // 合并两个排好序的子数组

    merge(nums, lo, mid, hi);

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

```

归并排序也有分治算法的思想。

【甚至可以说，只要涉及递归，都可以抽象成二叉树的问题。】

## 1.2 写递归算法的秘诀

写递归算法的关键是要明确函数的“定义”是什么，然后相信这个“定义”，利用这个“定义”推导最终结果，绝不要跳入递归的细节。

比如说让你计算一棵二叉树共有几个节点：

```java

// 定义：count(root) 返回以 root 为根的树有多少节点

int count(TreeNode root) {

    // base case

    if (root == null) return 0;

    // 自己加上子树的节点数就是整棵树的节点数

    return 1 + count(root.left) + count(root.right);

}

```

写树相关的算法，简单说就是，先搞清楚当前 root 节点“该做什么”以及“什么时候做”，然后根据函数定义递归调用子节点，递归调用会让孩子节点做相同的事情。

所谓“该做什么”就是让你想清楚写什么代码能够实现题目想要的效果，所谓“什么时候做”，就是让你思考这段代码到底应该写在前序、中序还是后序遍历的代码位置上。

写二叉树的算法题，都是基于递归框架的，我们先要搞清楚 root 节点它自己要做什么，然后根据题目要求选择使用前序，中序，后续的递归框架。

二叉树题目的难点在于如何通过题目的要求思考出每一个节点需要做什么，这个只能通过多刷题进行练习了。

【二叉树的问题难点在于，如何把题目的要求细化成每个节点需要做的事情】

## 1.3 算法实践

### leet-226 翻转二叉树

输入一个二叉树根节点 root，让你把整棵树镜像翻转，比如输入的二叉树如下：

```

     4

   /   \

  2     7

 / \   / \

1   3 6   9

算法原地翻转二叉树，使得以 root 为根的树变成：

     4

   /   \

  7     2

 / \   / \

9   6 3   1

```

只要把二叉树上的每一个节点的左右子节点进行交换，最后的结果就是完全翻转之后的二叉树。

```java

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    TreeNode invertTree(TreeNode root) {

        // base case

        if (root == null) {

            return null;

        }

        /\*\*\*\* 前序遍历位置 \*\*\*\*/

        // root 节点需要交换它的左右子节点

        TreeNode tmp = root.left;

        root.left = root.right;

        root.right = tmp;

        // 让左右子节点继续翻转它们的子节点

        invertTree(root.left);

        invertTree(root.right);

        return root;

    }

}

```

### leet-116 填充每个节点的下一个右侧节点指针

填充每个节点的下一个右侧节点指针

```java

/\*

// Definition for a Node.

class Node {

    public int val;

    public Node left;

    public Node right;

    public Node next;

    public Node() {}

    public Node(int \_val) {

        val = \_val;

    }

    public Node(int \_val, Node \_left, Node \_right, Node \_next) {

        val = \_val;

        left = \_left;

        right = \_right;

        next = \_next;

    }

};

\*/

class Solution {

    // 主函数

    Node connect(Node root) {

        if (root == null) return null;

        connectTwoNode(root.left, root.right);

        return root;

    }

    // 辅助函数

    void connectTwoNode(Node node1, Node node2) {

        if (node1 == null || node2 == null) {

            return;

        }

        /\*\*\*\* 前序遍历位置 \*\*\*\*/

        // 将传入的两个节点连接

        node1.next = node2;

        // 连接相同父节点的两个子节点

        connectTwoNode(node1.left, node1.right);

        connectTwoNode(node2.left, node2.right);

        // 连接跨越父节点的两个子节点

        connectTwoNode(node1.right, node2.left);

    }

}

```

### leet-114 二叉树展开为链表

```java

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    // 定义：将以 root 为根的树拉平为链表

    public void flatten(TreeNode root) {

        // base case

        if (root == null) return;

        flatten(root.left);

        flatten(root.right);

        /\*\*\*\* 后序遍历位置 \*\*\*\*/

        // 1、左右子树已经被拉平成一条链表

        TreeNode left = root.left;

        TreeNode right = root.right;

        // 2、将左子树作为右子树

        root.left = null;

        root.right = left;

        // 3、将原先的右子树接到当前右子树的末端

        TreeNode p = root;

        while (p.right != null) {

            p = p.right;

        }

        p.right = right;

    }

}

```