用递归实现遍历法和分治法

主讲人 令狐冲

递归、深搜和回溯法的区别

Recursion / DFS / Backtracking

递归 Recursion

- 递归函数：程序的一种实现方式，即函数进行了自我调用
- 递归算法：即大问题的结果依赖于小问题的结果，于是先用递归函数求解小问题
- 一般我们说递归的时候，大部分时候都在说递归函数而不是递归算法

深度优先搜索
Depth First Search

- 可以使用递归函数来实现
- 也可以不用递归函数来实现，如自己通过一个手动创建的栈 Stack 进行操作
- 深度优先搜索通常是指在搜索的过程中优先搜索深度更深的点而不是按照宽度搜索同层节点

回溯 Backtracking

- 回溯法：就是深度优先搜索算法
- 回溯操作：递归函数在回到上一层递归调用处的时候，一些参数需要改回到调用前的值，这个操作就是回溯，即让状态参数回到之前的值，递归调用前做了什么改动，递归调用之后都改回来

找点 vs 找路径

https://www.lintcode.com/problem/binary-tree-paths

是否需要手动“回溯”的判断标准

Python 代码


```python
def findNodes(self, node, nodes):
    if not node:
        return

    nodes.append(node.val)
    self.findNodes(node.left, nodes)
    self.findNodes(node.right, nodes)

def findPaths(self, node, path, paths):
    if not node:
        return
    if not node.left and not node.right:
        paths.append('->'.join([str(n.val) for n in path]))
        return

    path.append(node.left)
    self.findPaths(node.left, path, paths)
    path.pop()  # this is backtracking for variable path

    path.append(node.right)
    self.findPaths(node.right, path, paths)
    path.pop()  # this is backtracking for variable path
```

在 findNodes 中是否有回溯操作的存在呢?

Java 代码


```java
public void findNodes(TreeNode node, List<TreeNode> nodes) {
    if (node == null) {
        return;
    }

    nodes.add(node);
    findNodes(node.left, nodes);
    findNodes(node.right, nodes);
}

public void findPaths(TreeNode node, String path, List<String> paths) {
    if (node == null) {
        return;
    }
    if (node.left == null && node.right == null) {
        paths.add(path);
        return;
    }

    if (node.left != null) {
        findPaths(node.left, path + "->" + node.left.val, paths);
    }
    if (node.right != null) {
        findPaths(node.right, path + "->" + node.right.val, paths);
    }
}
```

使用 path + “->” + node.left.val 这种写法有什么问题？（Python同理）

遍历法 vs 分治法

遍历法 = 一个小人拿着一个记事本走遍所有的节点

分治法 = 分配小弟去做子任务，自己进行结果汇总

代码剖析 - 整棵树路径 = 左子树路径 + 右子树路径


```java
public List<String> binaryTreePaths(TreeNode root) {
    List<String> paths = new ArrayList<>();

    if (root == null) {
        return paths;
    }
    if (root.left == null && root.right == null) {
        paths.add("" + root.val);
        return paths;
    }

    for (String leftPath : binaryTreePaths(root.left)) {
        paths.add(root.val + "->" + leftPath);
    }
    for (String rightPath : binaryTreePaths(root.right)) {
        paths.add(root.val + "->" + rightPath);
    }

    return paths;
}
```

```python
def binaryTreePaths(self, root):
    paths = []
    if not root:
        return paths
    if not root.left and not root.right:
        return [str(root.val)]

    for path in self.binaryTreePaths(root.left):
        paths.append(str(root.val) + "->" + path)
    for path in self.binaryTreePaths(root.right):
        paths.append(str(root.val) + "->" + path)

    return paths
```

二叉树上的分治法模板


```
public 返回结果类型 divideConquer(TreeNode root) {
    if (root == null) {
        处理空树应该返回的结果
    }
    // if (root.left == null && root.right == null) {
    //     处理叶子应该返回的结果
    //     如果叶子的返回结果可以通过两个空节点的返回结果得到
    //     就可以省略这一段代码
    // }

    左子树返回结果 = divideConquer(root.left)
    右子树返回结果 = divideConquer(root.right)
    整棵树的结果 = 按照一定方法合并左右子树的结果

    return 整棵树的结果
}
```

遍历法 vs 分治法

遍历法：通常会用到一个全局变量或者是共享参数

分治法：通常将利用 return value 记录子问题结果

二叉树上的分治法本质上也是在做遍历

先序？中序？后序？

判断二叉树是否是平衡的

https://www.lintcode.com/problem/balanced-binary-tree/

给定一棵二叉树，判断是否是平衡的

平衡二叉树定义：任意节点左右子树高度之差不超过1

Python 代码


```python
def isBalanced(self, root):
    is_balanced, _ = self.divideConquer(root)
    return is_balanced

# 定义：判断 root 为根的二叉树是否是平衡树并且返回高度是多少
def divideConquer(self, root):
    # 出口：如果是空树，直接返回是平衡的，高度为0
    if not root:
        return True, 0

    # 拆解：拆解到左右子树，得到左右子树是否平衡和高度的信息
    is_left_balanced, left_height = self.divideConquer(root.left)
    is_right_balanced, right_height = self.divideConquer(root.right)
    root_height = max(left_height, right_height) + 1

    if not is_left_balanced or not is_right_balanced:
        return False, root_height
    if abs(left_height - right_height) > 1:
        return False, root_height
    return True, root_height
```

Java 如何返回多个值?

函数需要同时返回 isBalanced 和 height 的信息

Java 代码


```java
// 定义：求出 root 为根的子树是否是平衡树且返回高度
private Result divideConquer(TreeNode root) {
    // 出口：空树的时候，返回 isBalanced=true, height=0
    if (root == null) {
        return new Result(true, 0);
    }

    // 拆解：拆解到左右两边得到左右子树的平衡信息和高度信息
    Result leftResult = divideConquer(root.left);
    Result rightResult = divideConquer(root.right);

    int height = Math.max(leftResult.height, rightResult.height) + 1;
    boolean isBalanced = true;

    if (!leftResult.isBalanced || !rightResult.isBalanced) {
        isBalanced = false;
    }
    if (Math.abs(leftResult.height - rightResult.height) > 1) {
        isBalanced = false;
    }

    return new Result(isBalanced, height);
}
```

```java
public class Solution {
    class Result {
        public boolean isBalanced;
        public int height;
        public Result(boolean isBalanced, int height) {
            this.isBalanced = isBalanced;
            this.height = height;
        }
    }
```