树的求和篇

树的求和属于树的题目中比较常见的，因为可以有几种变体，灵活度比较高，也可以考察到对于树的数据结构和递归的理解。一般来说这些题目就不用考虑非递归的解法了（其实道理是跟 LeetCode 总结 -- 树的遍历篇一样）。

LeetCode 中关于树的求和有以下题目：

1. Path Sum
2. Path Sum II
3. Sum Root to Leaf Numbers
4. Binary Tree Maximum Path Sum

1. Path Sum，这道题是树操作的题目，判断是否从根到叶子的路径和跟给定 sum 相同的。还是用常规的递归方法来做，递归条件是看左子树或者右子树有没有满足条件的路径，也就是子树路径和等于当前 sum 减去当前节点的值。结束条件是如果当前节点是空的，则返回 false，如果是叶子，那么如果剩余的 sum 等于当前叶子的值，则找到满足条件的路径，返回 true。算法的复杂度是树的遍历，时间复杂度是 O(n)，空间复杂度是 O(logn)。

代码如下：

```
public boolean hasPathSum(TreeNode root, int sum) {
    if(root == null){
        return false;
    }
    if(root.left == null && root.right == null && root.val == sum){
        return true;
    }
    return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);
}
```

做题时的感悟：

1. 返回值是或的关系，左右子树有一个符合条件即可：

```
return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);
```

2. 树的题目基本都是用递归来解决，主要考虑两个问题：

1）如何把问题分治成子问题给左子树和右子树。这里就是看看左子树和右子树有没有存在和是sum减去当前结点值得路径，只要有一个存在，那么当前结点就存在路径。

2）考虑结束条件是什么。这里的结束条件一个是如果当前节点为空，则返回 false。另一个如果是叶子，那么如果剩余的 sum 等于当前叶子的值，则找到满足条件的路径，返回 true。

2. Path Sum II，思路和Path Sum是完全一样的，只是需要输出所有路径，所以需要数据结构来维护路径，添加两个参数，一个用来维护走到当前结点的路径，一个用来保存满足条件的所有路径，思路上递归条件和结束条件是完全一致的，空间上这里会依赖于结果的数量了。时间复杂度仍然只是一次遍历O(n)，而空间复杂度则取决于满足条件的路径和的数量（假设是k条），则空间是O(klogn)。

代码如下：

```
public ArrayList<ArrayList<Integer>> pathSum(TreeNode root, int sum) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(root == null){
        return res;
    }
    helper(root, sum, new ArrayList<Integer>(), res);
    return res;
}

private void helper(TreeNode root, int sum, ArrayList<Integer> item, ArrayList<ArrayList<Integer>> res){
    item.add(root.val);
    if(root.left == null && root.right == null && root.val == sum){
        res.add(new ArrayList<Integer>(item));
    }
    if(root.left != null){
        helper(root.left, sum - root.val, item, res);
    }
    if(root.right != null){
        helper(root.right, sum - root.val, item, res);
    }
    item.remove(item.size() - 1);
}
```

3. Sum Root to Leaf Numbers，这道题相对于Path Sum多了两个变化，一个是每一个结点相当于个位上的值，而不是本身有权重，不过其实没有太大变化，每一层乘以10加上自己的值就可以了。另一个变化就是要把所有路径累加起来，这个其实就是递归条件要进行调整，Path Sum中是判断左右子树有一个找到满足要求的路径即可，而这里则是把左右子树的结果相加返回作为当前节点的累加结果即可。结束条件的话就是如果一个节点是叶子，那么我们应该累加到结果总和中，如果节点到了空节点，则不是叶子节点，不需要加入到结果中，直接返回0即可。算法的本质是一次先序遍历，所以时间是O(n)，空间是栈大小，O(logn)。树的题目在LeetCode中还是有比较大的比例的，不过除了基本的递归和非递归的遍历之外，其他大部分题目都是用递归方式来求解特定量。

代码如下：

```
public int sumNumbers(TreeNode root) {
    return helper(root, 0);
}

private int helper(TreeNode root, int num){
    if(root == null){
        return 0;
    }
    if(root.left == null && root.right == null){
        return num * 10 + root.val;
    }
    return helper(root.left, num * 10 + root.val) + helper(root.right, num * 10 + root.val);
}
```

自创，虽然代码稍显冗长，但思路清晰，可以使用通用模板。代码如下：

```
public int sumNumbers(TreeNode root) {
    if(root == null){
        return 0;
    }
    ArrayList<Integer> res = new ArrayList<Integer>();
    res.add(0);
    int val = 0;
    helper(root, val, res);
    return res.get(0);
}
```

```
    private void helper(TreeNode root, int val, ArrayList<Integer> res){
        val = val * 10 + root.val;
        if(root.left == null && root.right == null){
            res.set(0, res.get(0) + val);
            return;
        }
        if(root.left != null){
            helper(root.left, val, res);
        }
        if(root.right != null){
            helper(root.right, val, res);
        }
    }
```

做题时的感悟:

只有当方法中需要另外的参数（如 sum）时，才会新创建一个 helper 方法。

4. Binary Tree Maximum Path Sum，这道题是求树的路径和的题目，不过和平常不同的是这里的路径不仅可以从根到某一个结点，而且路径可以从左子树某一个结点，然后到达右子树的结点，就像题目中所说的可以起始和终结于任何结点。在这里树没有被看成有向图，而是被当成无向图来寻找路径。因为这个路径的灵活性，我们需要对递归返回值进行一些调整，而不是通常的返回要求的结果。在这里，函数的返回值定义为以自己为根的一条从根到子结点的最长路径（这里路径就不是当成无向图了，必须往单方向走）。这个返回值是为了提供给它的父结点计算自身的最长路径用，而结点自身的最长路径（也就是可以从左到右那种）则只需计算然后更新即可。这样一来，一个结点自身的最长路径就是它的左子树返回值（如果大于0的话），加上右子树的返回值（如果大于0的话），再加上自己的值。而返回值则是自己的值加上左子树返回值或者右子树返回值或者0（注意这里是“或者”，而不是“加上”，因为返回值只取一支的路径和）。所以整个算法就是维护这两个量，一个是自己加上左或者右子树最大路径作为它的父节点考虑的中间量，另一个就是自己加上左再加上右作为自己最大路径。在过程中求得当前最长路径时比较一下是不是目前最长的，如果是则更新。算法的本质还是一次树的遍历，所以复杂度是O(n)。而空间上仍然是栈大小O(logn)。

树的题目大多是用递归方式，但是根据要求的量还是比较灵活多变的，这道题是比较有难度的，他要用返回值去维护一个中间量，而结果值则通过参数来维护，需要一点技巧。

代码如下：

```
public int maxPathSum(TreeNode root) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    res.add(Integer.MIN_VALUE);
    helper(root, res);
    return res.get(0);
}

private int helper(TreeNode root, ArrayList<Integer> res){
    if(root == null){
        return 0;
    }
    int left = helper(root.left, res);
    int right = helper(root.right, res);
    int value = Math.max(left, 0) + Math.max(right, 0) + root.val;
    if(value > res.get(0)){
        res.set(0, value);
    }
    return Math.max(Math.max(left, right), 0) + root.val;
}
```

做题时的感悟：

由于最后的结果可能为负值，所以开始的ArrayList中要加入的是Integer.MIN_VALUE而不是0，不然会出现错误答案。

总结：

这篇总结主要讲了LeetCode中关于树的求和的题目。总体来说，求和路径有以下三种：

（1）根到叶子结点的路径；

（2）父结点沿着子结点往下的路径；

（3）任意结点到任意结点（也就是看成无向图）。

这几种路径方式在面试中经常灵活变化，不同的路径方式处理题目的方法也会略有不同，不过最复杂也就是Binary Tree Maximum Path Sum这种路径方式，只要考虑清楚仍然是一次递归遍历的问题。