

# Leetcode 题解 - 树

- [Leetcode 题解 - 树](#)
  - [递归](#)
    - [1. 二叉树的最大深度](#)
    - [2. 二叉树的直径](#)
    - [3. 平衡二叉树](#)
  - [层次遍历](#)
    - [1. 二叉树的层序遍历【top100】](#)
  - [前中后序遍历](#)
    - [1. 实现二叉树的前序遍历](#)
    - [1.1 二叉树的前序遍历-递归版](#)
    - [2. 实现二叉树的后序遍历](#)
    - [2.1 二叉树的后序遍历-递归版](#)
    - [3. 实现二叉树的中序遍历【top100】](#)
    - [3.1 二叉树的中序遍历-递归版](#)

## 递归

树是一种递归结构，很多树的问题可以使用递归（深度优先遍历DFS和广度优先遍历BFS）来处理。

### 1. 二叉树的最大深度

#### 104. 二叉树的最大深度 (简单)

[Leetcode](#) / [力扣](#)

```
var maxDepth = function(root) {  
    // BFS迭代  
    if (!root) return 0;  
    const queue = [];  
    queue.push(root);  
    const levels = 0; // 定义有多少层  
    while (queue.length) {  
        const size = queue.length; // 记住当前层有多少个节点  
        // 循环遍历每一层节点进行处理 (for循环每次执行完，都代表当前层遍历完毕)  
        for (let i = 0; i < size; i++) {  
            const cur = queue.shift();  
            if (cur.left) queue.push(cur.left);  
            if (cur.right) queue.push(cur.right);  
        }  
        levels++;  
    }  
    return levels;  
};
```

```
// DFS 前序遍历 递归
const maxDepth = function(root) {
  if (!root) return 0;
  let res = 0;
  const preorder = (node, curLevel) => {
    if (!node) return;
    res = Math.max(res, curLevel);
    preorder(node.left, curLevel + 1);
    preorder(node.right, curLevel + 1);
  };

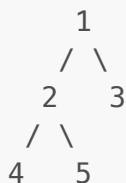
  preorder(root, 1);
  return res;
}
```

## 2. 二叉树的直径

### 543. 二叉树的直径 (简单)

[Leetcode](#) / [力扣](#)

给定二叉树



返回 3，它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

注意：两结点之间的路径长度是以它们之间边的数目表示。

思路：求每一个节点的左右子树的最大深度，然后左右子树最大深度之和的那个长度就是二叉树的直径。（和104题求二叉树的最大深度有关联）

```
var diameterOfBinaryTree = function(root) {
  // 采用的是先 后序迭代二叉树的最大深度，再求左右子树最大深度之和
  if (!root) return 0

  let res = 0
  const maxDepth = (node) => {
    if (!node) return 0
    const leftMaxDepth = maxDepth(node.left)
    const rightMaxDepth = maxDepth(node.right)
    res = Math.max(res, leftMaxDepth + rightMaxDepth)
    return Math.max(leftMaxDepth, rightMaxDepth) + 1
  }

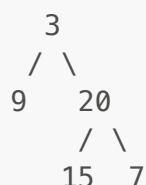
  maxDepth(root)
}
```

```
    return res
};
```

### 3. 平衡二叉树

#### 110. 平衡二叉树 (简单)

[Leetcode](#) / [力扣](#)



给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：

一个二叉树每个节点的左右两个子树的高度差的绝对值不超过 1。

输入：root = [3,9,20,null,null,15,7]

输出：true

思路：由104题二叉树的最大深度，自下而上的后序遍历衍生二来。求左右子树最大深度相减的绝对值。

注意：在遍历当前节点时，是否已经遇到标记为-1的不平衡二叉树节点了。

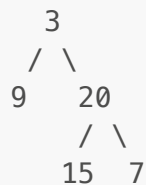
```
var isBalanced = function(root) {
    // 求以node为根节点的二叉树的最大深度
    const maxDepth = (node) => {
        if (!node) return 0; // 节点不存在，返回深度为0
        const leftMaxDepth = maxDepth(node.left);
        const rightMaxDepth = maxDepth(node.right);
        // 在遍历时已经遇到标记当前节点最大深度为-1的节点就不用往下执行了
        if (leftMaxDepth === -1 || rightMaxDepth === -1) {
            return -1;
        }
        // 左右子节点的最大深度相减绝对值小于1的话，已不平衡
        if (Math.abs(leftMaxDepth - rightMaxDepth) > 1) {
            return -1; // 标记当前节点的最大深度为 -1
        }
        // 处理根节点
        return Math.max(leftMaxDepth, rightMaxDepth) + 1;
    };
    return maxDepth(root) >= 0;
};
```

## 层次遍历

## 1. 二叉树的层序遍历 【top100】

### 102. 二叉树的层序遍历 (中等)

[Leetcode](#) / [力扣](#)



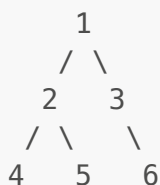
输入: root = [3,9,20,null,null,15,7]

输出: [[3],[9,20],[15,7]]

思路：用队列保存每一层的节点，当处理每一层节点时，应该先从队列的队头拉出来，放进当前层级的结果集，然后找它下一层的左右节点，有的话直接放进队列，循环往复就能得到结果

```
var levelOrder = function(root) {  
    if (!root) return [];  
  
    // 队列，先进先出  
    const res = [], queue = [];  
    queue.push(root);  
    while (queue.length) {  
        const levelNodes = []; // 新建一个存储当前层结点的数组  
        const size = queue.length; // 每轮循环遍历处理一层的节点  
        for (let i = 0; i < size; i++) {  
            const cur = queue.shift();  
            levelNodes.push(cur.val);  
            // 将遍历处理的节点的左右节点入队，等待后续的处理  
            if (cur.left) queue.push(cur.left);  
            if (cur.right) queue.push(cur.right);  
        }  
        res.push(levelNodes)  
    }  
    return res;  
};
```

## 前中后序遍历



- 层次遍历顺序: [1 2 3 4 5 6]

- 前序遍历顺序: [1 2 4 5 3 6]
- 中序遍历顺序: [4 2 5 1 3 6]
- 后序遍历顺序: [4 5 2 6 3 1]

## 1. 实现二叉树的前序遍历

### 144. 二叉树的前序遍历 (简单)

[Leetcode](#) / [力扣](#)

```
var preorderTraversal = function(root) {  
  /*  
    1. 声明一个空数组, 用来返回前序遍历的返回结果  
    2. 如果root节点为空, 则直接返回  
    3. 新建一个栈来存放非空节点, 然后根据“先进后出”的特点先将存在的右节点入栈,  
       再将存在的左节点入栈, 直到栈空为止  
  */  
  let arr = [];  
  if (!root) return arr;  
  const stack = [root]  
  while(stack.length) {  
    let cur = stack.pop();  
    arr.push(cur.val);  
    cur.right && stack.push(cur.right);  
    cur.left && stack.push(cur.left);  
  }  
  
  return arr;  
};
```

#### 1.1 二叉树的前序遍历-递归版

```
var preorderTraversal = function (root) {  
  if (!root) return [];  
  
  const preorder = (node, res) => {  
    if (!node) return; // // 递归的终止条件  
    res.push(node.val);  
    preorder(node.left, res);  
    preorder(node.right, res);  
  }  
  // 把 root 前序遍历结果放到 arr 中  
  let res = [];  
  preorder(root, res);  
  
  return res;  
};
```

## 2. 实现二叉树的后序遍历

## 145. 二叉树的后序遍历 (简单)

[Leetcode](#) / [力扣](#)

前序遍历为 root -> left -> right，后序遍历为 left -> right -> root。可以修改前序遍历成为 root -> right -> left，那么这个顺序就和后序遍历正好相反。

```
var postorderTraversal = function(root) {
  // 前: root -> left -> right
  // 后: left -> right -> root
  if (!root) return [];
  const res = [], stack = [];
  stack.push(root);
  while (stack.length) {
    const cur = stack.pop();
    res.push(cur.val);
    // 将前序遍历的 左右子树的遍历顺序翻转一下
    if (cur.left) stack.push(cur.left);
    if (cur.right) stack.push(cur.right);
  }
  res.reverse();
  return res;
};
```

### 2.1 二叉树的后序遍历-递归版

```
var postorderTraversal = function(root) {
  // 前: root -> left -> right
  // 后: left -> right -> root
  if (!root) return [];

  const postorder = (node, res) => {
    if (!node) return;
    postorder(node.left, res);
    postorder(node.right, res);
    res.push(node.val);
  };

  const res = [];
  postorder(root, res);
  return res;
};
```

## 3. 实现二叉树的中序遍历 【top100】

### 94. 二叉树的后序遍历 (简单)

[Leetcode](#) / [力扣](#)

```
var inorderTraversal = function(root) {
  if (!root) return [];
  let res = []; // 存储遍历后的结果
  let stack = []; // 用栈遍历左子树等
  let cur = root; // 用来找到最左边的子节点
  while (cur || stack.length) {
    while (cur) {
      stack.push(cur); // 把当前节点存入栈
      cur = cur.left; // 继续找左节点
    }
    // 退出while循环, 就代表当前节点的左节点已为空
    const node = stack.pop(); // 找到左节点为空的那个节点的根节点
    res.push(node.val);
    cur = node.right; // 继续寻找右节点
  }
  return res;
};
```

### 3.1 二叉树的中序遍历-递归版

```
var inorderTraversal = function(root) {
  if (!root) return [];
  // 递归
  const inorder = (node, res) => {
    if (!node) return;
    inorder(node.left, res);
    res.push(node.val);
    inorder(node.right, res);
  };
  const res = [];
  inorder(root, res);
  return res;
};
```