基础：二叉树的前、中、后序遍历

144. 二叉树的前序遍历 - 力扣（LeetCode）

94. 二叉树的中序遍历 - 力扣（LeetCode）

145. 二叉树的后序遍历 - 力扣（LeetCode）

Pre-In-Post遍历为二叉树的基础且重要的必会知识点，因为有许多相似，所以放在一起处理。

一.递归版

思路：区别只在于三行核心代码的顺序不同，

前序遍历：NLR，中左右

```
class Solution {
public:
    //--------------------------------------------------------------------------------
--------

    void preOrder(TreeNode* node,vector<int> &res){ //
        if(node == NULL) return;              //在递归函数中写上终止条件
        res.push_back(node->val);                   //中
        if(node->left) preOrder(node->left,res);   //左
        if(node->right) preOrder(node->right,res); //右

    }
     //--------------------------------------------------------------------------------
---------
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> res;
        preOrder(root,res);
        return res;
    }
};
```

中序遍历：LNR，左中右

```
    void inOrder(TreeNode* node,vector<int> &res){
        if(node == NULL) return;
        if(node->left) inOrder(node->left,res);    //左
        res.push_back(node->val);                  //中
        if(node->right) inOrder(node->right,res);  //右
    }
```

后序遍历：LRN，左右中

```
    void postOrder(TreeNode* node,vector<int> &res){
        if(node == NULL) return;
        if(node->left) postOrder(node->left,res);   //左
        if(node->right) postOrder(node->right,res); //右
        res.push_back(node->val);                   //中
    }
```

二.迭代版

思路：总体上是利用栈

> 要注意前序和后序遍历相对来说比较相似，中序遍历要单独拿出来分析。
>
> 原因在于，前序遍历和后序遍历，是访问和处理一致的遍历方式，而中序遍历情况较为复杂。
>
> **访问：指针指到当前节点**
>
> **处理：当前节点的值传进res数组。**

前序遍历（stack）

```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> res;
        stack<TreeNode*> s;
        if(root == NULL) return{};
        s.push(root);

        while(!s.empty()){
            TreeNode* tmp = s.top();
            s.pop();
            res.push_back(tmp->val);
            if(tmp->right) s.push(tmp->right);
            if(tmp->left) s.push(tmp->left);
        }

        return res;
    }
};
```

后序遍历（stack）

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        if(root == NULL) return{};
        vector<int> res;
        stack<TreeNode*> s;
        s.push(root);

        while(!s.empty()){
            TreeNode* tmp = s.top();
            s.pop();
            res.push_back(tmp->val);
            if(tmp->left) s.push(tmp->left);
            if(tmp->right) s.push(tmp->right);
        }

        reverse(res.begin(),res.end());
        return res;
    }
};
```

中序遍历（单独分析）（cur指针）

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        if(root == NULL) return{};
        stack<TreeNode*> s;

        TreeNode* cur = root;

        while(cur != NULL || !s.empty()){
            if(cur != NULL){  //扫到非空节点，就继续往左孩子走
                s.push(cur);
                cur = cur->left;
            }
            else{
                cur = s.top();
                s.pop();
                res.push_back(cur->val);
                cur = cur->right;
            }
        }
        return res;
    }
```

```
};
```

三.统一迭代版（新思路：栈 + 标记法）

中序遍历：LNR

从中序遍历开始引出。

前面提到，中序遍历的迭代和前后序遍历的迭代有区别的原因在于，

前序，后序遍历的迭代，都是访问和处理一致的。（后序是利用了前序，并进行reverse，所以也可以看成是访问和处理相一致。）

因为使用stack，所以访问顺序反过来，为右中左。

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        stack<TreeNode*> s;
        if(root == NULL) return{};
        else s.push(root);


        while(!s.empty()){
            TreeNode* tmp = s.top();
            if(tmp != NULL){
                s.pop();
                //--------------------------------------------------------------------
----------
                if(tmp->right) s.push(tmp->right);// 添加右节点（空节点不入栈）

                s.push(tmp); // 添加中节点
                s.push(NULL);// 中节点访问过，但是还没有处理，加入空节点做为标记。

                if(tmp->left) s.push(tmp->left);// 添加左节点（空节点不入栈）
                //--------------------------------------------------------------------
----------
            }
            else{         // 只有遇到空节点的时候，才将下一个节点放进结果集
                s.pop(); //弹出空元素
                tmp = s.top(); //重新指向栈顶元素
                s.pop(); //处理step1
                res.push_back(tmp->val); // 处理step2
            }
        }
        return res;
```

```
    }
};
```

前序遍历：NLR

因为迭代使用stack，所以结点访问是反过来的，即右左中。

```
    if(tmp->right) s.push(tmp->right); //右
    if(tmp->left) s.push(tmp->left); //左

    s.push(tmp);  //中结点
    s.push(NULL);
```

后序遍历：LRN

同理，因使用stack，反过来，所以中右左。

```
    s.push(tmp); // 添加中节点
    s.push(NULL);// 中节点访问过，但是还没有处理，加入空节点做为标记。
    if(tmp->right) s.push(tmp->right);// 添加右节点（空节点不入栈）
    if(tmp->left) s.push(tmp->left); //左
```