二叉树的四种遍历手法

前中后三种遍历手法的代码及其相似，都是套路。

二叉树结点定义

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
```

递归写法

Preorder Traversal

```
class Solution {
public:
    void helper(TreeNode* root, vector<int>& res) {
        if(root == NULL) return;
        res.push_back(root->val);
        helper(root->left, res);
        helper(root->right, res);
    }
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> res;
        helper(root, res);
        return res;
    }
};
```

Inorder Traversal

```
class Solution {
public:
    void helper(TreeNode* root, vector<int>& res) {
        if(root == NULL) return;
        helper(root->left, res);
        res.push_back(root->val);
        helper(root->right, res);
    }
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        helper(root, res);
        return res;
    }
};
```

Postorder Traversal

```cpp
class Solution {
public:
    void helper(TreeNode* root, vector<int>& res) {
        if(root == NULL) return;
        helper(root->left, res);
        helper(root->right, res);
        res.push_back(root->val);
    }
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> res;
        helper(root, res);
        return res;
    }
};
```

迭代写法

Preorder Traversal

```cpp
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> res;
        stack<TreeNode*> st;
        while(root || !st.empty()) {
            while(root) {
                res.push_back(root->val);
                st.push(root);
                root = root->left;
            }
            TreeNode* temp = st.top();
            st.pop();
            root = temp->right;
        }
        return res;
    }
};
```

n叉树的前序遍历

```cpp
class Solution {
public:
    vector<int> preorder(Node* root) {
        vector<int> res;
        if(root == NULL) return res;
        stack<Node*> st;
        st.push(root); // 这里要先将根节点放进栈里面，与二叉树的不同
        while(!st.empty()) {
            Node* cur = st.top();
            st.pop();
            res.push_back(cur->val);
            int size = cur->children.size();
            for(int i = size - 1; i >= 0; i--) {  // 这里也要注意，因为栈的特性，所以孩子进入栈的次序是从后往前。刚开始本来也想着跟二叉树一样，第一次先放第一个孩子，但是发现这种方法根本行不通，因为后面想放其他孩子的时候没有了索引跟踪，所以不知道已经进行到哪个孩子了，除非使用vector的删除功能，孩子放一个就删掉一个，可惜时间复杂度会比较高
```

```
                st.push(cur->children[i]);
            }
        }
        return res;
    }
};


// 以上这个方法跟我今天写得一直诶(2020.07.11)
// 下面是这个写法的二叉树版本
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> res;
        if(root == nullptr) return res;
        stack<TreeNode*> st;
        st.push(root);
        while(!st.empty()) {
            TreeNode* cur = st.top();
            res.push_back(cur->val);
            st.pop();
            if(cur->right) {
                st.push(cur->right);
            }
            if(cur->left) {
                st.push(cur->left);
            }
        }
        return res;
    }
```

Inorder Traversal

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        stack<TreeNode*> st;
        while(root || !st.empty()) {
            while(root) {
                st.push(root);
                root = root->left;
            }
            TreeNode* temp = st.top();
            st.pop();
            res.push_back(temp->val);
            root = temp->right;
        }
        return res;
    }
};
```

Postorder Traversal

这个有点小复杂（参考解法）：https://leetcode.com/explore/learn/card/data-structure-tree/134/traverse-a-tree/930/discuss/327048/Cpp-3-iterative-solutions

所以我们的策略是跟前序遍历类似，但是遍历完根节点后，要先遍历右结点，再遍历左结点，最后再翻转结果向量即可。

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> res;
        stack<TreeNode*> st;
        while(root || !st.empty()) {
            while(root) {
                st.push(root);
                res.push_back(root->val);
                root = root->right;  // 注意这里与前序遍历的细微差别
            }
            TreeNode* temp = st.top();
            st.pop();
            root = temp->left;
        }
        reverse(res.begin(), res.end());  // 记得翻转
        return res;
    }
};
```

层序遍历

当然这得需要使用队列去实现啦，即BFS。

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> res;
        if(root == NULL) return res;  // 不要忘记这个边界条件判断，为防止遗漏，最好是不管后面代码怎样，先加上
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()) {
            int size = q.size();
            vector<int> temp;
            for(int i = 0; i < size; i++) {
                TreeNode* cur = q.front();
                q.pop();
                temp.push_back(cur->val);
                if(cur->left) q.push(cur->left);
                if(cur->right) q.push(cur->right);
            }
            res.push_back(temp);
        }
        return res;
    }
};
```

根据中序遍历与后序遍历重建二叉树

注意体会对postorder向量中用于指向根节点的下标p的引用。根据中序遍历和前序遍历重建二叉树一样。

```cpp
class Solution {
public:
    TreeNode* helper(vector<int>& inorder, vector<int>& postorder, int l, int r, int& p) {  // 注意这里使用p的引用非常重要，因为使用了引用，随着递归的进行，p的值会慢慢地从右边移动到左边
        if(l > r) return NULL;
        TreeNode* root = new TreeNode(postorder[p]);
        p--;
        if(l == r) return root;
        int k;
        for(k = l; k <= r; k++) {
            if(inorder[k] == root->val) break;
        }
        root->right = helper(inorder, postorder, k+1, r, p); // 注意下面这两句的顺序不能反过来
        root->left = helper(inorder, postorder, l, k-1, p); // 当程序运行都这里的时候，p的值与上面已经不一样了
        return root;
    }
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        if(inorder.empty()) return NULL;
        int l = 0;
        int r = inorder.size() - 1;
        int p = postorder.size() - 1;
        return helper(inorder, postorder, l, r, p);
    }
};
```