# 前言

注：本文没有囊括所有遍历方法，本文只记录作者平常使用的遍历方法

## 结构定义

```
# Definition for a binary tree node.
class TreeNode:
     def __init__(self, val=0, left=None, right=None):
         self.val = val
         self.left = left
         self.right = right
```

## 概述

前、中、后序的递归方法都很简单，迭代方法需要用到栈，每个方法的具体实现各有不同，而层次遍历要用到队列

前、中、后序的递归方法伪代码如下

```
def func(root):
    if root:
        # 前序 visit root
        func(root.left)
        # 中序 visit root
        func(root.right)
        # 后续 visit root
```

而迭代方法具体提到再说

# 前序

相关题目 144. 二叉树的前序遍历

## *递归*

伪代码

```
def func(root):
    if root:
        visit root # do something
        func(root.left)
        func(root.right)
```

提交代码

```
class Solution:
    def preorderTraversal(self, root: TreeNode) -> List[int]:
        self.ans = []
        self.helper(root)
        return self.ans

    def helper(self,root):
        if root:
            self.ans.append(root.val)   # visit root
            self.helper(root.left)
            self.helper(root.right)
```

## *迭代*

前序遍历的递归实现一个简单的方法就是，利用栈来遍历，每次遇到节点的时候先入栈右结点再入栈左结点，这样访问的序列就是先序
伪代码

```
def func(root):
    如果root==None，不做任何事
    stack = [root]
    # 栈非空
    while stack and root:
        # 弹出栈顶
        top = stack.pop()
        访问top
        if top.right:
            stack.append(top.right)
        if top.left:
            stack.append(top.left)
```

提交代码

```
class Solution:
    def preorderTraversal(self, root: TreeNode) -> List[int]:
        if not root:
            return []
        stack = [root]
        ans = []
        while stack:
            top = stack.pop(-1)
            ans.append(top.val)
            if top.right:
                stack.append(top.right)
            if top.left:
                stack.append(top.left)
        return ans
```

# 中序

94. 二叉树的中序遍历

## *递归*

伪代码

```
def func(root):
    if root:
        func(root.left)
        visit root # do something
        func(root.right)
```

提交代码

```python
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        self.ans = []
        self.helper(root)
        return self.ans

    def helper(self,root):
        if root:
            self.helper(root.left)
            # visit root
            self.ans.append(root.val)
            self.helper(root.right)
```

## *迭代*

观察中序遍历的过程，**中序始终是往左孩子走**，走到空结点时再返回，然后访问上一个根节点，再访问右孩子，而右子树的遍历过程也是中序遍历，也就是重复上述过程，整理上述遍历过程可写出迭代遍历的伪代码

```python
def func(root):
    if not root:
        return # do nothing
    # 初始栈空，不存放结点
    stack = []
    # 从根节点开始遍历
    top = root
    while stack or root:
        if top:
            # 当前根节点不为空，不断尝试往左子树走，同时结点入栈
            while top:
                stack.append(top)
                top = top.left
        # 当碰到空结点的时候
        else:
            # 弹出栈顶结点
            top = stack.pop()
            visit top #访问根节点
            # 往右子树遍历
            top = top.right
```

提交代码

```python
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        if not root:
            return # do nothing

        ans = []
        stack = []
        top = root
        while stack or top:
            if top:
                # 当前根节点不为空，不断访问左孩子
                while top:
                    stack.append(top)
                    top = top.left
            else:
```

```
                # 在访问左孩子的过程中碰到根节点了
                # 弹出栈顶结点
                top = stack.pop()
                ans.append(top.val) # visit root
                # 往右子树走，重复上述过程
                top = top.right

        return ans
```

# 后序

145. 二叉树的后序遍历

## *递归*

伪代码

```
def func(root):
    if root:
        func(root.left)
        func(root.right)
        visit root # do something
```

提交代码

```
class Solution:
    def postorderTraversal(self, root: TreeNode) -> List[int]:
        self.ans = []
        self.helper(root)
        return self.ans

    def helper(self,root):
        if root:
            self.helper(root.left)
            self.helper(root.right)
            self.ans.append(root.val)
```

## *迭代*

后序的迭代是基于中序迭代的，理解了中序迭代，后序迭代就很容易理解
另外后序迭代有多种方法，这里我们依次介绍

## 基于中序迭代

中序迭代的一个重要思想就是，第一次遇到根节点的时候不进行访问，第二次遇到根节点的时候也就是从左子树返回的时候才访问根节点
而后序迭代的思想是，第一次遇到根节点的时候不访问，此时往左子树走，第二次遇到根节点的时候仍然不访问，此时往右子树走，而第三次遇到根节点，也就是从右子树返回的时候，才会访问根节点，这也符合后序递归的过程
从上面的分析来看，如何标记从右子树返回时遇到根节点才是关键，因为只有这个时候我们才需要访问根节点

下面我们尝试写一下伪代码

```
def func(root):
    if not root:
        return # do nothing
    # 初始栈空，不存放结点
    stack = []
    # 从根节点开始遍历
    top = root
    while stack or root:
        if top:
            # 当前根节点不为空，不断尝试往左子树走，同时结点入栈
            while top:
                stack.append(top)
                top = top.left
        # 当碰到空结点的时候
        else:
            # 弹出栈顶结点
            top = stack.pop()
            # 从左子树返回时不访问根节点
            # visit top #访问根节点
            # 往右子树遍历
            top = top.right
```

上面代码的问题就是我们并不知道当top==None的时候是何种情况，而我们期望能够标记出从右孩子处返回的情况
如何标记当前遇到根节点的情况是从右孩子处返回？一个简单的想法就是利用指针，我们用last来标记**最后访问的结点**，如果

```
top.right == last 为True
```

那么我们就能认为当前遇到根节点top可以访问，而进一步的问题时我们怎么在遍历的时候及时的更新last指针呢？
注意到last指针的功能是**标记最后访问的结点**，而只有在visit root的时候才需要更新，那么visit root一共有下述情况

1. root.right== None ，也就是根节点的右子树为空，此时毫无疑问访问根节点，然后往右子树遍历
2. root.right == last，也就是从右子树返回的时候，此时访问根节点

又由于我们需要判断两次root是否能够访问，那么我们与中序迭代遍历时对栈顶元素的操作不同，只有确定会访问的时候才会弹出栈顶，而其余情况只是取得栈顶元素
top = stack[-1]
只有在上述两种情况下更新，那么我们基于上述的分析更新伪代码

```
def func(root):
    if not root:
        return # do nothing
    # 初始栈空，不存放结点
    stack = []
    # 从根节点开始遍历
    top = root
    last = None
    while stack or top:
```

```
        if top:
            # 当前根节点不为空，不断尝试往左子树走，同时结点入栈
            while top:
                stack.append(top)
                top = top.left
        # 当碰到空结点的时候
        else:
            # 取得栈顶结点
            top = stack[-1]
            # 左子树为空或者从右子树处返回的时候访问根节点
            if top.right== None and top.right == last:
                visit top   # 访问根节点
                last = top  # 更新last
                stack.pop() # 弹出栈顶元素
                top = None # 访问根节点后需要置空，取得下一个栈顶元素判断
            else:
                # 不满足上述两种情况
                top = top.right
```

下面是提交的代码

```
class Solution:
    def postorderTraversal(self, root: TreeNode) -> List[int]:
        if not root:
            return []# do nothing
        ans = []
        # 初始栈空，不存放结点
        stack = []
        # 从根节点开始遍历
        top = root
        last = None
        while stack or top:
            if top:
                # 当前根节点不为空，不断尝试往左子树走，同时结点入栈
                while top:
                    stack.append(top)
                    top = top.left
            # 当碰到空结点的时候
            else:
                # 获取栈顶
                top = stack[-1]
                # 左子树为空或者从右子树处返回的时候访问根节点
                if top.right == None or top.right == last:
                    print(top.val)
                    ans.append(top.val)
                    last = top
                    stack.pop()
                    top = None
                else:
                    # 往右子树遍历
                    top = top.right

        return ans
```

## 基于翻转的迭代

这是一个非常取巧的做法，我们仍然使用栈来迭代，但是我们的访问现在变得很简单，就是按照**中右左**的顺序访问，当我们访问完毕的时候，我们**翻转**访问列表，这样就能得到**左右中**的访问序列
这个做法虽然简单，但是我们有时候需要用基于中序的迭代去做题，例如236. 二叉树的最近公共祖先需要我们找到两个结点的最近公共祖先，那么就可以使用上面的方法去做，依据就是访问到某个结点p的时候，此时访问栈`stack`里面的结点都是p的祖先
因此这个方法只能用于得到后序访问的序列

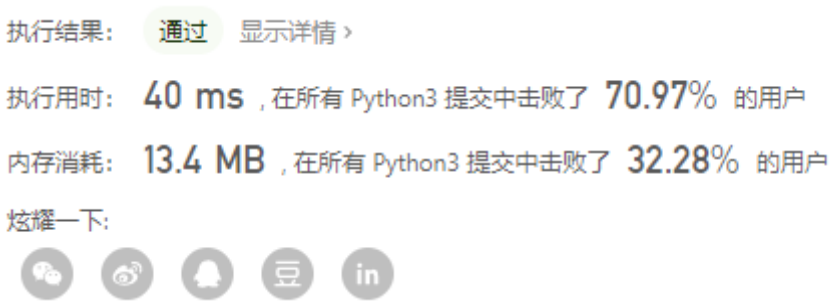
伪代码

```
def func(root):
    if not root:
        return # do nothing
    stack = [root]
    visit_list = []
    while stack:
        top = stack的栈顶
        visit_list添加top
        if top的左孩子存在:
            stack添加top的左孩子
        if top的右孩子存在:
            stack添加top的右孩子

    return 反转后的visit_list
```

提交代码

```
class Solution:
    def postorderTraversal(self, root: TreeNode) -> List[int]:
        if not root:
            return [] # do nothing
        stack = [root]
        visit_list = []
        while stack:
            top = stack.pop()
            visit_list.append(top.val)
            if top.left:
                stack.append(top.left)
            if top.right:
                stack.append(top.right)

        return visit_list[::-1]
```

结果：




注：执行用时由于力扣服务器资源的影响会不稳定，特别是python提交的代码，对本例来说关注是否能通过即可

# 前、中、后序迭代的统一解决方法

出处：
https://leetcode-cn.com/problems/binary-tree-inorder-traversal/solution/yan-se-biao-ji-fa-yi-chong-tong-yong-qie-jian-ming/
下面是原文

官方题解中介绍了三种方法来完成树的中序遍历，包括：

- 递归
- 借助栈的迭代方法
- 莫里斯遍历

在树的深度优先遍历中（包括前序、中序、后序遍历），递归方法最为直观易懂，但考虑到效率，我们通常不推荐使用递归。

栈迭代方法虽然提高了效率，但其嵌套循环却非常烧脑，不易理解，容易造成“一看就懂，一写就废”的窘况。而且对于不同的遍历顺序（前序、中序、后序），循环结构差异很大，更增加了记忆负担。

因此，我在这里介绍一种“颜色标记法”（瞎起的名字……），**兼具栈迭代方法的高效，又像递归方法一样简洁易懂，更重要的是，这种方法对于前序、中序、后序遍历，能够写出完全一致的代码。**

其核心思想如下：

使用颜色标记节点的状态，新节点为白色，已访问的节点为灰色。
如果遇到的节点为白色，则将其标记为灰色，然后将其右子节点、自身、左子节点依次入栈。
如果遇到的节点为灰色，则将节点的值输出。
使用这种方法实现的中序遍历如下：

```
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        WHITE, GRAY = 0, 1
        res = []
        stack = [(WHITE, root)]
        while stack:
            color, node = stack.pop()
            if node is None: continue
            if color == WHITE:
                stack.append((WHITE, node.right))
                stack.append((GRAY, node))
                stack.append((WHITE, node.left))
            else:
                res.append(node.val)
        return res

作者：hzhu212
链接：https://leetcode-cn.com/problems/binary-tree-inorder-traversal/solution/yan-se-biao-ji-fa-yi-chong-tong-yong-qie-jian-ming/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

如要实现前序、后序遍历，只需要调整左右子节点的入栈顺序即可。

---

分析上面的代码，就是用0、1标记结点的访问次数，当结点为GRAY的时候，我们就将其输入，否则我们按照**右-中-左**的顺序入栈，然后弹出来还是**左-中-右**的访问次序
如果是后序遍历的话，只需要改为

```python
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        WHITE, GRAY = 0, 1
        res = []
        stack = [(WHITE, root)]
        while stack:
            color, node = stack.pop()
            if node is None: continue
            if color == WHITE:
                stack.append((GRAY, node))
                stack.append((WHITE, node.right))
                stack.append((WHITE, node.left))
            else:
                res.append(node.val)
        return res
```

即按照**中-右-左**的顺序入栈

然后我们以中序遍历为例，代码还可以在实现层面上优化，来自评论区

```python
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        stack,rst = [root],[]
        while stack:
            i = stack.pop()
            if isinstance(i,TreeNode):
                stack.extend([i.right,i.val,i.left])
            elif isinstance(i,int):
                rst.append(i)
        return rst
```

这里就不解释了，涉及到其它无关的知识

最后这个方法虽然很神奇，但是会造成节点两次入栈，增加空间开销

# 层次

## *迭代*

层次遍历的写法就很简单了，这里就直接贴代码了

```python
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []
        ans = []
        que = [root]
        while que:
            size = len(que)
            lst = []
            for i in range(size):
                front = que.pop(0)
                lst.append(front.val)
                if front.left:
                    que.append(front.left)
                if front.right:
```

```
                que.append(front.right)
            ans.append(lst)
        return ans
```