94-二叉树的中序遍历

leetcode #94-二叉树的中序遍历(简单)

> 给定一个二叉树的根节点 `root` ，返回它的 **中序** 遍历。节点类如下。

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) {
        this.val = val;
    }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}
```

解法一：递归

算法描述

最为简单直观的一种方法。有一驱动方法传入最初的根节点，驱动实际的遍历方法`void inorderTraversalCur(TreeNode root, List<Integer> res)`，传入本次访问的节点和用于输出的结果列表res。基准情形是`root == null`，直接返回。只要root不为null，对其左右子节点递归调用遍历方法，如下三行的顺序决定了遍历方式。2,1,3为前序遍历，1,2,3为中序遍历，1,3,2为后序遍历，语句的排列和遍历顺序的定义一致，非常容易记忆。

前序(根左右)	中序(左根右)	后序(左右根)
2. res.add(root.val); 1. preorderTraversalCur(root.left, res); 3. preorderTraversalCur(root.right, res);	1. inorderTraversalCur(root.left, res); 2. res.add(root.val); 3. inorderTraversalCur(root.right, res);	1. postorderTraversalCur(root.left, res); 3. postorderTraversalCur(root.right, res); 2. res.add(root.val);

时空复杂度

n为该二叉树的节点总数。每个节点都会被遍历(遍历方法以其为参数的执行次数)且只遍历一次，因此时间复杂度为O(n)。

空间复杂度取决于栈深，而栈深与该二叉树的形状有关，如果为链状，达到最大空间复杂度O(n)，如果为完全二叉树(complete binary tree)，栈深为O(logn)。

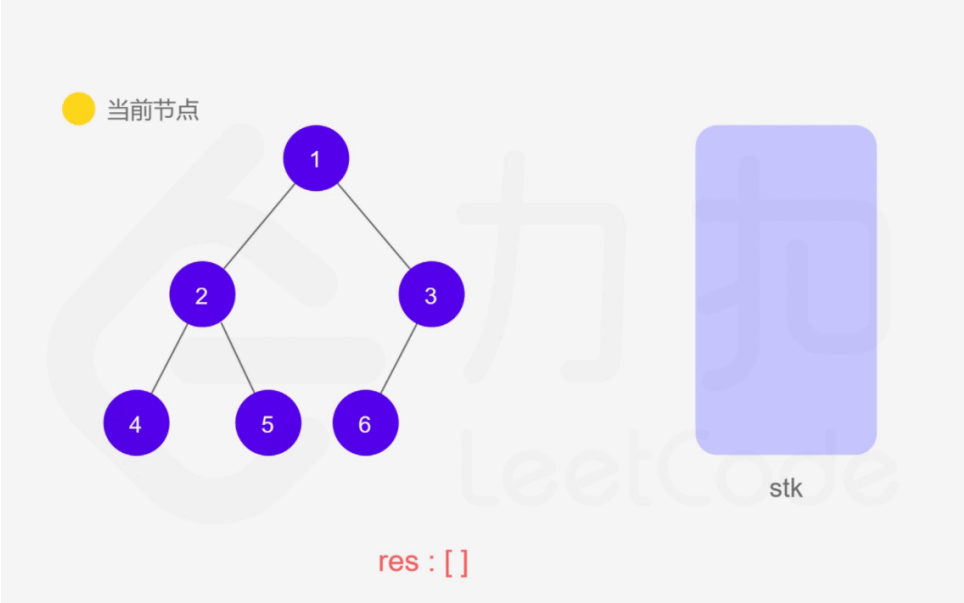
代码

```
public List<Integer> inorderTraversalCur(TreeNode root) {
    List<Integer> res = new ArrayList<>();
    inorderTraversalCur(root, res);
    return res;
}

private void inorderTraversalCur(TreeNode root, List<Integer> res) {
    if(root != null){
        inorderTraversalCur(root.left, res);
        res.add(root.val);
        inorderTraversalCur(root.right, res);
    }
    return;
}
```

解法二：栈+迭代

算法描述

解法一在方法调用过程中，隐式地使用了方法栈。本解法利用循环迭代来代替遍历方法的递归调用，配合一个显式的TreeNode栈来控制节点的访问和输出顺序。请结合动图和代码注释理解。如下动图来自leetcode。



时空复杂度

n为该二叉树的节点总数。每个节点都会被遍历(遍历方法以其为参数的执行次数)且只遍历一次，因此时间复杂度为O(n)。

空间复杂度取决于栈深，而栈深与该二叉树的形状有关，如果为链状，达到最大空间复杂度O(n)，如果为完全二叉树(complete binary tree)，栈深为O(logn)。

代码

```java
public List<Integer> inorderTraversalLoop(TreeNode root) {
    List<Integer> res = new ArrayList<>();
    Deque<TreeNode> stack = new LinkedList<>();
    // 同时满足root为null及stack为空，才将结果返回
    while(root != null || !stack.isEmpty()) {
        // root不为null时入栈，向left方向推进直到叶子节点
        while(root != null) {
            stack.push(root);
            root = root.left;
        }
        // 向left推进到叶子节点后，将该叶子节点弹出并将其val加入结果
        root = stack.pop();
        res.add(root.val);
        // 向right推进，如果right也是null，那么回到一开始的while判断栈
        // 是否空，不空则跳过第二个while，再次弹出栈顶。也就是某节点的
        // 左子树遍历完毕，回到该节点。
        root = root.right;
    }
    return res;
}
```

解法三：标记栈+迭代

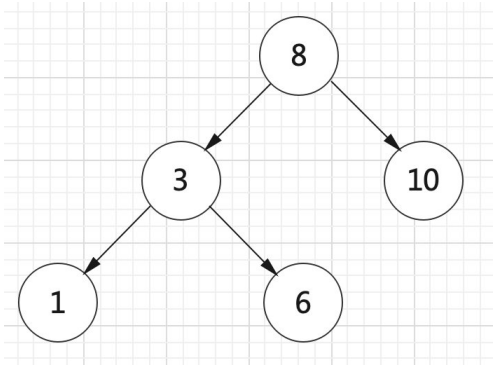
算法描述

用一个栈存储节点，先将root压入栈中。从root出发，每次弹出当前栈顶并考察其左右孩子，以中序遍历的逆序(即“右根左”)，将节点压入栈中(此时栈为[左 根 右]，右侧为栈底)，注意“根”此时为第二次入栈。对每个节点都如此操作，每一个节点都会入栈两次，出栈两次，当一个节点x第二次弹出时，说明x的左子树已完成遍历，x此时是被遍历的节点，将其值加入到res结果中。因此在x第一次弹出的时候需要记录一个`isMarked = true`的信息，表示已被弹出过一次，第二次弹出时就可以根据`if(isMarked)`来决定是否完成对x的遍历(加入到res中)。

如下图中序遍历为[1, 3, 6, 8, 10]，过程如下(括号代表被遍历过一次)。

```
[8] // 8入栈
8 [] // 8出栈并考察是否有左右孩子
[3 (8) 10] // 按右孩子>自己>左孩子的顺序入栈，自己入栈时将标记置为已被弹出过一次
3 [(8) 10]
[1 (3) 6 (8) 10]
1 [(3) 6 (8) 10]
[(1) (3) 6 (8) 10]
[(3) 6 (8) 10], res = [1]
[6 (8) 10], res = [1, 3]
6 [(8) 10]
[(6) (8) 10]
```

```
[(8) 10], res = [1, 3, 6]
[10], res = [1, 3, 6, 8]
10 []
[(10)]
[], res = [1, 3, 6, 8, 10]
```



与解法二一样是配合栈的迭代方式，区别在于这个栈用于保存一个ColorNode类，该类的实例持有一个TreeNode类实例和一个用于标记该TreeNode是否已被访问过的布尔值isMarked。该方法利用迭代，但在书写上与递归写法有同样的优点，即对于二叉树的前序，中序，后序遍历，写法类似，只需改动在考察一个节点C的左孩子L和右孩子R后，CLR三者的入栈顺序即可。前序遍历按RLC(“根左右”的逆序“右左根”)顺序入栈，中序遍历按RCL(“左根右”的逆序“右根左”)顺序入栈，后序遍历按CRL(“左右根”的逆序“根右左”)。代价是多出了ColorNode类的空间开销，以及对每个元素的两次入栈两次出栈的时间开销。

时空复杂度

每个节点均会入栈两次出栈两次，系数比常规迭代方法大，但时间复杂度仍为O(n)，空间复杂度为栈最大深度，当树为链状时达到最大复杂度O(n)。

代码

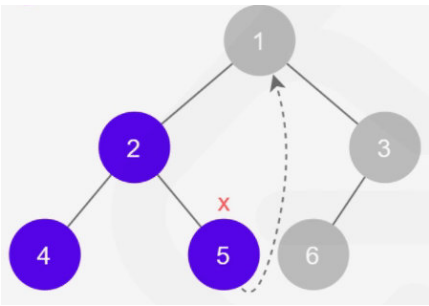
```java
/**
 * 内部类
 * 持有一个TreeNode及是否被标记过的信息
 */
private class ColorNode{
    private TreeNode treeNode;
    // false表示未被标记过(未访问过)，true表示标记过(访问过)
    private boolean isMarked;

    public ColorNode(TreeNode treeNode, boolean isMarked) {
        this.treeNode = treeNode;
        this.isMarked = isMarked;
    }
}

public List<Integer> inorderTraversalColor(TreeNode root) {
    if(root == null) {
        return new ArrayList<>();
    }
```

```
20      List<Integer> res = new ArrayList<>();
21      Deque<ColorNode> stack = new LinkedList<>();
22      // 将不为null的root推入颜色标记节点栈中
23      stack.push(new ColorNode(root, false));
24      // 栈非空则执行
25      while(!stack.isEmpty()) {
26          // 弹出栈顶颜色标记节点
27          ColorNode colorNode = stack.pop();
28          // 若未被标记过
29          if(!colorNode.isMarked) {
30              // colorNode若有右子节点则将该右子节点与未标记信息组合成颜色标记节点后入栈
31              if(colorNode.treeNode.right != null) {
32                  stack.push(new ColorNode(colorNode.treeNode.right, false));
33              }
34              // 将colorNode打上标记后推回栈内
35              colorNode.isMarked = true;
36              stack.push(colorNode);
37              // colorNode若有左子节点则将该左子节点与未标记信息组合成颜色标记节点后入栈
38              if(colorNode.treeNode.left != null) {
39                  stack.push(new ColorNode(colorNode.treeNode.left, false));
40              }
41          }
42          else {
43              // 访问到一个被标记过的节点，加入结果
44              res.add(colorNode.treeNode.val);
45          }
46      }
47      return res;
48  }
```

解法四：Morris遍历算法

算法描述

二叉树的前中后序遍历均涉及到从根出发，向叶子方向推进后再返回的过程。无论是递归方法还是显式地使用栈的迭代方法，能够返回的关键均在于栈。思考如何在不使用栈的情况下实现二叉树遍历时的返回动作。以下图的中序遍历为例，在完成[4, 2, 5]的遍历之后要遍历1(将1加入到结果中)，如果在完成5的遍历后能够通过5的指针信息返回到1，就有可能在不使用栈的情况下完成二叉树的遍历。实际上这就是线索二叉树(Threaded Binary Tree)的思想，而Morris遍历算法正是这种思想的具体应用。



该遍历算法由Joseph M.Morris于1979年提出(Morris遍历算法论文)。该算法遍历到每一个有左子树的节点x时，首先找到x在中序遍历下的前驱p(x及其左子树完成一次中序遍历后x在这个遍历结果中的前一个节点，如上图中x为1时，p为5)。p必然是一个叶子结点，p的右子节点pr为空，即p.right == null。赋予pr到x的线索，即p.right = x，使得在后续遍历推进到p时，可以根据这个线索回到x，这也是线索树名称的含义。可以结合下述算法过程，动图和代码注释理解本解法。动图来自leetcode。

从root出发，当root有左子节点(即有左子树)时：

- 找到root的前驱predecessor(即root及其左子树完成一次中序遍历后root的前驱，简称p)
  - root.left没有右孩子时p即为root.left，否则为root.left的右孩子链上最末端的节点(root.left.right.right.right...)。

  根据上述确定了p后，若其右子节点为空时(p为叶子节点，此时其右子节点必然为null)：
  - `predecessor.right = root;` // 赋予将来用于返回的线索 (所以第二次为x寻找p时其右子节点为此时的x)
  - `root = root.left;` // 向左遍历推进
- p的右子节点不为空时，说明对于此时的root来说，其左子树已经完成了遍历。(遍历动作通过p的线索返回到x后，再次对x寻找其p)
  - `res.add(root.val);` // 将root的值加入结果中 (“左根右”的遍历过程中将“根”加入结果)
  - `predecessor = null;` // 重置predecessor
  - `root = root.right;` // 向右遍历推进

当root无左子节点时：

- `res.add(root.val);` // 将root的值加入结果中 (“左根右”的遍历过程中将“左”(当p是叶子结点时)或“根”(当p是非叶子结点时)加入结果)
- `root = root.right` // 向右边遍历推进

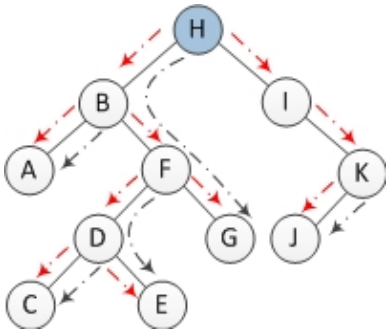
时空复杂度

时间复杂度：O(n)

二叉树节点数为n。

简单理解：n个节点共有n-1条边，每条边最少走一次，最多走三次。第一次(下图红线)是遍历推进(`root = root.left`和`root = root.right`)，其余两次(下图黑线)产生于寻找predecessor的过程，每个predecessor都会被寻找两次。于是整个遍历过程所走过的边的总次数小于3n，时间复杂度为O(n)。

图片来源



详细理解：考虑代码中的两个while。

a) 首先考虑外层while：

对于有左子树(也可以说有左子节点)的节点x，外层while会对其判断两次，第一次判断发生在初次遍历到x时，随后寻找x的predecessor(内层while，后述)后向左推进，第二次判断发生在x的左子树遍历完成后根据线索返回到x的位置，随后再次寻找x的predecessor，因为此时的predecessor.right != null，于是遍历向右(x.right)推进。对于没有左子树的节点y (y是具有右子树也可以说具有右子节点的节点或叶子节点)，外层while只会对其判断一次(因为不会返回)，随即向右推进，因此n个节点执行外层while的次数小于2n。

b) 其次考虑内层while：

内层while用于寻找上述x的predecessor，代码片段如下。只要x的左子节点xl不具有右子节点xlr，就不会进入内层while。此时x的predecessor就是x.left。如果xl具有xlr，则进入while，向右循环考察是否有右子节点。也就是说，在寻找x的predecessor过程中， 进入内层while的判断次数等于x的左子节点的右子树中右子节点的个数。对于整棵树来说，进入内层while的次数等于这棵树中z节点的个数的两倍(每一个predecessor都会被寻找两次)，z节点是某个节点x的左儿子y的右儿子链上的节点。于是，遍历整棵树，进入内层while的总次数小于2n。

```
predecessor = root.left;
while(predecessor.right != null && predecessor.right != root) {
    predecessor = predecessor.right;
}
```

综合a和b的分析，时间复杂度计算如下：

设进入外层while的总次数为m1(m1 < 2n)，除去内层while之后外层while内的语句数量是一个较小的常数k1，而进入内层while的总次数为m2(m2 < 2n)，内层while内的语句数量是k2(实际上k2=1)。程序整体执行总语句数为`k1*m1 + k2*m2 < (2k1+2)*n`

最终时间复杂度为O(n)。

空间复杂度：O(1)

该遍历算法无栈，只用到了predecessor变量。赋予线索时利用了叶子结点的右空指针，且在线索利用完成后(返回后)及时解除了该线索(`predecessor.right = null;`)，故空间复杂度为O(1)。

代码

```java
public List<Integer> inorderTraversalMorris(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if(root == null) {
            return res;
        }
        TreeNode predecessor = null;

        while(root != null) {
            // 当root有左子节点时
            if(root.left != null) {
                // 找到root左子树中在中序遍历中最后的节点predecessor,
                // 也就是root和它的左子树一起完成中序遍历后，root的前驱。
                // 寻找它的目的是之后不断往left方向推进root，到达最左子节点(叶子节点)后,
                // 能够返回，实现这个返回操作看后述。
                predecessor = root.left;
                // 与上predecessor.right != root条件的原因:
                // 见后续if中的内容，在第一次找到当前root的predecessor时,
                // 执行了predecessor.right = right。加入该条件,
                // 则当前root是完成其左子树遍历后后返回的root时，即此时有predecessor == root,
                // 不执行predecessor = predecessor.right;
                while(predecessor.right != null && predecessor.right != root) {
                    predecessor = predecessor.right;
                }
                // 最初找到的root的predecessor为叶子结点，必有predecessor.right == null
                if(predecessor.right == null) {
                    // 将它的右子节点赋为root
                    // 目的是root不断往left方向推进到最左子节点(叶子节点)后,
                    // 能够返回到此时的root的位置(可以称作返回的线索)。
                    predecessor.right = root;
                    // 向left方向推进root
                    root = root.left;
                }
                // predecessor != null只可能是它在上述if中执行过predecessor.right = root;
                // 这说明此时的root已经遍历了它的左子树，是通过本方法底部else里的
                // root = root.right回到当前root位置的。于是:
                // 首先断开此处predecessor的链接: predecessor.right = null
                // 将当前的root.val加入结果: res.add(root.val)
                // 然后继续向right方向前进: root = root.right
                else {
                    // 此线索已完成使命，解除该线索
                    predecessor.right = null;
                    res.add(root.val);
                    root = root.right;
                }
            }
```

```
            // 当root.left == null时，说明root无左子树，将root加入到结果中。
            // 然后往right方向遍历，此时的root节点(记做p)有两种情况：
            // 情况1：p是非叶子节点，直接移动到right，即root = root.right
            // 情况2：p是叶子节点：
            // 要能够回到当初以p为predecessor节点时的根(r)，
            // 因为在r第一次找到p为其predecessor时，执行过p.right = r，所以当前的p的右子节点是r
            // 则让root = p.right即可，即root = root.right。
            else {
                res.add(root.val);
                root = root.right;
            }
        }
        return res;
    }
```

去掉注释后的代码

```
public List<Integer> inorderTraversalMorris(TreeNode root) {
    List<Integer> res = new ArrayList<>();
    if(root == null) {
            return res;
    }
    TreeNode predecessor = null;
    while(root != null) {
        if(root.left != null) {
            predecessor = root.left;
            while(predecessor.right != null && predecessor.right != root) {
                predecessor = predecessor.right;
            }
            if(predecessor.right == null) {
                predecessor.right = root;
                root = root.left;
            }
            else {
                predecessor.right = null;
                res.add(root.val);
                root = root.right;
            }
        }
        else {
            res.add(root.val);
            root = root.right;
        }
    }
    return res;
}
```

参考

[1] leetcode-94

[2] Threaded binary tree

[3] Morris Traversal方法遍历二叉树（非递归，不用栈，O(1)空间

[4] How is the time complexity of Morris Traversal o(n)?