树的遍历

阅读更多

1 前言

本篇博客将以不同的角度带你体验二叉树的花式遍历

2 节点定义

```
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode parent;

    TreeNode(int x) {
        val = x;
    }
}
```

3 先序遍历

LeetCode：144

3.1 递归

先访问当前节点，然后递归左子树，再递归右子树

```
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> preorderTraversal(TreeNode root) {
        visitedList = new ArrayList<Integer>();

        helper(root);

        return visitedList;
    }

    private void helper(TreeNode root) {
        if (root != null) {
            visit(root);
            helper(root.left);
            helper(root.right);
        }
    }

    private void visit(TreeNode root) {
        visitedList.add(root.val);
    }
}
```

3.2 栈

对于某个节点

1. 沿着左孩子往下走依次访问经过的节点，该过程的所有节点会进栈
2. 当前节点为`null`，意味着遍历完毕或者说该访问该节点的右子树了

```
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> preorderTraversal(TreeNode root) {
        visitedList = new ArrayList<Integer>();

        LinkedList<TreeNode> stack = new LinkedList<TreeNode>();

        TreeNode cur = root;

        while (cur != null || !stack.isEmpty()) {
            while (cur != null) {
                visit(cur);
                stack.push(cur);
                cur = cur.left;
            }

            if (!stack.isEmpty()) {
                TreeNode top = stack.pop();

                cur = top.right;
            }
        }

        return visitedList;
    }

    private void visit(TreeNode root) {
        visitedList.add(root.val);
    }
}
```

或者

```
public class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        visitedList = new ArrayList<>();

        LinkedList<TreeNode> stack = new LinkedList<>();

        if (root != null) {
            stack.push(root);
        }

        while (!stack.isEmpty()) {
            TreeNode top = stack.pop();

            visit(top);

            if (top.right != null) {
                stack.push(top.right);
            }

            if (top.left != null) {
                stack.push(top.left);
            }
        }

        return visitedList;
    }

    private void visit(TreeNode root) {
        visitedList.add(root.val);
    }
}
```

3.3 非栈非递归

这个方法本质上与栈差不多，只是利用的空间更少了，但是要求TreeNode的定义必须有parent字段，而栈的方法不需要parent字段

```
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> preorderTraversal(TreeNode root) {
        visitedList = new ArrayList<Integer>();

        TreeNode cur = root;

        TreeNode pre = null;

        while (cur != null) {
            pre = cur;
            if (pre == cur.parent) {
                visit(cur);
                if (cur.left != null) {
                    cur = cur.left;
                } else if (cur.right != null) {
                    cur = cur.right;
                } else {
                    cur = cur.parent;
                }
            } else if (pre == cur.left) {
                if (cur.right != null) {
                    cur = cur.right;
                } else {
                    cur = cur.parent;
                }
            } else {
                cur = cur.parent;
            }
        }

        return visitedList;
    }

    private void visit(TreeNode root) {
        visitedList.add(root.val);
    }
}
```

4 中序遍历

LeetCode：94

4.1 递归

先递归左子树，访问当前节点，再递归右子树

```
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> inorderTraversal(TreeNode root) {
        visitedList = new ArrayList<Integer>();

        helper(root);

        return visitedList;
    }

    private void helper(TreeNode root) {
        if (root != null) {
            helper(root.left);
            visit(root);
            helper(root.right);
        }
    }

    private void visit(TreeNode root) {
        visitedList.add(root.val);
    }
}
```

4.2 栈

对于某个节点

1. 首先沿着左孩子节点一直到叶节点，该过程的所有节点会进栈
2. 当前节点为null，意味着遍历完毕或者说该访问栈中的元素了

```java
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> inorderTraversal(TreeNode root) {
        visitedList = new ArrayList<Integer>();

        LinkedList<TreeNode> stack = new LinkedList<TreeNode>();

        TreeNode cur = root;

        while (cur != null || !stack.isEmpty()) {
            while (cur != null) {
                stack.push(cur);
                cur = cur.left;
            }

            if (!stack.isEmpty()) {
                TreeNode top = stack.pop();

                visit(top);

                cur = top.right;
            }
        }

        return visitedList;
    }

    private void visit(TreeNode root) {
        visitedList.add(root.val);
    }
}
```

或者

```java
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> inorderTraversal(TreeNode root) {
        visitedList = new ArrayList<>();

        LinkedList<TreeNode> stack = new LinkedList<>();

        TreeNode iter = root;
        while (iter != null) {
            stack.push(iter);
            iter = iter.left;
        }

        while (!stack.isEmpty()) {
            TreeNode top = stack.pop();

            visit(top);

            if (top.right != null) {
                iter = top.right;
                while (iter != null) {
                    stack.push(iter);
                    iter = iter.left;
                }
            }
        }

        return visitedList;
    }

    private void visit(TreeNode root) {
        visitedList.add(root.val);
    }
}
```

4.3 非栈非递归

这个方法本质上与栈差不多，只是利用的空间更少了，但是要求TreeNode的定义必须有parent字段，而栈的方法不需要parent字段

```
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> inorderTraversal(TreeNode root) {
        visitedList = new ArrayList<Integer>();

        TreeNode cur = root;

        TreeNode pre = null;

        while (cur != null) {
            pre = cur;
            if (pre == cur.parent) {
                if (cur.left != null) {
                    cur = cur.left;
                } else if (cur.right != null) {
                    visit(cur);
                    cur = cur.right;
                } else {
                    visit(cur);
                    cur = cur.parent;
                }
            } else if (pre == cur.left) {
                visit(cur);
                if (cur.right != null) {
                    cur = cur.right;
                } else {
                    cur = cur.parent;
                }
            } else {
                cur = cur.parent;
            }
        }

        return visitedList;
    }

    private void visit(TreeNode root) {
        visitedList.add(root.val);
    }
}
```

5 后续遍历

LeetCode：145

5.1 递归

先递归左子树，然后递归右子树，再访问当前节点

```
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> postorderTraversal(TreeNode root) {
        visitedList = new ArrayList<Integer>();

        helper(root);

        return visitedList;
    }

    private void helper(TreeNode root) {
        if (root != null) {
            helper(root.left);
            helper(root.right);
            visit(root);
        }
    }

    private void visit(TreeNode root) {
        visitedList.add(root.val);
    }
}
```

5.2 栈1

由于后续遍历是：左子树-右子树-当前节点。反过来看就是，当前节点-右子树-左子树，这是相反方向的先序遍历

对于某个节点

1. 沿着右孩子往下走依次访问(将元素添加到访问List的头部即可，即做一个逆序操作)经过的节点，该过程的所有节点会进栈
2. 当前节点为null，意味着遍历完毕或者说该访问该节点的左子树了

```
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> postorderTraversal(TreeNode root) {
        visitedList = new LinkedList<Integer>();//这里用ListedList作为实现，因为要在头

        LinkedList<TreeNode> stack = new LinkedList<TreeNode>();

        TreeNode cur = root;

        while (cur != null || !stack.isEmpty()) {
            while (cur != null) {
                visit(cur);
                stack.push(cur);
                cur = cur.right;
            }

            if (!stack.isEmpty()) {
                TreeNode top = stack.pop();
                cur = top.left;
            }
        }

        return visitedList;
    }

    private void visit(TreeNode root) {
        visitedList.add(0,root.val);
    }
}
```

或者

```
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> postorderTraversal(TreeNode root) {
        visitedList = new LinkedList<Integer>();

        LinkedList<TreeNode> stack = new LinkedList<>();

        if (root != null) {
            stack.push(root);
        }

        while (!stack.isEmpty()) {
            TreeNode top = stack.pop();

            visit(top);

            if (top.left != null) {
                stack.push(top.left);
            }

            if (top.right != null) {
                stack.push(top.right);
            }
        }

        return visitedList;
    }

    private void visit(TreeNode root) {
        visitedList.add(0, root.val);
    }
}
```

5.3 栈2

另一种栈的思路

1. 首先将根节点入栈
2. 访问栈顶节点，如果栈顶节点没有孩子，或者栈顶节点是pre的父节点(说明回溯上去了)，此时访问该节点，并更新pre
3. 否则若右孩子不为空，则右孩子入栈，左孩子不为空，则左孩子入栈(因为先访问的节点要后入栈，因此是先右后左的顺序)

```
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> postorderTraversal(TreeNode root) {
        visitedList = new ArrayList<>();

        LinkedList<TreeNode> stack = new LinkedList<>();

        if (root != null) {
            stack.push(root);
        }

        TreeNode pre = null;

        while (!stack.isEmpty()) {
            TreeNode peek = stack.peek();

            if (peek.left == null && peek.right == null
                    || (pre != null && (peek.left == pre || peek.right == pre))) {
                stack.pop();
                visit(peek);
                pre = peek;
            } else {
                if (peek.right != null) {
                    stack.push(peek.right);
                }
                if (peek.left != null) {
                    stack.push(peek.left);
                }
            }
        }

        return visitedList;
    }

    private void visit(TreeNode root) {
        visitedList.add(root.val);
    }
}
```

5.4 栈3

对于某个节点

1. 首先沿着左孩子节点一直到叶节点，该过程的所有节点会进栈，并且记录入栈次数为1
2. 当前节点为`null`，意味着遍历完毕或者说该访问栈中的元素了，取出栈顶元素，如果该元素入栈2次，那么访问该元素，否则重新入栈，并递增入栈计数值

```
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> postorderTraversal(TreeNode root) {
        visitedList = new ArrayList<Integer>();

        LinkedList<TreeNode> stack = new LinkedList<TreeNode>();

        Map<TreeNode, Integer> count = new HashMap<TreeNode, Integer>();

        TreeNode cur = root;

        while (cur != null || !stack.isEmpty()) {
            while (cur != null) {
                stack.push(cur);
                count.put(cur, 1);
                cur = cur.left;
            }

            if (!stack.isEmpty()) {
                TreeNode top = stack.pop();

                if (count.get(top) == 2) {
                    visit(top);
                } else {
                    count.put(top, 2);
                    stack.push(top);
                    cur = top.right;
                }
            }
        }

        return visitedList;
    }

    private void visit(TreeNode root) {
        visitedList.add(root.val);
    }
}
```

或者

```
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> postorderTraversal(TreeNode root) {
        visitedList = new ArrayList<>();

        LinkedList<TreeNode> stack = new LinkedList<>();
        Map<TreeNode, Integer> count = new HashMap<>();

        if (root != null) {
            stack.push(root);
            count.put(root, 1);
        }

        while (!stack.isEmpty()) {
            TreeNode top = stack.pop();

            if (count.get(top) == 1) {
                stack.push(top);
                count.put(top, 2);
                if (top.right != null) {
                    stack.push(top.right);
                    count.put(top.right, 1);
                }

                if (top.left != null) {
                    stack.push(top.left);
                    count.put(top.left, 1);
                }
            } else {
                visit(top);
            }
        }

        return visitedList;
    }

    private void visit(TreeNode root) {
        visitedList.add(root.val);
    }
}
```

5.5 非栈非递归

这个方法本质上与栈差不多，只是利用的空间更少了，但是要求TreeNode的定义必须有parent字段，而栈的方法不需要parent字段

```
public class Solution {
    private List<Integer> visitedList;

    public List<Integer> postorderTraversal(TreeNode root) {
        visitedList = new ArrayList<Integer>();

        TreeNode cur = root;

        TreeNode pre = null;

        while (cur != null) {
            pre = cur;
            if (pre == cur.parent) {
                if (cur.left != null) {
                    cur = cur.left;
                } else if (cur.right != null) {
                    cur = cur.right;
                } else {
                    visit(cur);
                    cur = cur.parent;
                }
            } else if (pre == cur.left) {
                if (cur.right != null) {
                    cur = cur.right;
                } else {
                    visit(cur);
                    cur = cur.parent;
                }
            } else {
                visit(cur);
                cur = cur.parent;
            }
        }

        return visitedList;
    }

    private void visit(TreeNode root) {
        visitedList.add(root.val);
    }
}
```

6 层序遍历

6.1 队列

遍历每层前先记录队列的大小，该大小就是该层元素的个数，并且依次将左右孩子入队列

```java
public class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> visitedLevel = new ArrayList<List<Integer>>();

        Queue<TreeNode> queue = new LinkedList<TreeNode>();

        if (root != null)
            queue.offer(root);

        while (!queue.isEmpty()) {
            List<Integer> curLevel = new ArrayList<Integer>();
            int count = queue.size();

            while (--count >= 0) {
                TreeNode cur = queue.poll();
                if (cur.left != null)
                    queue.offer(cur.left);
                if (cur.right != null)
                    queue.offer(cur.right);
                curLevel.add(cur.val);
            }

            visitedLevel.add(curLevel);
        }

        return visitedLevel;
    }
}
```