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

```
1  public class TreeNode {
2      int val;
3      TreeNode left;
4      TreeNode right;
5      TreeNode parent;
6
7      TreeNode(int x) {
8          val = x;
9      }
10 }
```

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

```
1  public class Solution {
2      private List<Integer> visitedList;
3
4      public List<Integer> preorderTraversal(TreeNode root) {
5          visitedList = new ArrayList<Integer>();
6
7          helper(root);
8
9          return visitedList;
10     }
11
12     private void helper(TreeNode root) {
13         if (root != null) {
14             visit(root);
15             helper(root.left);
16             helper(root.right);
17         }
18     }
19
20     private void visit(TreeNode root) {
21         visitedList.add(root.val);
```

```
22     }
23 }
```

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

```
1  public class Solution {
2      private List<Integer> visitedList;
3
4      public List<Integer> preorderTraversal(TreeNode root) {
5          visitedList = new ArrayList<Integer>();
6
7          LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
8
9          TreeNode cur = root;
10
11         while (cur != null || !stack.isEmpty()) {
12             while (cur != null) {
13                 visit(cur);
14                 stack.push(cur);
15                 cur = cur.left;
16             }
17
18             if (!stack.isEmpty()) {
19                 TreeNode top = stack.pop();
20
21                 cur = top.right;
22             }
23         }
24
25         return visitedList;
26     }
27
28     private void visit(TreeNode root) {
29         visitedList.add(root.val);
30     }
31 }
```

或者

```
1  public class Solution {
2      public List<Integer> preorderTraversal(TreeNode root) {
3          visitedList = new ArrayList<>();
4
5          LinkedList<TreeNode> stack = new LinkedList<>();
6
7          if (root != null) {
8              stack.push(root);
```

```
9          }
10
11         while (!stack.isEmpty()) {
12             TreeNode top = stack.pop();
13
14             visit(top);
15
16             if (top.right != null) {
17                 stack.push(top.right);
18             }
19
20             if (top.left != null) {
21                 stack.push(top.left);
22             }
23         }
24
25         return visitedList;
26     }
27
28     private void visit(TreeNode root) {
29         visitedList.add(root.val);
30     }
31 }
```

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

```
1  public class Solution {
2      private List<Integer> visitedList;
3
4      public List<Integer> preorderTraversal(TreeNode root) {
5          visitedList = new ArrayList<Integer>();
6
7          TreeNode cur = root;
8
9          TreeNode pre = null;
10
11         while (cur != null) {
12             pre = cur;
13             if (pre == cur.parent) {
14                 visit(cur);
15                 if (cur.left != null) {
16                     cur = cur.left;
17                 } else if (cur.right != null) {
18                     cur = cur.right;
19                 } else {
20                     cur = cur.parent;
21                 }
22             } else if (pre == cur.left) {
23                 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

```
1  public class Solution {
2      private List<Integer> visitedList;
3
4      public List<Integer> inorderTraversal(TreeNode root) {
5          visitedList = new ArrayList<Integer>();
6
7          LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
8
9          TreeNode cur = root;
10
11         while (cur != null || !stack.isEmpty()) {
12             while (cur != null) {
13                 stack.push(cur);
14                 cur = cur.left;
15             }
16
17             if (!stack.isEmpty()) {
18                 TreeNode top = stack.pop();
19
20                 visit(top);
21
22                 cur = top.right;
23             }
24         }
25
26         return visitedList;
27     }
28
29     private void visit(TreeNode root) {
30         visitedList.add(root.val);
31     }
32 }
```

或者

```
1  public class Solution {
2      private List<Integer> visitedList;
3
4      public List<Integer> inorderTraversal(TreeNode root) {
5          visitedList = new ArrayList<>();
6
7          LinkedList<TreeNode> stack = new LinkedList<>();
8
9          TreeNode iter = root;
10         while (iter != null) {
11             stack.push(iter);
12             iter = iter.left;
13         }
14
```

```
15         while (!stack.isEmpty()) {
16             TreeNode top = stack.pop();
17 
18             visit(top);
19 
20             if (top.right != null) {
21                 iter = top.right;
22                 while (iter != null) {
23                     stack.push(iter);
24                     iter = iter.left;
25                 }
26             }
27         }
28 
29         return visitedList;
30     }
31 
32     private void visit(TreeNode root) {
33         visitedList.add(root.val);
34     }
35 }
```

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

```
1  public class Solution {
2      private List<Integer> visitedList;
3 
4      public List<Integer> inorderTraversal(TreeNode root) {
5          visitedList = new ArrayList<Integer>();
6 
7          TreeNode cur = root;
8 
9          TreeNode pre = null;
10 
11         while (cur != null) {
12             pre = cur;
13             if (pre == cur.parent) {
14                 if (cur.left != null) {
15                     cur = cur.left;
16                 } else if (cur.right != null) {
17                     visit(cur);
18                     cur = cur.right;
19                 } else {
20                     visit(cur);
21                     cur = cur.parent;
22                 }
23             } else if (pre == cur.left) {
24                 visit(cur);
25                 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

```
1  public class Solution {
2      private List<Integer> visitedList;
3
4      public List<Integer> postorderTraversal(TreeNode root) {
5          visitedList = new LinkedList<Integer>();//这里用ListedList作为实现，因为要在头
6
7          LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
8
9          TreeNode cur = root;
10
11         while (cur != null || !stack.isEmpty()) {
12             while (cur != null) {
13                 visit(cur);
14                 stack.push(cur);
15                 cur = cur.right;
16             }
17
18             if (!stack.isEmpty()) {
19                 TreeNode top = stack.pop();
20                 cur = top.left;
21             }
22         }
23
24         return visitedList;
25     }
26
27     private void visit(TreeNode root) {
28         visitedList.add(0,root.val);
29     }
30 }
```

或者

```
1  public class Solution {
2      private List<Integer> visitedList;
3
4      public List<Integer> postorderTraversal(TreeNode root) {
5          visitedList = new LinkedList<Integer>();
6
7          LinkedList<TreeNode> stack = new LinkedList<>();
8
9          if (root != null) {
10             stack.push(root);
11         }
12
13         while (!stack.isEmpty()) {
```

```
14              TreeNode top = stack.pop();
15
16              visit(top);
17
18              if (top.left != null) {
19                  stack.push(top.left);
20              }
21
22              if (top.right != null) {
23                  stack.push(top.right);
24              }
25          }
26
27          return visitedList;
28      }
29
30      private void visit(TreeNode root) {
31          visitedList.add(0, root.val);
32      }
33  }
```

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)

```
1   public class Solution {
2       private List<Integer> visitedList;
3
4       public List<Integer> postorderTraversal(TreeNode root) {
5           visitedList = new ArrayList<>();
6
7           LinkedList<TreeNode> stack = new LinkedList<>();
8
9           if (root != null) {
10              stack.push(root);
11          }
12
13          TreeNode pre = null;
14
15          while (!stack.isEmpty()) {
16              TreeNode peek = stack.peek();
17
18              if (peek.left == null && peek.right == null
19                      || (pre != null && (peek.left == pre || peek.right == pre))) {
20                  stack.pop();
21                  visit(peek);
```

```
22                 pre = peek;
23             } else {
24                 if (peek.right != null) {
25                     stack.push(peek.right);
26                 }
27                 if (peek.left != null) {
28                     stack.push(peek.left);
29                 }
30             }
31         }
32 
33         return visitedList;
34     }
35 
36     private void visit(TreeNode root) {
37         visitedList.add(root.val);
38     }
39 }
```

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

```
1  public class Solution {
2      private List<Integer> visitedList;
3 
4      public List<Integer> postorderTraversal(TreeNode root) {
5          visitedList = new ArrayList<Integer>();
6 
7          LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
8 
9          Map<TreeNode, Integer> count = new HashMap<TreeNode, Integer>();
10 
11         TreeNode cur = root;
12 
13         while (cur != null || !stack.isEmpty()) {
14             while (cur != null) {
15                 stack.push(cur);
16                 count.put(cur, 1);
17                 cur = cur.left;
18             }
19 
20             if (!stack.isEmpty()) {
21                 TreeNode top = stack.pop();
22 
23                 if (count.get(top) == 2) {
24                     visit(top);
25                 } 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;
    }
```

```
37
38     private void visit(TreeNode root) {
39         visitedList.add(root.val);
40     }
41 }
```

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

```
1  public class Solution {
2      private List<Integer> visitedList;
3
4      public List<Integer> postorderTraversal(TreeNode root) {
5          visitedList = new ArrayList<Integer>();
6
7          TreeNode cur = root;
8
9          TreeNode pre = null;
10
11         while (cur != null) {
12             pre = cur;
13             if (pre == cur.parent) {
14                 if (cur.left != null) {
15                     cur = cur.left;
16                 } else if (cur.right != null) {
17                     cur = cur.right;
18                 } else {
19                     visit(cur);
20                     cur = cur.parent;
21                 }
22             } else if (pre == cur.left) {
23                 if (cur.right != null) {
24                     cur = cur.right;
25                 } else {
26                     visit(cur);
27                     cur = cur.parent;
28                 }
29             } else {
30                 visit(cur);
31                 cur = cur.parent;
32             }
33         }
34
35         return visitedList;
36     }
37
38     private void visit(TreeNode root) {
39         visitedList.add(root.val);
40     }
41 }
```

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

```
1  public class Solution {
2      public List<List<Integer>> levelOrder(TreeNode root) {
3          List<List<Integer>> visitedLevel = new ArrayList<List<Integer>>();
4
5          Queue<TreeNode> queue = new LinkedList<TreeNode>();
6
7          if (root != null)
8              queue.offer(root);
9
10         while (!queue.isEmpty()) {
11             List<Integer> curLevel = new ArrayList<Integer>();
12             int count = queue.size();
13
14             while (--count >= 0) {
15                 TreeNode cur = queue.poll();
16                 if (cur.left != null)
17                     queue.offer(cur.left);
18                 if (cur.right != null)
19                     queue.offer(cur.right);
20                 curLevel.add(cur.val);
21             }
22
23             visitedLevel.add(curLevel);
24         }
25
26         return visitedLevel;
27     }
28 }
```