二叉树在面试实战中，花样非常多。本节只是个开头，在后面几个专题、包括最后的大厂真题实战环节中，我们都不会停止对二叉树相关考点的学习和探讨。

在本节，有以下三个命题方向需要大家重点掌握：

- 迭代法实现二叉树的先、中、后序遍历
- 二叉树层序遍历的衍生问题
- 翻转二叉树

这三个方向对应的考题都比较经典。与此同时，解决这些问题涉及到的思路和编码细节，也会成为各位日后解决更加复杂的问题的基石。因此，虽然本节篇幅略长，但还是希望各位能够倾注耐心，给自己充分的时间去理解和消化这些知识。

“遍历三兄弟”的迭代实现

经过第5节的学习，相信各位已经将二叉树先、中、后序遍历的递归实现吃得透透的了。在使用递归实现遍历的过程中，我们明显察觉到，“遍历三兄弟”的编码实现也宛如孪生兄弟一样，彼此之间只有代码顺序上的不同，整体内容基本是一致的。这正是递归思想的一个重要的优点——简单。

这里的“简单”并不是说学起来简单。相反，结合笔者早期的读者调研来看，大部分同学都认为递归学起来让人很难受（这也是正常的）。
初学递归的人排斥递归，大部分是出于对“函数调用自身”这种骚操作的不适应。但只要你能克服这种不适应，并且通过反复的演练去吸收这种解题方法，你就会发现递归真的是个好东西。因为通过使用递归，我们可以把原本复杂的东西，拆解成非常简单的、符合人类惯用脑回路的逻辑。

这样说可能还是有点抽象，不过没关系，接下来我会讲解“遍历三兄弟”对应的迭代解法。等我们学完这坨东西之后，心怀疑惑的同学不妨拿迭代法的代码和第5节中递归法的代码比较一下，相信你会毫不犹豫地回头对递归说上一句“真香！”。

从先序遍历说起

> 题目描述：给定一个二叉树，返回它的前序（先序）遍历序列。

> 示例: 输入: [1,null,2,3]

```
1
 \
  2
 /
3
```

> 输出: [1,2,3]
> 进阶: 递归算法很简单，你可以通过迭代算法完成吗?

思路分析

注意最后那一行小字：“递归算法很简单，你可以通过迭代算法完成吗？”，对递归算法有疑问的同学，趁这个机会赶紧复习下第五小节，本节我们只讲迭代法。
前面两个小节，我们一直在强调，递归和栈有着脱不开的干系。当一道本可以用递归做出来的题，突然不许你用递归了，此时我们本能的反应，就应该是往栈上想。

在基于栈来解决掉这个题之前，我要先跟平时用 `leetcode` 刷题的各位强调一个常识。

现在大家回头看这道题目给我们的**输入**和**输出**：输入看似是一个数组，实则不是。大家谨记，二叉树题目的输入只要没有额外强调，那么一般来说它都是基于这样的一个对象结构嵌套而来的：

```
function TreeNode(val) {
    this.val = val;
    this.left = this.right = null;
}
```

比如这样：

```
const root = {
  val: "A",
  left: {
    val: "B",
    left: {
      val: "D"
    },
    right: {
      val: "E"
    }
  },
  right: {
    val: "C",
    right: {
      val: "F"
```

```
      }
    }
  };
```

话说回来，为啥题上给的不是对象，而是这样的一个数组呢：

```
[1,null,2,3]
```

这其实是一种简化的写法，性质跟咱们写伪代码差不多。它的作用主要是描述二叉树的值，至于二叉树的结构，我们以题中给出的树形结构为准：

```
1
  \
   2
  /
3
```

OK，了解了输入内容，现在再来看输出：

```
[1,2,3]
```

这里这个输出就简单多了，它是一个**真数组**。为什么可以判断它是一个真数组呢？因为题目中要求你输出的是一个遍历序列，而不是一个**二叉树**。因此大家最后需要塞入结果数组的不是结点对象，而是结点的值。

> 注意： 以上的出参入参规律，是针对 leetcode 及其周边 OJ 来说的。OJ 中这样编写题目描述，是情理之中，因为 OJ 本身是支持多语言的，它只能对最通用的一部分信息进行透出。在面试场景下，不排除一些公司可能会贴心地把 JS 版本的出参和入参给出来，但更多的还是会直接复制粘贴 leetcode 或者其它一些算法书中的原题。如果你对题目的出参和入参有疑问，请大胆地对面试官说出你的困惑——没有一个正常面试官会在题目描述上为难你， 他比你更急切地想看到你刷刷写代码的英姿。

回到题目上来。我们接着栈往下说，题目中的出参是一个数组，大家仔细看这个数组，它像不像是一个栈的出栈序列？实际上，做这道题的一个根本思路，就是通过**合理地安排入栈和出栈的时机、使栈的出栈序列符合二叉树的前序遍历规则**。

前序遍历的规则是，先遍历根结点、然后遍历左孩子、最后遍历右孩子——这正是我们所期望的出栈序列。按道理，入栈序列和出栈序列相反，我们似乎应该按照 右->左->根 这样的顺序将结点入栈。不过需

要注意的是，我们遍历的起点就是根结点，难道我们要假装没看到这个根结点、一鼓作气找到最右侧结点之后才开始进行入栈操作吗？答案当然是否定的，我们的出入栈顺序应该是这样的：

1. 将根结点入栈
2. 取出栈顶结点，将结点值 `push` 进结果数组
3. 若栈顶结点有右孩子，则将右孩子入栈
4. 若栈顶结点有左孩子，则将左孩子入栈

这整个过程，本质上是将当前子树的根结点入栈、出栈，随后再将其对应左右子树入栈、出栈的过程。

重复 2、3、4 步骤，直至栈空，我们就能得到一个先序遍历序列。

编码实现

```
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
const preorderTraversal = function(root) {
  // 定义结果数组
  const res = []
  // 处理边界条件
  if(!root) {
      return res
  }
  // 初始化栈结构
  const stack = []
  // 首先将根结点入栈
  stack.push(root)
  // 若栈不为空，则重复出栈、入栈操作
  while(stack.length) {
      // 将栈顶结点记为当前结点
      const cur = stack.pop()
      // 当前结点就是当前子树的根结点，把这个结点放在结果数组的尾部
      res.push(cur.val)
      // 若当前子树根结点有右孩子，则将右孩子入栈
      if(cur.right) {
          stack.push(cur.right)
```

```
        }
        // 若当前子树根结点有左孩子，则将左孩子入栈
        if(cur.left) {
            stack.push(cur.left)
        }
    }
    // 返回结果数组
    return res
};
```

异曲同工的后序遍历迭代实现

思路分析

后序遍历的出栈序列，按照规则应该是 `左 -> 右 -> 根` 。这个顺序相对于先序遍历，最明显的变化就是根结点的位置从第一个变成了倒数第一个。
如何做到这一点呢？与其从 `stack` 这个栈结构上入手，不如从 `res` 结果数组上入手：我们可以直接把 `pop` 出来的当前结点 `unshift` 进 `res` 的头部，改造后的代码会变成这样：

```
while(stack.length) {
    // 将栈顶结点记为当前结点
    const cur = stack.pop()
    // 当前结点就是当前子树的根结点，把这个结点放在结果数组的头部
    res.unshift(cur.val)
    // 若当前子树根结点有右孩子，则将右孩子入栈
    if(cur.right) {
        stack.push(cur.right)
    }
    // 若当前子树根结点有左孩子，则将左孩子入栈
    if(cur.left) {
        stack.push(cur.left)
    }
}
```

大家可以尝试在大脑里预判一下这个代码的执行顺序：由于我们填充 `res` 结果数组的顺序是从后往前填充（每次增加一个头部元素），因此先出栈的结点反而会位于 `res` 数组相对靠后的位置。出栈的顺序是 `当前结点 -> 当前结点的左孩子 -> 当前结点的右孩子` ，其对应的 `res` 序列顺序就是 `右 -> 左 -> 根` 。这样

一来， 根结点就成功地被我们转移到了遍历序列的最末尾。
现在唯一让人看不顺眼的只剩下这个右孩子和左孩子的顺序了，这两个孩子结点进入结果数组的顺序与其被 `pop` 出栈的顺序是一致的，而出栈顺序又完全由入栈顺序决定，因此只需要相应地调整两个结点的入栈顺序就好了：

```
// 若当前子树根结点有左孩子，则将左孩子入栈
if(cur.left) {
  stack.push(cur.left)
}
// 若当前子树根结点有右孩子，则将右孩子入栈
if(cur.right) {
  stack.push(cur.right)
}
```

如此一来，右孩子就会相对于左孩子优先出栈，进而被放在 `res` 结果数组相对靠后的位置， `左 -> 右 ->根` 的排序规则就稳稳地实现出来了。

我们把以上两个改造点结合一下，就有了以下代码：

编码实现

```
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
const postorderTraversal = function(root) {
  // 定义结果数组
  const res = []
  // 处理边界条件
  if(!root) {
      return res
  }
  // 初始化栈结构
  const stack = []
  // 首先将根结点入栈
  stack.push(root)
  // 若栈不为空，则重复出栈、入栈操作
  while(stack.length) {
```

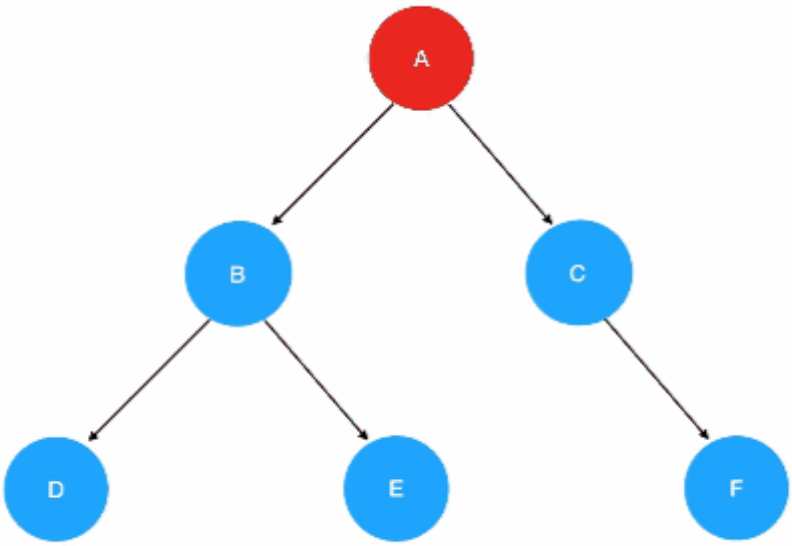
```
        // 将栈顶结点记为当前结点
        const cur = stack.pop()
        // 当前结点就是当前子树的根结点，把这个结点放在结果数组的头部
        res.unshift(cur.val)
        // 若当前子树根结点有左孩子，则将左孩子入栈
        if(cur.left) {
          stack.push(cur.left)
        }
        // 若当前子树根结点有右孩子，则将右孩子入栈
        if(cur.right) {
          stack.push(cur.right)
        }
    }
    // 返回结果数组
    return res
};
```

思路清奇的中序遍历迭代实现

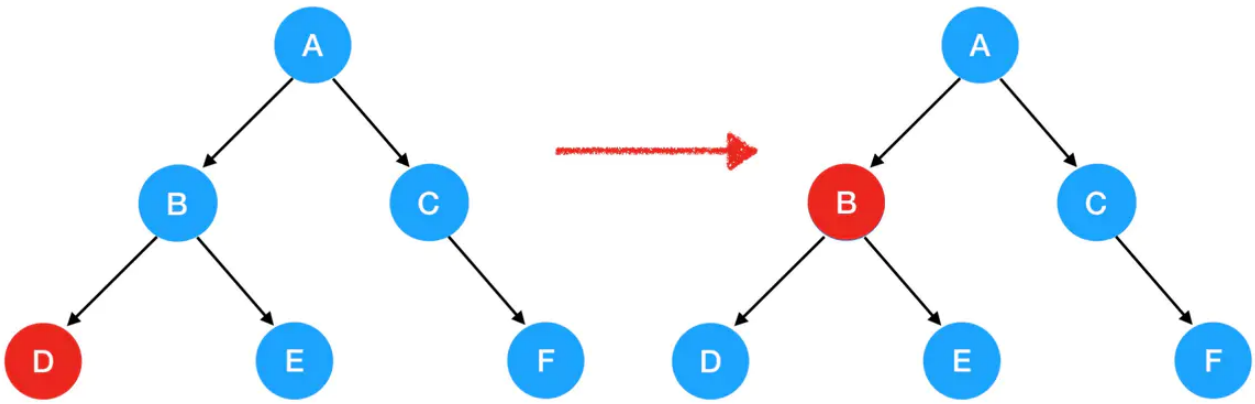
思路分析

经过上面的讲解，大家会发现先序遍历和后序遍历的编码实现其实是非常相似的，它们遵循的都是同一套基本框架。那么我们能否通过对这个基本框架进行微调、从而同样轻松地实现中序遍历呢？
答案是不能，为啥不能？因为先序遍历和后序遍历之所以可以用同一套代码框架来实现，本质上是因为两者的出栈、入栈逻辑差别不大——都是先处理根结点，然后处理孩子结点。而中序遍历中，根结点不再出现在遍历序列的边界、而是出现在遍历序列的中间。这就意味着无论如何我们不能再将根结点作为第一个被 `pop` 出来的元素来处理了——出栈的时机被改变了，这意味着入栈的逻辑也需要调整。这一次我们不能再通过对 `res` 动手脚来解决问题，而是需要和 `stack` 面对面 battle。

中序遍历的序列规则是 `左 -> 中 -> 右` ，这意味着我们必须首先定位到最左的叶子结点。在这个定位的过程中，必然会途径目标结点的父结点、爷爷结点和各种辈分的祖宗结点：



途径过的每一个结点，我们都要及时地把它入栈。这样当最左的叶子结点出栈时，第一个回溯到的就是它的父结点：



有了父结点，就不愁找不到兄弟结点，遍历结果就变得唾手可得了~
基于这个思路，我们来写代码：

编码实现

```
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
const inorderTraversal = function(root) {
```

```
    // 定义结果数组
    const res = []
    // 初始化栈结构
    const stack = []
    // 用一个 cur 结点充当游标
    let cur = root
    // 当 cur 不为空、或者 stack 不为空时，重复以下逻辑
    while(cur || stack.length) {
        // 这个 while 的作用是把寻找最左叶子结点的过程中，途径的所有结点都记录下来
        while(cur) {
            // 将途径的结点入栈
            stack.push(cur)
            // 继续搜索当前结点的左孩子
            cur = cur.left
        }
        // 取出栈顶元素
        cur = stack.pop()
        // 将栈顶元素入栈
        res.push(cur.val)
        // 尝试读取 cur 结点的右孩子
        cur = cur.right
    }
    // 返回结果数组
    return res
};
```

编码复盘

读完这段编码示范，一部分同学可能已经开始懵逼了：看着前面给出的思路分析，似乎完全写不出上面这样的代码啊！所以这段代码到底在干嘛？？？
如果你没有这样的困惑，说明你是一位悟性比较高的同学，可以直接跳过编码复盘部分往下读了（btw给你点个赞~）。
实不相瞒，如果你是初学者，这段代码可能确实需要大家在脑内反复运行、反复跑 demo 才能理解其中的逻辑。为了加快这个过程，我把其中看上去稍微拐弯抹角一点的逻辑摘出来，给大家点拨一下：

1. 两个 `while` ：内层的 `while` 的作用是在寻找最左叶子结点的过程中，把途径的所有结点都记录到 `stack` 里。记录工作完成后，才会走到外层 `while` 的剩余逻辑里——这部分逻辑的作用是从最左的叶子结点开始，一层层回溯遍历左孩子的父结点和右侧兄弟结点，进而完成整个中序遍历任务。
2. 外层 `while` 的两个条件： `cur` 的存在性和 `stack.length` 的存在性，各自是为了限制什么？

1. `stack.length` 的存在性比较好理解， `stack` 中存储的是没有被推入结果数组 `res` 的待遍历元素。只要 `stack` 不为空，就意味着遍历没有结束， 遍历动作需要继续重复。
2. `cur` 的存在性就比较有趣了。它对应以下几种情况：
    1. 初始态， `cur` 指向 `root` 结点，只要 `root` 不为空， `cur` 就不为空。此时判断了 `cur` 存在后，就会开始最左叶子结点的寻找之旅。这趟“一路向左”的旅途中， `cur` 始终指向当前遍历到的左孩子。
    2. 第一波内层 `while` 循环结束， `cur` 开始承担中序遍历的遍历游标职责。 `cur` 始终会指向当前栈的栈顶元素，也就是“一路向左”过程中途径的某个左孩子，然后将这个左孩子作为中序遍历的第一个结果元素纳入结果数组。假如这个左孩子是一个叶子结点，那么尝试取其右孩子时就只能取到 `null` ，这个 `null` 的存在，会导致内层循环 `while` 被跳过，接着就直接回溯到了这个左孩子的父结点，符合 `左->根` 的序列规则
    3. 假如当前取到的栈顶元素不是叶子结点，同时有一个右孩子，那么尝试取其右孩子时就会取到一个存在的结点。 `cur` 存在，于是进入内层 `while` 循环，重复“一路向左”的操作，去寻找这个右孩子对应的子树里最靠左的结点，然后去重复刚刚这个或回溯、或“一路向左”的过程。如果这个右孩子对应的子树里没有左孩子，那么跳出内层 `while` 循环之后，紧接着被纳入 `res` 结果数组的就是这个右孩子本身，符合 `根->右` 的序列规则

结合上面的分析，大家会不会觉得中序遍历迭代法的这一通操作非常奇妙呢？短短的几行代码，里面竟然藏着这么广阔的乾坤，牛x、牛x。
作为初学者，即便第一次写不出来上面的解法，也没什么好丧气的——大家谨记，关于二叉树的先、中、后序遍历，**你对自己的要求应该是能够默写，也就是说要对上面这些逻辑充分熟悉、深刻记忆。**
在熟悉和记忆的过程中，你会渐渐地对这些乍一看似乎很巧妙的操作产生一种“这也很自然嘛”的感觉，这种感觉就意味着你对这个思路的充分吸收。还是那句话，千万不要以为理解就是终点，你需要做的是记忆！记忆！理解是一种感觉，记忆却能保证你在做题时一秒钟映射到具体的套路和代码——只有靠自己的双手写出来的代码，才是最可靠的伙伴。

层序遍历的衍生问题

在 `DFS 和 BFS` 这一节，我们已经讲过了二叉树层序遍历的实现方法。层序遍历本本身难度不大，但一想到这是一个关键考点，出题这帮人就每天绞尽脑汁地想要把简单的问题复杂化。于是，就有了我们眼下这个命题方向——层序遍历的衍生问题。
对于这类问题，我们接下来会讲最有代表性的一道作为例题。各位只要能吃透这一道的基本思路，就能够轻松地在类似的变体中举一反三（例题请大家好好把握，在大厂真题训练环节，我会给出一道变体来检验各位的学习效果）。

> 题目描述：给你一个二叉树，请你返回其按 层序遍历 得到的节点值。 （即逐层地，从左到右访问所有节点）。

> 示例： 二叉树：[3,9,20,null,null,15,7],

```
  3
 / \
```

```
  9  20
    /  \
   15   7
```

> 返回其层次遍历结果：
> [
> [3],
> [9,20],
> [15,7]
>]

思路分析

层序遍历没有那么多幺蛾子，大家看到层序遍历就应该条件反射出 `BFS+队列` 这对好基友。所谓变体，不过是在 `BFS` 的过程中围绕结果数组的内容做文章。
拿这道题来说，相对于我们 14 节中讲过的层序遍历基本思路，它变出的花样仅仅在于要求我们对层序遍历结果进行**分层**。也就是说只要我们能在 `BFS` 的过程中感知到当前层级、同时用不同的数组把不同的层级区分开，这道题就得解了。

如何做到这一点？大家需要知道一个非常重要的信息：我们在对二叉树进行层序遍历时，每一次 `while` 循环其实都对应着二叉树的某一层。只要我们在进入 `while` 循环之初，记录下这一层结点个数，然后将这个数量范围内的元素 `push` 进同一个数组，就能够实现二叉树的分层。

编码实现

```
/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
const levelOrder = function(root) {
    // 初始化结果数组
    const res = []
    // 处理边界条件
    if(!root) {
        return res
    }
    // 初始化队列
    const queue = []
    // 队列第一个元素是根结点
    queue.push(root)
```

```
    // 当队列不为空时，反复执行以下逻辑
    while(queue.length) {
        // level 用来存储当前层的结点
        const level = []
        // 缓存刚进入循环时的队列长度，这一步很关键，因为队列长度后面会发生改变
        const len = queue.length
        // 循环遍历当前层级的结点
        for(let i=0;i<len;i++) {
            // 取出队列的头部元素
            const top = queue.shift()
            // 将头部元素的值推入 level 数组
            level.push(top.val)
            // 如果当前结点有左孩子，则推入下一层级
            if(top.left) {
                queue.push(top.left)
            }
            // 如果当前结点有右孩子，则推入下一层级
            if(top.right) {
                queue.push(top.right)
            }
        }
        // 将 level 推入结果数组
        res.push(level)
    }
    // 返回结果数组
    return res
};
```

翻转二叉树

翻转二叉树是一个非常经典的问题。之前有一个关于这道题的笑话，说是 Homebrew 的作者去面 Google，结果因为不会翻转二叉树被挂掉了。Google 在给这位大佬的拒信中写道：

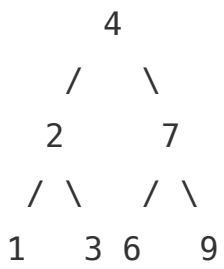
> 我们90%的工程师使用您编写的软件(Homebrew)，但是您却无法在面试时在白板上写出翻转二叉树这道题，这太糟糕了。

这个故事之所以是个笑话，是因为翻转二叉树在算法面试中实在太常见了——只要你准备算法面试，你就不得不做这个题。在面试中做不出这道题的同学，会给面试官留下基础不牢的糟糕印象。

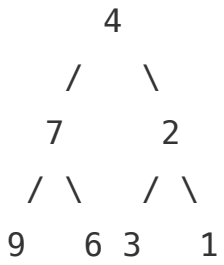
接下来我们就一起来搞定这道翻转二叉树，成为比 homebrew 作者更懂算法面试的人（逃

> 题目描述：翻转一棵二叉树。

> 示例：
> 输入：



> 输出：



思路分析

这道题是一道非常经典的递归应用题。
一棵二叉树，经过翻转后会有什么特点？答案是每一棵子树的左孩子和右孩子都发生了交换。既然是“每一棵子树”，那么就意味着重复，既然涉及了重复，就没有理由不用递归。

于是这道题解题思路就非常明确了：以递归的方式，遍历树中的每一个结点，并将每一个结点的左右孩子进行交换。

编码实现

```
/**
 * @param {TreeNode} root
 * @return {TreeNode}
 */
const invertTree = function(root) {
    // 定义递归边界
    if(!root) {
        return root;
    }
    // 递归交换右孩子的子结点
    let right = invertTree(root.right);
    // 递归交换左孩子的子结点
    let left = invertTree(root.left);
    // 交换当前遍历到的两个左右孩子结点
    root.left = right;
    root.right = left;
    return root;
};
```

（阅读过程中有任何想法或疑问，或者单纯希望和笔者交个朋友啥的，欢迎大家添加我的微信xyalinode与我交流哈~）