平衡不平衡看高度，注意不是深度

110.平衡二叉树

给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过1。

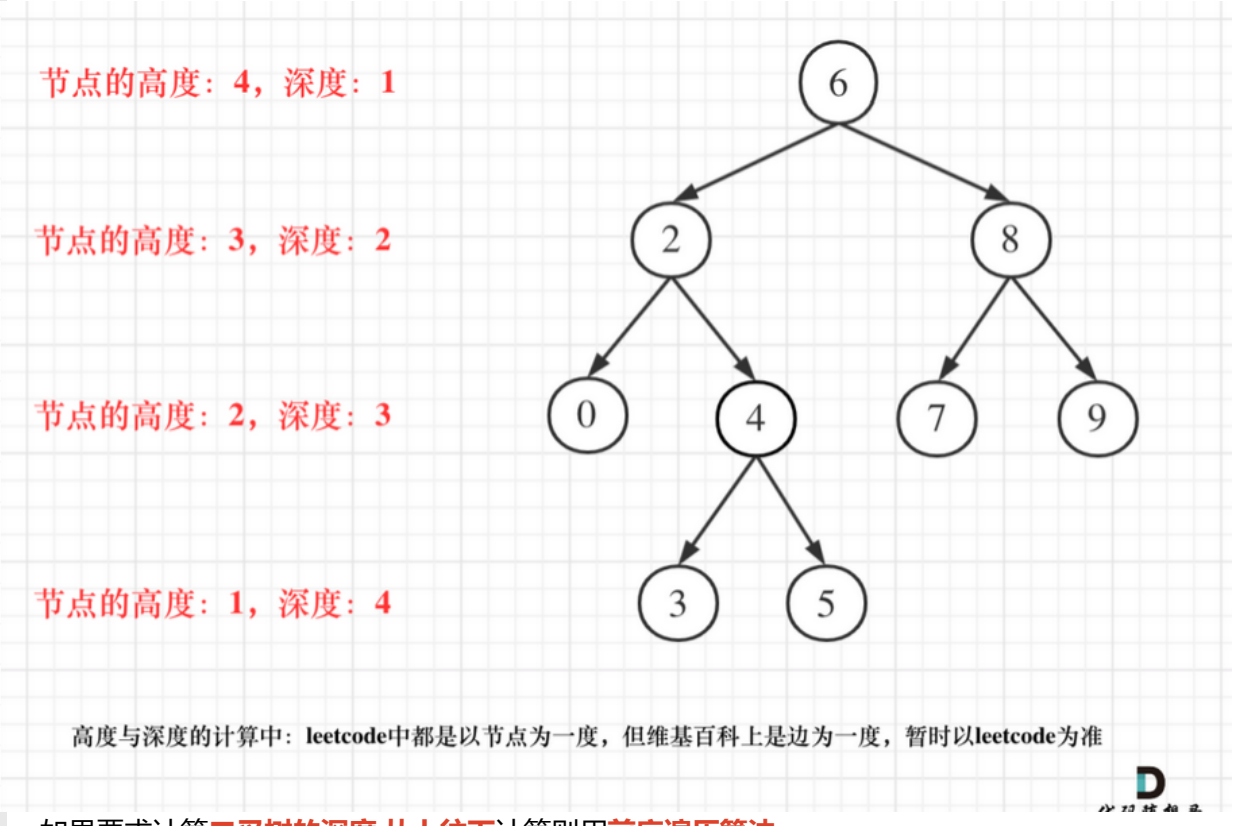
示例 1:

给定二叉树 [3,9,20,null,null,15,7]

```
    3
   / \
  9  20
    /  \
   15   7
```

返回 true 。

二叉树的节点的深度：从**根节点**到**该节点**的**最长简单路径所经过的节点数**
二叉树的节点的高度：从该节点到叶子节点的最长简单路径所经过的节点数



如果要求计算**二叉树的深度**,**从上往下**计算则用**前序遍历算法**

> 如果要求计算**二叉树的高度**,**从下往上**计算则用**后序遍历算法**

用前序遍历算法求最大深度此时用到了回溯的思想

```
class Solution {
public:
    int result
 void getDepth(TreeNode* node, int depth) {
 if(node->left==NULL&&node->right==NULL){//终止条件
 result=max(result,depth);
 return ;
 }
 if(node->left){
  depth++;//回溯算法
  getDepth(node->left,depth);
  depth--;
  }
  if(node->right){
  depth++;
  getDepth(node->right,depth);
  depth--;
  }
  }
  int maxDepth(TreeNode* root) {
        result=0;
  if(root==NULL) return result;
  getDepth(root,1);
  result;
     }
 };
```

用后序遍历算法求是否是平衡二叉树

```
class Solution {
public:
    bool flag=true;
    int getMaxHeight(TreeNode*root){//返回值为该节点的高度
        if(flag==false) return -1;//如果flag==false则无需计算高度了直接返回即可
        if(root->left==NULL&&root->right==NULL) return 1;
        if(root->left==NULL&&root->right!=NULL){
            int right=getMaxHeight(root->right);
            if(right>1) flag=false;
```

```
            return right+1;
        }
        if(root->right==NULL&&root->left!=NULL){
            int left=getMaxHeight(root->left);
            if(left>1) flag=false;
            return left+1;
        }
        //左右子树都不为空
        int left=getMaxHeight(root->left);
        int right=getMaxHeight(root->right);
        if((left-right)>1||(left-right)<-1)flag=false;
        return max(left,right)+1;
    }
    bool isBalanced(TreeNode* root) {
        if(root==NULL) return true;
        if(root->left==NULL&&root->right==NULL) return true;
        int height=getMaxHeight(root);
        return flag;
    }
};
```

无需用一个flag表示直接在返回值中表示出来即可，即如果返回为-1则不再是平衡二叉树

```
class Solution {
public:
 //返回值为该节点的高度，如果返回-1表示已不再是一个平衡二叉树
 int getDepth(TreeNode* root){
 if(root==NULL) return 0;
 int left=getDepth(root->left);
 if(left==-1) return -1;//已经不再是一棵平衡二叉树了没必要进行递归下去
 int right=getDepth(root->right);
 if(right==-1) return -1;//已经不再是一棵平衡二叉树没必要进行递归下去
  return ((left-right)>1||(left-right)<-1)?-1:max(left,right)+1;
  }
    bool isBalanced(TreeNode* root) {
  return getDepth==-1?false:true;
    }
};
```

二叉树：找我的所有路径?

以为只用了**递归**，其实还用了**回溯**

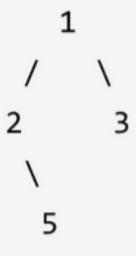
257. 二叉树的所有路径

给定一个二叉树，返回所有从根节点到叶子节点的路径。

说明: 叶子节点是指没有子节点的节点。

示例:

输入:



输出: ["1->2->5", "1->3"]

解释: 所有根节点到叶子节点的路径为: 1->2->5, 1->3

路径的记录必须是先记录当前节点然后去记录左右子树的路径因此采用的前序遍历算法
此时由于还有回退的过程，则采用的是回溯算法+前序遍历算法

step1：递归函数函数参数以及返回值

void traversal(TreeNode* cur,vector<int>&path,vector<string>& result);

step2:确定递归终止条件

一般二叉树的终止条件为两个
找到叶子节点
if(cur->left==NULL&&cur->right==NULL) 终止处理逻辑;
或者遍历到叶子节点的左右空孩子
if(cur==NULL)终止处理逻辑;

1、为什么使用vector结构来记录路径呢?

因为在下面处理单层递归逻辑的时候，要做回溯，用vector方便来做回溯

2、一个递归就要一个回溯,递归出来就要紧跟着一个回溯，

```
class Solution {
public:
 void traversal(TreeNode* cur,vector<int>& sPath,vector<string>& result){
 if(cur->left==NULL&&cur->right==NULL){
 sPath.push_back(cur->val);
 int size=sPath.size();
 string s;
 for(int i=0;i<size-1;i++){
```

```
 s+=to_string(sPath[i]);
 s+="->";
 }
 s+=to_string(sPath[size-1]);
 result.push_back(s);
 sPath.pop_back();//有一个递归就要一个回溯
 return ;
 }
 //左子树
 if(cur->left){
 sPath.push_back(cur->val);
 traversal(cur->left,sPath,result);
 sPath.pop_back();
 }
 //右子树
 if(cur->right){
 sPath.push_back(cur->val);
 traversal(cur->right,sPath,result);
 sPath.pop_back();
 }
 }
 vector<string> binaryTreePaths(TreeNode* root) {
 vector<string>result;
 if(root==NULL) return result;
 vector<int>sPath;
 traversal(root,sPath,result);
 return result;
    }
};

```

简洁版本回溯算法

```
//使用回溯的时候就要用引用保持数据的一致性
class Solution {
public:
 void traversal(TreeNode* root,string path,vector<string>&result){
 path+=to_string(root->val);
 //找到叶子节点
 if(root->left==NULL&&root->right==NULL){
 result.push_back(path);
 }
```

```
  if(root->left)traversal(root->left,path+"->",result);
  if(root->right)traversal(root->right,path+"->",result);
  }


     vector<string> binaryTreePaths(TreeNode* root) {
  vector<string> result;
  if(root==NULL) return result;
  traversal(root,"",result);
  return result;
     }
};
```

分析该版本的代码：

看过去好像没有回溯的过程实则不然，

```
if (cur->left) traversal(cur->left, path + "->", result); // 左
if (cur->right) traversal(cur->right, path + "->", result); // 右
```

path+"->"在递归传递的值为path+"->"，返回出来的时候path还是原来的path没有加上"->"

迭代写法：

用两个栈来模拟递归，
入栈->就是将函数放在栈中排列等候运行
出栈->就是进入一个函数获取相应的形参,while循环结束后就是返回值

```
class Solution {
public:
     vector<string> binaryTreePaths(TreeNode* root) {
  stack<TreeNode*> TreeSet;//存放遍历的节点
  stack<string> pathSet;//保存遍历路径的节点
  vector<string>result;
  if(root==NULL) return result;
  pathSet.push_back(to_string(root->val));//已经访问了第一个节点
  TreeSet.push_back(root);
   while(!TreeSet.empty()){
   TreeNode* node=TreeSet.top();TreeSet.pop();
   string path=pathSet.top();pathSet.pop();
   if(node->left==NULL&&node->right==NULL){
   result.push_back(path);
   }
   if(node->right){
   TreeSet.push(node->right);
   pathSet.push(path+"->"+to_string(node->right->val));
```

```
 }
 if(node->left){
 TreeSet.push(node->left);
 pathSet.push(path+"->"+to_string(node->left->val));
 }
 }
 return result;
    }
};
```

比较前后两个解法的pathSet的区别

```
stack<TreeNode*> TreeSet;//存放遍历的节点
      stack<string> pathSet;//保存遍历路径的节点
      vector<string>result;
      if(root==NULL) return result;
      TreeSet.push(root);
      pathSet.push("");
      while(!TreeSet.empty()){
          TreeNode* node=TreeSet.top();TreeSet.pop();
          string path="";
           if(node==root)path=pathSet.top()+to_string(node->val);
           else path=pathSet.top()+"->"+to_string(node->val);
           pathSet.pop();
           if((node->left==NULL)&&(node->right==NULL)){
               result.push_back(path);
           }
           if(node->right){
               TreeSet.push(node->right);
               pathSet.push(path);
           }
            if(node->left){
               TreeSet.push(node->left);
               pathSet.push(path);
           }
       }
       return result;
```

总结：

两道比较子树的问题:

那么做完本题之后，再看如下两个题目。

- 100.相同的树
- 572.另一个树的子树

「二叉树：我对称么？中的递归法和迭代法只需要稍作修改其中一个树的遍历顺序，便可刷了100.相同的树。」