LeetCode 第 110 号问题：平衡二叉树

> *本文首发于公众号「图解面试算法」，是 图解 LeetCode 系列文章之一。*
>
> *同步博客：https://www.algomooc.com*

题目来源于 LeetCode 上第 110 号问题：平衡二叉树。

题目描述

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:

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

返回 `true` 。

示例 2:

给定二叉树 `[1,2,2,3,3,null,null,4,4]`

```
       1
      / \
     2   2
    / \
   3   3
  / \
 4   4
```

返回 `false` 。

题目解析 - 自顶向下

这道题可以算是递归的充分使用了, 每一个子树都是子问题.

根据题意, 直观的想法就是计算当前节点左右子树的高度差了, 具体算法流程如下:

定义 方法 `depth(root)` 计算 root 最大高度

- **终止条件：** 当 `root` 为空，即越过叶子节点，则返回高度 0
- **返回值：** Max(左子树高度, 右子树高度) + 1

定义 方法 `isBalanced(root)` 判断树 `root` 是否平衡

- **特例处理：** 若树根节点 `root` 为空，则直接返回 true
- **返回值：** 所有子树都需要满足平衡树性质，因此以下三者使用与 逻辑与 连接

- `abs(depth(root.left) - depth(root.right)) < 2` ：判断 **当前子树** 是否是平衡树
- `isBalanced(root.left)` ： 先序遍历递归，判断 **当前子树的左子树** 是否是平衡树；
- `isBalanced(root.right)` ： 先序遍历递归，判断 **当前子树的右子树** 是否是平衡树；

> *通过流程能发现, 暴力法虽然容易想到, 但是会产生大量冗余计算, 因此时间复杂度也就会高;*
>
> *想避免这种情况, 移步向下看 自底向上 方法*

动画描述

 Animation1

参考代码

```
/**
 * JavaScript 描述
 * 自顶向下递归
 */
function depth(root) {
    if (root == null) {
        return 0;
    }
    return Math.max(depth(root.left), depth(root.right)) + 1;
};
var isBalanced = function(root) {
    if (root == null) {
        return true;
    }
    return Math.abs(depth(root.left) - depth(root.right)) < 2 &&
           isBalanced(root.left) &&
           isBalanced(root.right);
};
```

复杂度分析

- 时间复杂度: **O(Nlog_2 N)**

  最差情况下， isBalanced(root) 遍历树所有节点，占用 O(N)O(N)；判断每个节点的最大高度 depth(root) 需要遍历 各子树的所有节点 ，子树的节点数的复杂度为 O(log_2 N)

- 空间复杂度: **O(N)**

  最差情况下（树退化为链表时），系统递归需要使用 O(N) 的栈空间

题目解析 - 自底向上

自顶向下 计算 `depth` 存在大量冗余, 每次调用 `depth` 时，要同时计算其子树高度。

自底向上 计算每个子树的高度只会计算一次。先递归计算当前节点的子节点高度，然后再通过子节点高度判断当前节点是否平衡，从而消除冗余。

自底向上 与 **自顶向下** 的逻辑相反，首先判断子树是否平衡，然后比较子树高度判断父节点是否平衡。算法如下：

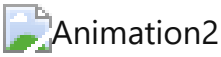
定义 方法 `recur(root)` ：判断子树是否平衡 | 返回当前节点高度

- **递归终止条件：**
  - 当越过叶子节点时, 返回高度 0
  - 当左（右）子树高度 `left== -1` 时，代表此子树的 **左（右）子树** 不是平衡树, 因此直接返回 `-1`
- **递归返回值：**
  - 当节点 `root` 左 / 右子树的高度差 < 2：返回以节点 root 为根节点的子树的最大高度Max(left, right) + 1
  - 当节点 `root` 左 / 右子树的高度差 >= 2 ： 则返回 `-1` , 代表 **此子树不是平衡树**

定义 方法 `isBalanced(root)` ：判断当前树是否平衡

- **返回值：** 若 `recur(root) != 1` , 则说明此树平衡, 返回 `true` , 否则返回 `false`

动画描述



参考代码

```
/**
 * JavaScript 描述
 * 自底向上递归
 */
function recur(root) {
    if (root == null) {
        return 0;
    }
    let leftHeight = recur(root.left);
    if (leftHeight == -1) {
        return -1;
    }
    let rightHeight = recur(root.right);
    if (rightHeight == -1) {
        return -1;
    }
    return Math.abs(leftHeight - rightHeight) < 2 ?
           Math.max(leftHeight,rightHeight) + 1 : -1;
};
var isBalanced = function(root) {
    return recur(root) != -1;
};
```

复杂度分析

- 时间复杂度 **O(N)**： N为树的节点数；最差情况下，需要递归遍历树的所有节点。
- 空间复杂度 **O(N)**： 最差情况下（树退化为链表时），系统递归需要使用 O(N) 的栈空间。