题目地址（297. 二叉树的序列化与反序列化）

https://leetcode-cn.com/problems/serialize-and-deserialize-binary-tree/

入选理由

1. 难度不小的一个题，但是我的树专题其实已经分析了这道题，同样可以使用两种方式来解决，大家来试试吧

标签

- 树
- BFS
- DFS

难度

- 困难

题目描述

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这

示例：

你可以将以下二叉树：

```
    1
   / \
  2   3
     / \
    4   5
```

序列化为 `"[1,2,3,null,null,4,5]"`

提示：这与 LeetCode 目前使用的方式一致，详情请参阅 LeetCode 序列化二叉树的格式。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。

说明：不要使用类的成员 / 全局 / 静态变量来存储状态，你的序列化和反序列化算法应该是无状态的。

思路(BFS)

如果我将一个二叉树的完全二叉树形式序列化，然后通过 BFS 反序列化，这不就是力扣官方序列化树的方式么？比如：

```
  1
 / \
2   3
   / \
  4   5
```

序列化为 "[1,2,3,null,null,4,5]"。 这不就是我刚刚画的完全二叉树么？就是将一个普通的二叉树硬生生当成完全二叉树用了。

> 其实这并不是序列化成了完全二叉树，下面会纠正。

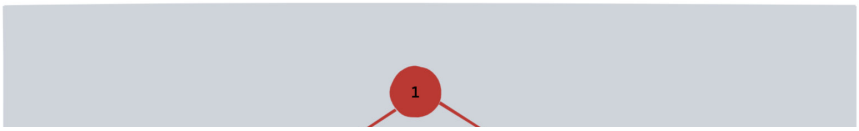
将一颗普通树序列化为完全二叉树很简单，只要将空节点当成普通节点入队处理即可。代码：

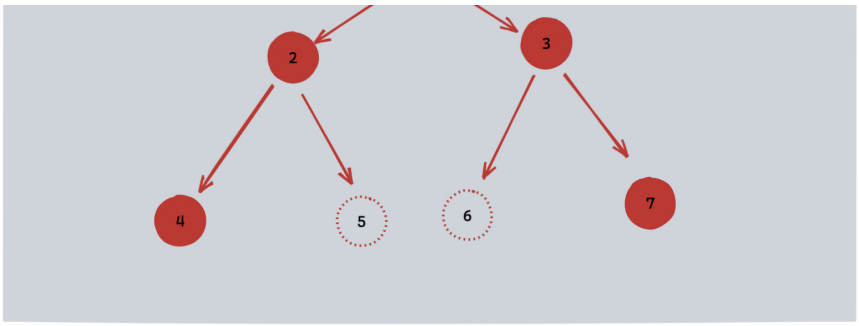
```
class Codec:

    def serialize(self, root):
        q = collections.deque([root])
        ans = ''
        while q:
            cur = q.popleft()
            if cur:
                ans += str(cur.val) + ','
                q.append(cur.left)
                q.append(cur.right)
            else:
                # 除了这里不一样，其他和普通的不记录层的 BFS 没区别
                ans += 'null,'
        # 末尾会多一个逗号，我们去掉它。
        return ans[:-1]
```

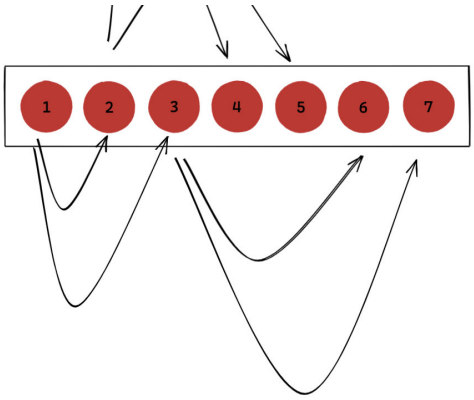
细心的同学可能会发现，我上面的代码其实并不是将树序列化成了完全二叉树，这个我们稍后就会讲到。另外后面多余的空节点也一并序列化了。这其实是可以优化的，优化的方式也很简单，那就是去除末尾的 null 即可。

你只要彻底理解我刚才讲的 我们可以给完全二叉树编号，这样父子之间就可以通过编号轻松求出。比如我给所有节点从左到右从上到下依次从 1 开始编号。那么已知一个节点的编号是 i，那么其左子节点就是 2 * i，右子节点就是 2 * i + 1，父节点就是 i / 2。 这句话，那么反序列化对你就不是难事。

如果我用一个箭头表示节点的父子关系，箭头指向节点的两个子节点，那么大概是这样的：







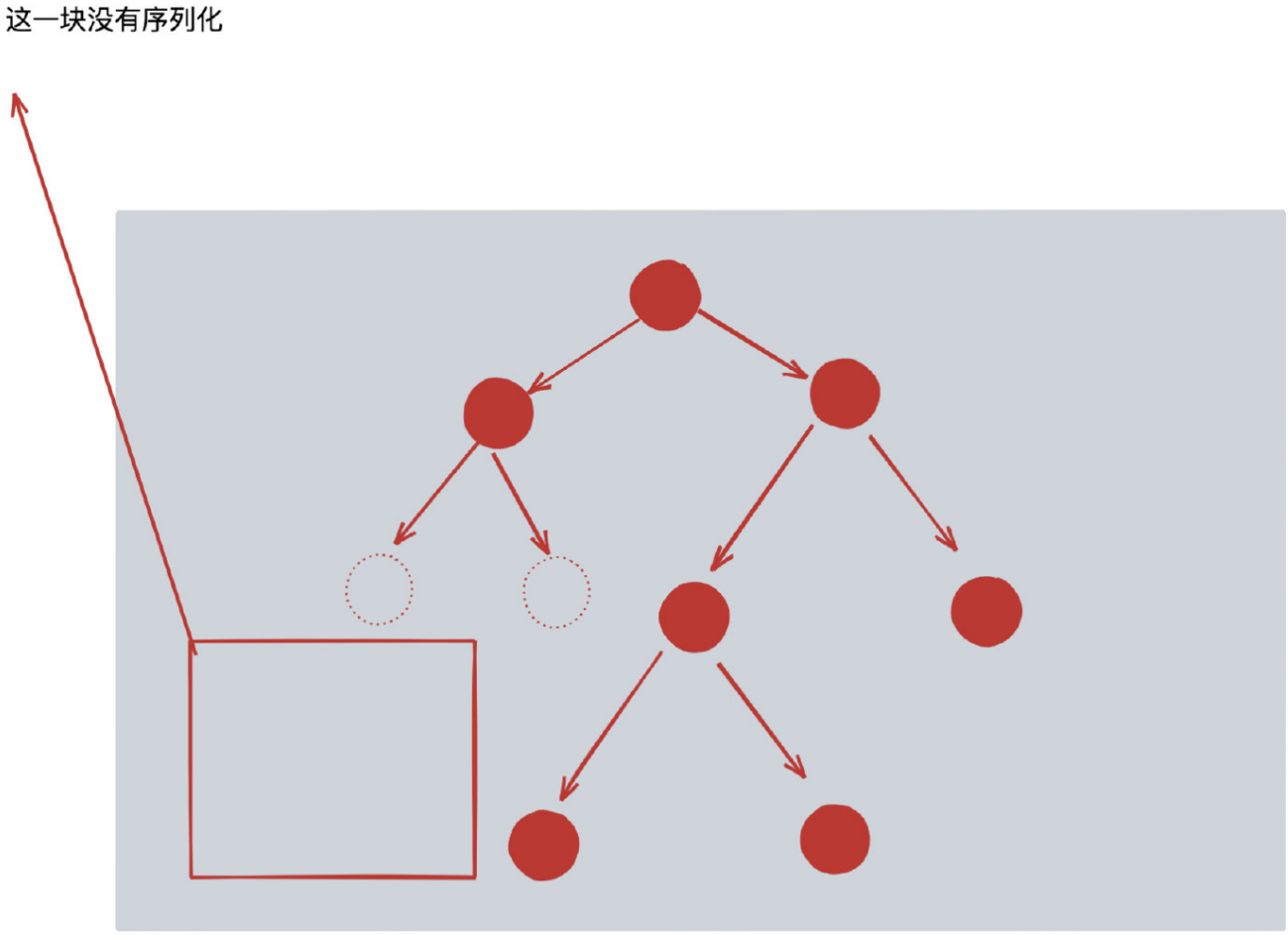
我们刚才提到了：

- 1 号节点的两个子节点的 2 号 和 3 号。
- 2 号节点的两个子节点的 4 号 和 5 号。
- 。。。
- i 号节点的两个子节点的 `2 * i` 号 和 `2 * i + 1` 号。

此时你可能会写出类似这样的代码：

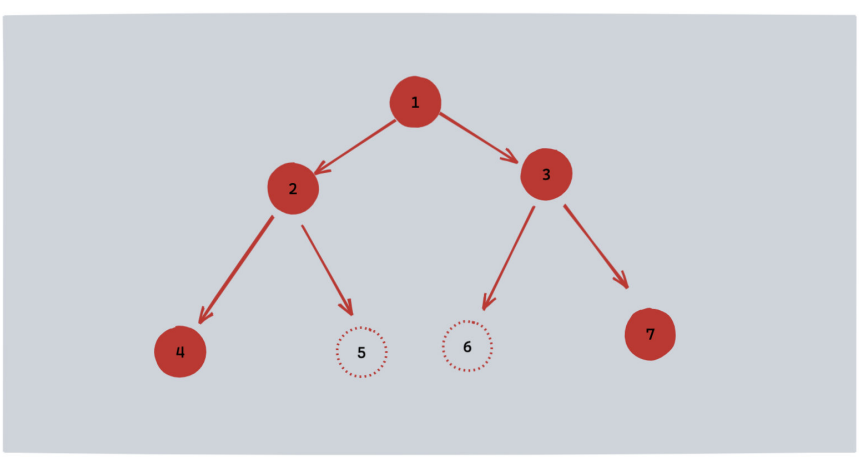
```
def deserialize(self, data):
    if data == 'null': return None
    nodes = data.split(',')
    root = TreeNode(nodes[0])
    # 从一号开始编号，编号信息一起入队
    q = collections.deque([(root, 1)])
    while q:
        cur, i = q.popleft()
        # 2 * i 是左节点，而 2 * i 编号对应的其实是索引为 2 * i - 1 的元素， 右节点同理。
        if 2 * i - 1 < len(nodes): lv = nodes[2 * i - 1]
        if 2 * i < len(nodes): rv = nodes[2 * i]
        if lv != 'null':
            l = TreeNode(lv)
            # 将左节点和 它的编号 2 * i 入队
            q.append((l, 2 * i))
            cur.left = l
        if rv != 'null':
            r = TreeNode(rv)
            # 将右节点和 它的编号 2 * i + 1 入队
            q.append((r, 2 * i + 1))
            cur.right = r

    return root
```

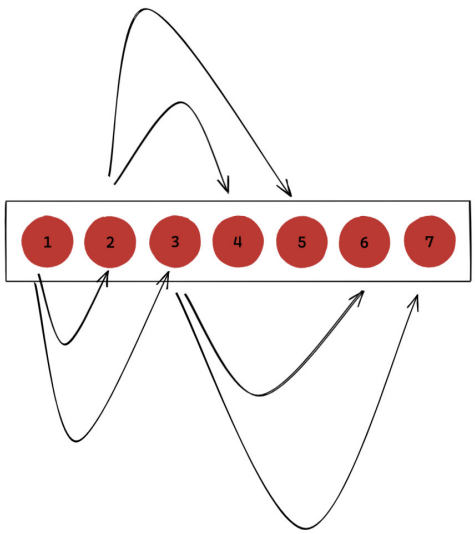
但是上面的代码是不对的，因为我们序列化的时候其实不是完全二叉树，这也是上面我埋下的伏笔。因此遇到类似这样的 case 就会挂：



这也是我前面说”上面代码的序列化并不是一颗完全二叉树“的原因。

其实这个很好解决， 核心还是上面我画的那种图：





其实我们可以：

- 用三个指针分别指向数组第一项，第二项和第三项（如果存在的话），这里用 p1，p2，p3 来标记，分别表示当前处理的节点，当前处理的节点的左子节点和当前处理的节点的右子节点。
- p1 每次移动一位，p2 和 p3 每次移动两位。
- p1.left = p2; p1.right = p3。
- 持续上面的步骤直到 p1 移动到最后。

因此代码就不难写出了。反序列化代码如下：

```
def deserialize(self, data):
    if data == 'null': return None
    nodes = data.split(',')
    root = TreeNode(nodes[0])
    q = collections.deque([root])
    i = 0
    while q and i < len(nodes) - 2:
        cur = q.popleft()
        lv = nodes[i + 1]
        rv = nodes[i + 2]
        i += 2
        if lv != 'null':
            l = TreeNode(lv)
            q.append(l)
            cur.left = l
        if rv != 'null':
            r = TreeNode(rv)
            q.append(r)
            cur.right = r

    return root
```

这个题目虽然并不是完全二叉树的题目，但是却和完全二叉树很像，有借鉴完全二叉树的地方。

代码

代码支持：JS，Python

JS Code：

```
const serialize = (root) => {
  const queue = [root];
  let res = [];
  while (queue.length) {
    const node = queue.shift();
```

```
    if (node) {
      res.push(node.val);
      queue.push(node.left);
      queue.push(node.right);
    } else {
      res.push("#");
    }
  }
  return res.join(",");
};

const deserialize = (data) => {
  if (data == "#") return null;

  const list = data.split(",");

  const root = new TreeNode(list[0]);
  const queue = [root];
  let cursor = 1;

  while (cursor < list.length) {
    const node = queue.shift();

    const leftVal = list[cursor];
    const rightVal = list[cursor + 1];

    if (leftVal != "#") {
      const leftNode = new TreeNode(leftVal);
      node.left = leftNode;
      queue.push(leftNode);
    }
    if (rightVal != "#") {
      const rightNode = new TreeNode(rightVal);
      node.right = rightNode;
      queue.push(rightNode);
    }
    cursor += 2;
  }
  return root;
};
```

Python Code:

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
```

```python
class Codec:
    def serialize(self, root):
        ans = ''
        queue = [root]
        while queue:
            node = queue.pop(0)
            if node:
                ans += str(node.val) + ','
                queue.append(node.left)
                queue.append(node.right)
            else:
                ans += '#,'
        print(ans[:-1])
        return ans[:-1]



    def deserialize(self, data: str):
        if data == '#': return None
        nodes = data.split(',')
        if not nodes: return None
        root = TreeNode(nodes[0])
        queue = [root]
        # 已经有 root 了，因此从 1 开始
        i = 1

        while i < len(nodes) - 1:
            node = queue.pop(0)
            lv = nodes[i]
            rv = nodes[i + 1]
            i += 2
            if lv != '#':
                l = TreeNode(lv)
                node.left = l
                queue.append(l)

            if rv != '#':
                r = TreeNode(rv)
                node.right = r
                queue.append(r)
        return root
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为树的节点数。
- 空间复杂度：$O(Q)$，其中 Q 为队列长度，最坏的情况是满二叉树，此时和 N 同阶，其中 N 为树的节点总数

DFS 其实思路也是类似的， 比如我使用前序遍历，那么代码就是这样的：

Python Code:

```python
class Codec:
    def serialize(self, root):
        def preorder(root):
            if not root:
                return "null,"
            return str(root.val) + "," + preorder(root.left) + preorder(root.right)

        return preorder(root)[:-1]

    def deserialize(self, data: str):
        nodes = data.split(",")

        def preorder(i):
            if i >= len(nodes) or nodes[i] == "null":
                return i, None
            root = TreeNode(nodes[i])
            j, root.left = preorder(i + 1)
            k, root.right = preorder(j + 1)
            return k, root

        return preorder(0)[1]
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为树的节点数。
- 空间复杂度：$O(h)$，其中 h 为树的高度，最坏的情况是链表，此时和 N 同阶，其中 N 为树的节点总数

上一页 下一页

© 2020 lucifer. 保留所有权利