```
        String space = " ";
        StringBuffer buf = new StringBuffer("");
        for (int i = 0; i < num; i++) {
                buf.append(space);
        }
        return buf.toString();
    }
```

【扩展】

有关功能设计的面试题，其实最难的部分并不是设计，而是在设计的优良性和实现的复杂程度之间找到一个平衡性最好的设计方案。在满足功能要求的同时，也要保证在面试场上能够完成大致的代码实现，同时对边界条件的梳理能力和代码逻辑的实现能力也是一大挑战。读者可以看到本书提供的方法在完成功能的同时其代码很少，也请读者设计自己的方案并实现它。

二叉树的序列化和反序列化

【题目】

二叉树被记录成文件的过程叫作二叉树的序列化，通过文件内容重建原来二叉树的过程叫作二叉树的反序列化。给定一棵二叉树的头节点 head，并已知二叉树节点值的类型为 32 位整型。请设计一种二叉树序列化和反序列化的方案，并用代码实现。

【难度】

士 ★☆☆☆

【解答】

本书提供两套序列化和反序列化的实现，供读者参考。

方法一：通过先序遍历实现序列化和反序列化。

先介绍先序遍历下的序列化过程，首先假设序列化的结果字符串为 str，初始时 str=""。先序遍历二叉树，如果遇到 null 节点，就在 str 的末尾加上“#!”，“#”表示这个节点为空，节点值不存在，“!”表示一个值的结束；如果遇到不为空的节点，假设节点值为 3，就在 str 的末尾加上“3!”。比如图 3-6 所示的二叉树。

根据上文的描述，先序遍历序列化，最后的结果字符串 str 为：12!3!#!#!#!。

为什么在每一个节点值的后面都要加上“!”呢？因为如果不标记一个值的结束，最后产生的结果会有歧义，如图 3-7 所示。

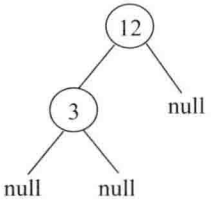

图 3-6

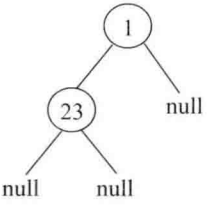

图 3-7

如果不在一个值结束时加入特殊字符，那么图 3-6 和图 3-7 的先序遍历序列化结果都是 123###。也就是说，生成的字符串并不代表唯一的树。

先序遍历序列化的全部过程请参看如下代码中的 serialByPre 方法。

```
public class Node {
        public int value;
        public Node left;
        public Node right;

        public Node(int data) {
                this.value = data;
        }
}

public String serialByPre(Node head) {
        if (head == null) {
                return "#!";
        }
        String res = head.value + "!";
        res += serialByPre(head.left);
        res += serialByPre(head.right);
        return res;
}
```

接下来介绍如何通过先序遍历序列化的结果字符串 str，重构二叉树的过程，即反序列化。

把结果字符串 str 变成字符串类型的数组，记为 values，数组代表一棵二叉树先序遍历的节点顺序。例如，str="12!3!#!#!#! "，生成的 values 为["12","3","#","#","#"]，然后用 values[0..4]按照先序遍历的顺序建立整棵树。

1．遇到"12"，生成节点值为 12 的节点(head)，然后用 values[1..4]建立节点 12 的左子树。

2．遇到"3"，生成节点值为 3 的节点，它是节点 12 的左孩子，然后用 values[2..4]建立节点 3 的左子树。

3．遇到"#"，生成 null 节点，它是节点 3 的左孩子，该节点为 null，所以这个节点没有后续建立子树的过程。回到节点 3 后，用 values[3..4]建立节点 3 的右子树。

4．遇到"#"，生成 null 节点，它是节点 3 的右孩子，该节点为 null，所以这个节点没有后续建立子树的过程。回到节点 3 后，再回到节点 1，用 values[4]建立节点 1 的右子树。

5．遇到"#"，生成 null 节点，它是节点 1 的右孩子，该节点为 null，所以这个节点没有后续建立子树的过程。整个过程结束。

先序遍历反序列化的全部过程请参看如下代码中的 reconByPreString 方法。

```
public Node reconByPreString(String preStr) {
        String[] values = preStr.split("!");
        Queue<String> queue = new LinkedList<String>();
        for (int i = 0; i != values.length; i++) {
                queue.offer(values[i]);
        }
        return reconPreOrder(queue);
}

public Node reconPreOrder(Queue<String> queue) {
        String value = queue.poll();
        if (value.equals("#")) {
                return null;
        }
        Node head = new Node(Integer.valueOf(value));
        head.left = reconPreOrder(queue);
        head.right = reconPreOrder(queue);
        return head;
}
```

方法二：通过层遍历实现序列化和反序列化。

先介绍层遍历下的序列化过程，首先假设序列化的结果字符串为 str，初始时 str="空"。然后实现二叉树的按层遍历，具体方式是利用队列结构，这也是宽度遍历图的常见方式。例如，图 3-8 所示的二叉树。

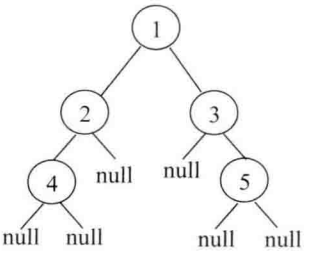

图 3-8

按层遍历图 3-8 所示的二叉树，最后 str="1!2!3!4!#!#!5!#!#!#!#! "。

层遍历序列化的全部过程请参看如下代码中的 serialByLevel 方法。

```
public String serialByLevel(Node head) {
        if (head == null) {
                return "#!";
        }
        String res = head.value + "!";
        Queue<Node> queue = new LinkedList<Node>();
        queue.offer(head);
        while (!queue.isEmpty()) {
                head = queue.poll();
                if (head.left != null) {
                        res += head.left.value + "!";
                        queue.offer(head.left);
                } else {
                        res += "#!";
                }
                if (head.right != null) {
                        res += head.right.value + "!";
                        queue.offer(head.right);
                } else {
                        res += "#!";
                }
        }
        return res;
}
```

先序遍历的反序列化其实就是重做先序遍历，遇到"#"就生成 null 节点，结束生成后续子树的过程。

与根据先序遍历的反序列化过程一样，根据层遍历的反序列化是重做层遍历，遇到"#"就生成 null 节点，同时不把 null 节点放到队列里即可。

层遍历反序列化的全部过程请参看如下代码中的 reconByLevelString 方法。

```
public Node reconByLevelString(String levelStr) {
        String[] values = levelStr.split("!");
        int index = 0;
        Node head = generateNodeByString(values[index++]);
        Queue<Node> queue = new LinkedList<Node>();
        if (head != null) {
                queue.offer(head);
        }
        Node node = null;
        while (!queue.isEmpty()) {
                node = queue.poll();
                node.left = generateNodeByString(values[index++]);
                node.right = generateNodeByString(values[index++]);
                if (node.left != null) {
```

```
                queue.offer(node.left);
            }
            if (node.right != null) {
                queue.offer(node.right);
            }
        }
        return head;
    }

    public Node generateNodeByString(String val) {
        if (val.equals("#")) {
            return null;
        }
        return new Node(Integer.valueOf(val));
    }
```

遍历二叉树的神级方法

【题目】

给定一棵二叉树的头节点 head，完成二叉树的先序、中序和后序遍历。如果二叉树的节点数为 N，要求时间复杂度为 $O(N)$，额外空间复杂度为 $O(1)$。

【难度】

将 ★★★★

【解答】

本题真正的难点在于对复杂度的要求，尤其是额外空间复杂度为 $O(1)$的限制。之前的题目已经剖析过如何用递归和非递归的方法实现遍历二叉树，很不幸，之前所有的方法虽然常用，但都无法做到额外空间复杂度为 $O(1)$。这是因为遍历二叉树的递归方法实际使用了函数栈，非递归的方法使用了申请的栈，两者的额外空间都与树的高度相关，所以空间复杂度为 $O(h)$，h 为二叉树的高度。如果完全不用栈结构能完成三种遍历吗？可以。答案是使用二叉树节点中大量指向 null 的指针，本题实际上就是大名鼎鼎的 Morris 遍历，由 Joseph Morris 于 1979 年发明。

首先来看普通的递归和非递归解法，其实都使用了栈结构，在处理完二叉树某个节点后可以回到上层去。为什么从下层回到上层会如此之难？因为二叉树的结构如此，每个节点都有指向孩子节点的指针，所以从上层到下层容易，但是没有指向父节点的指针，所以