框架刷通二叉树(一)

二叉树的重要性

快速排序就是二叉树的前序遍历，归并排序就是二叉树的后续遍历。 快排的逻辑： 若要对`nums[lo...hi]`进行排序，首先找到一个分界点`p`,通过交换使得`nums[lo...p-1]`都小于`nums[p]`;`nums[p+1...hi]`都大于`nums[p]`. 快排的框架代码：

```
void quickSort(std::vector<int> &nums, int lo, int hi){
  // 前序便利位置
  // 通过交换元素，构建分界点p
  int p = partition(std::vector<int> & nums, int lo, int hi);
  sort(nums, lo, p-1);
  sort(nums, p+1, hi);
}
```

对归并排序，若要对`nums[lo...hi]`进行排序，首先对`nums[lo...mid]`进行排序，在对`nums[mid+1...hi]`进行排序，最后合并 代码框架：

```
void mergeSort(int []nums. int lo, int hi){
  int mid = lo + (hi - lo )/ 2;
  mergeSort(nums, lo, mid);
  mergeSort(nums, mid+1, hi);

  merge(nums, lo, mid, hi);
}
```

递归算法写法

如求二叉树的节点数:

```
class Solution {
public:
  int countNodes(TreeNode *root) {
    if (root == nullptr) {
      return 0;
    }

    return 1 + countNodes(root->left) + countNodes(root->right);
  }
};
```

以根节点为递归退出条件，在分别处理左右子树。

翻转二叉树

```
class Solution {
public:
  TreeNode *invertTree(TreeNode *root) {
    if (root == nullptr) {
      return root;
    }
    TreeNode *tmp = root->left;
    root->left    = root->right;
    root->right   = tmp;
    invertTree(root->left);
    invertTree(root->right);

    return root;
  }
};
```

填充二叉树右侧指针

> 把二叉树的每一层节点都用next指针连接起来

分析： 直接连接left与right节点，只连接了具有公共父节点的next指针； 还需要连接下一层节点的right与left指针。

```
class Solution {
public:
  Node* connect(Node* root) {
    if (!root) {
      return root;
    }
    connectTwoNode(root->left, root->right);

    return root;
  }

private:
  void connectTwoNode(Node* node1, Node* node2) {
    if (nullptr == node1 || nullptr == node2) {
      return;
    }

    node1->next = node2;
    connectTwoNode(node1->left, node1->right);
    connectTwoNode(node2->left, node2->right);

    connectTwoNode(node1->right, node2->left);
  }
};
```

将一颗二叉树拉平成链表

> 给定一颗二叉树，将二叉树的左节点置为空，右节点相连成单链表，链表顺序为先根顺序。

分析：

1. 将`root`的左子树替换到`root`的右节点上；
2. 将`root`的右节点连接到原左子树的最右侧。

```
class Solution {
public:
  void flatten(TreeNode *root) {
    if (root == nullptr) {
      return;
    }

    // 后续遍历
    flatten(root->left);
    flatten(root->right);

    TreeNode *left  = root->left;
    TreeNode *right = root->right;

    // 左子树连接到右子节点
    root->left  = nullptr;
    root->right = left;

    // 原右子树，连接到原左子树的最右侧
    TreeNode *p = root;
    while (p->right) {
      p = p->right;
    }
    p->right = right;
  }
};
```