回溯算法详解

Original labuladong labuladong 2019-05-16 10:19

> 预计阅读时间： 12分钟

回溯算法是啥，怎么听起来那么高端？

实际上回溯算法就是一个 N 叉树的前序遍历加上后序遍历而已，而且回溯算法是有模板的，一旦掌握，就能秒杀相关问题。下面，我们来循序渐进地理解。

```
// 二叉树遍历框架
def traverse(root):
    if root is None: return
    # 前序遍历代码写在这
    traverse(root.left)
    # 中序遍历代码写在这
    traverse(root.right)
    # 后序遍历代码写在这

// N 叉树遍历框架
def traverse(root):
    if root is None: return
    for child in root.children:
        # 前序遍历代码写在这
        traverse(child)
        # 后序遍历代码写在这
```

“树”的遍历框架虽然简单，但是再怎么强调都不为过。因为见的算法多了之后，你会发现，有关递归的算法，都离不开“树”的遍历这一抽象模型。只不过对于不同的算法，在前（中）后序遍历的时候，所做的事不同而已。

比如前文 动态规划详解 中，斐波那契的例子就是一个二叉树，凑零钱的例子就是个 N 叉树，N 为硬币面值的种数。看他们的递归解法代码，都是上边代码模板的具体变形而已。

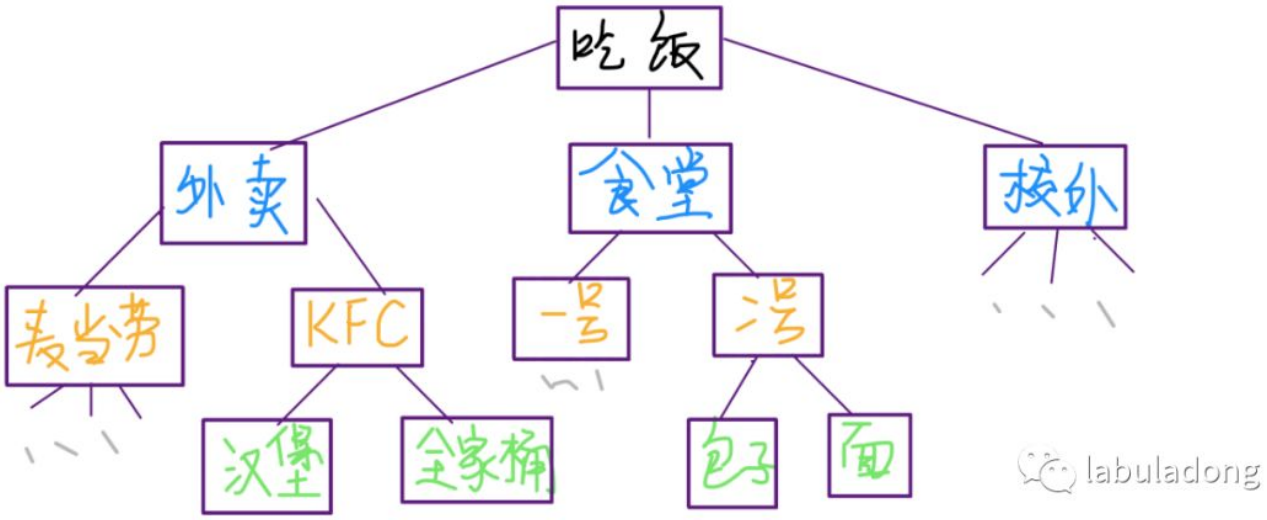
0、算法的整体框架

言归正传，回溯算法就是 N 叉树的遍历，这个 N 等于当前可做的选择（choices）的总数，同时，在前序遍历的位置作出当前选择（choose 过程），然后开始递归，最后在后序遍历的位置取消当前选择（unchoose 过程）。回溯算法伪代码模板如下：

```
"""
choiceList：当前可以进行的选择列表
track：可以理解为决策路径，即已经做出一系列选择
answer：用来储存我们的符合条件决策路径
"""

def backtrack(choiceList, track, answer):
  if track is OK:
    answer.add(track)
  else:
    for choice in choiceList:
      # choose：选择一个 choice 加入 track
      backtrack(choices, track, answer)
      # unchoose：从 track 中撤销上面的选择
```

模板很简单吧，是不是就是一个 N 叉树的遍历？**你可以理解为，回溯算法相当于一个决策过程，递归地遍历一棵决策树，穷举所有的决策，同时把符合条件的决策挑出来。**

举个例子，假如你要吃饭，但是去哪里吃，然后具体吃什么呢？这是个决策问题，而且不太容易，因为选择实在太多了，如下图。



作为人类，也许就看心情随便选一个决策就行了，比如说我可能会选择：吃饭 -> 食堂 -> 一号 -> 大碗宽面。但是计算机会如何选择呢？

前文 动态规划详解 最后说了，计算机处理问题的方法就是穷举。这里计算机会穷举并比较所有决策，然后选出某个最优的决策。比如说，计算机可能会选择：吃饭 -> 外卖 -> KFC -> 全家桶。

如何让计算机正确地穷举并比较所有决策，就需要回溯算法的设计技巧了，**回溯算法的核心就在于如何设计 choose 和 unchoose 部分的逻辑。**

下文通过讲解全排列问题（permutation）和 N 皇后问题来详述回溯算法的原理和技巧。

一、全排列问题

给定一个**没有重复**数字的序列，返回其所有可能的全排列。

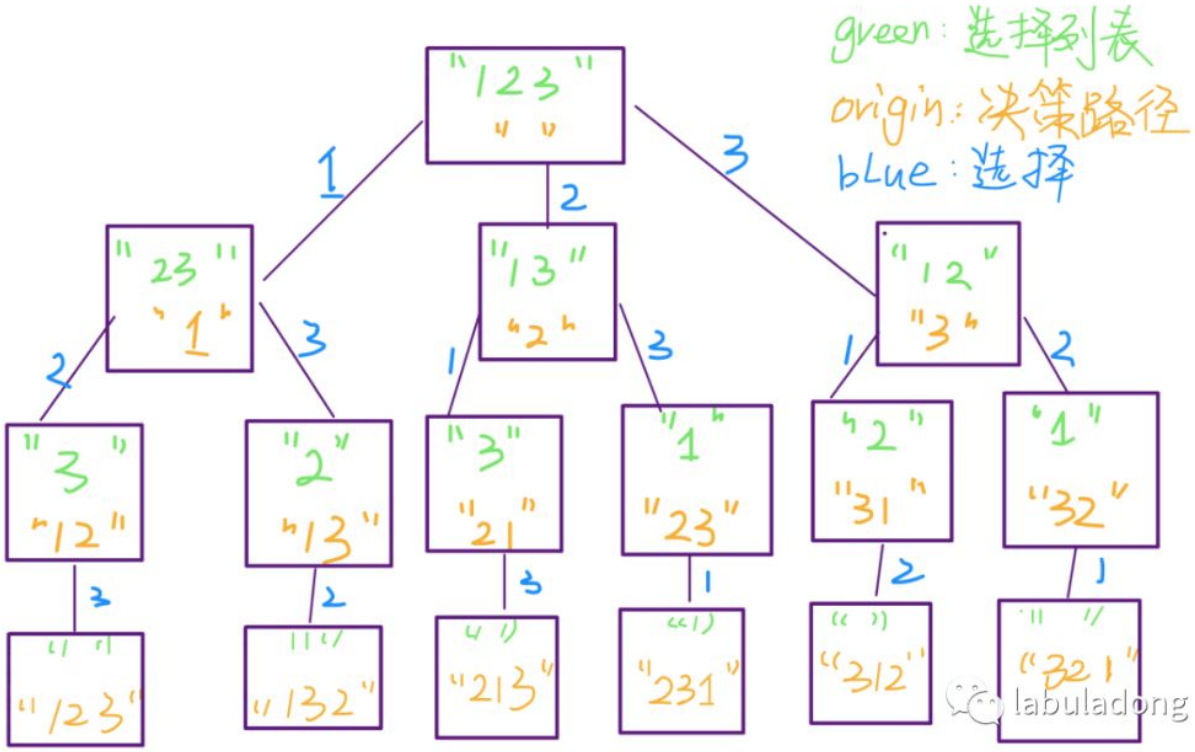
示例:

```
输入: [1,2,3]
输出:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

(截图来自 www.leetcode-cn.com)

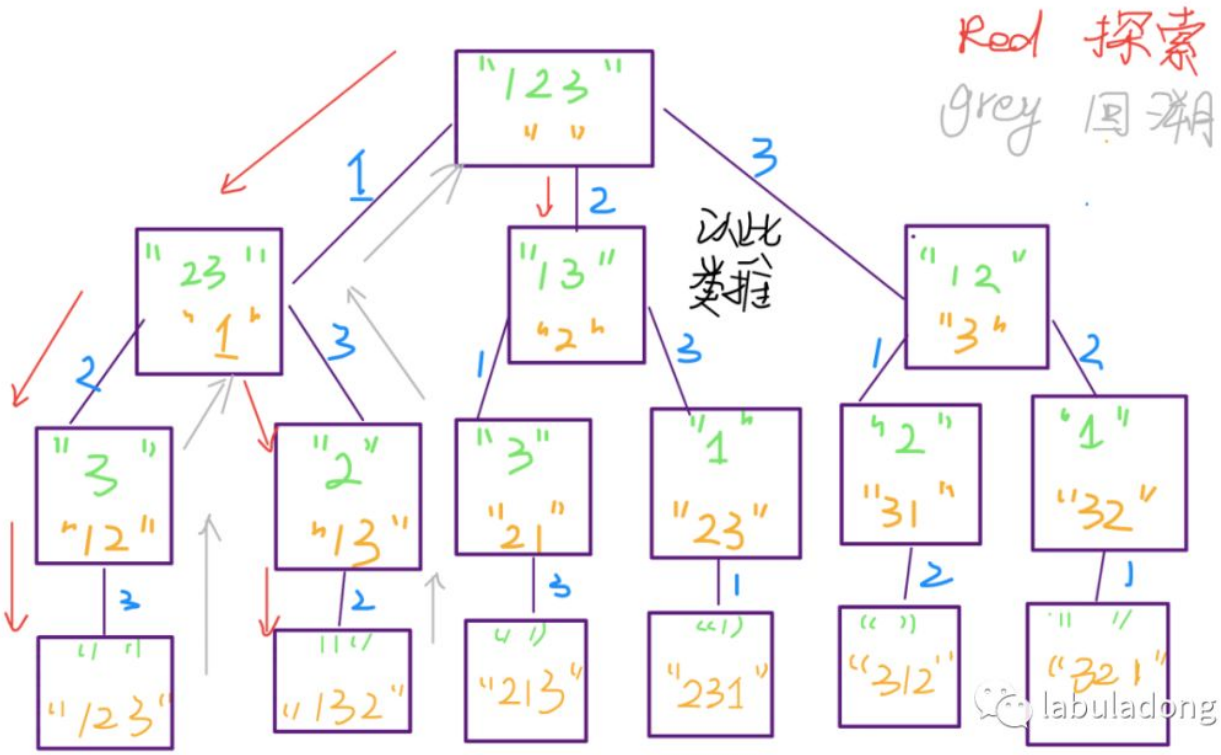
这个问题我们高中就做过，我们的思路是，先把第一个数固定为 1，然后全排列 2，3；再把第一个数固定为 2，全排列 1，3；再把第一个数固定为 3，全排列 1，2 。

这其实就是个决策的过程。转化成决策树表示一下：



可以发现每向下走一层就是在「选择列表」中挑一个「选择」加入「决策路径」，然后把这个选择从「选择列表」中删除（以免之后重复选择）；当一个决策分支探索完成后，我们就要向上回溯，要把该分支的「选择」从「决策列表」中取出，然后把这个「选择」重新加入「选择列表」（供其他的决策分支使用）。

以上，就是模板中 choose 和 unchoose 的过程，**choose 过程是向下探索，进行一选择；unchoose 过程是向上回溯，撤销刚才的选择。**

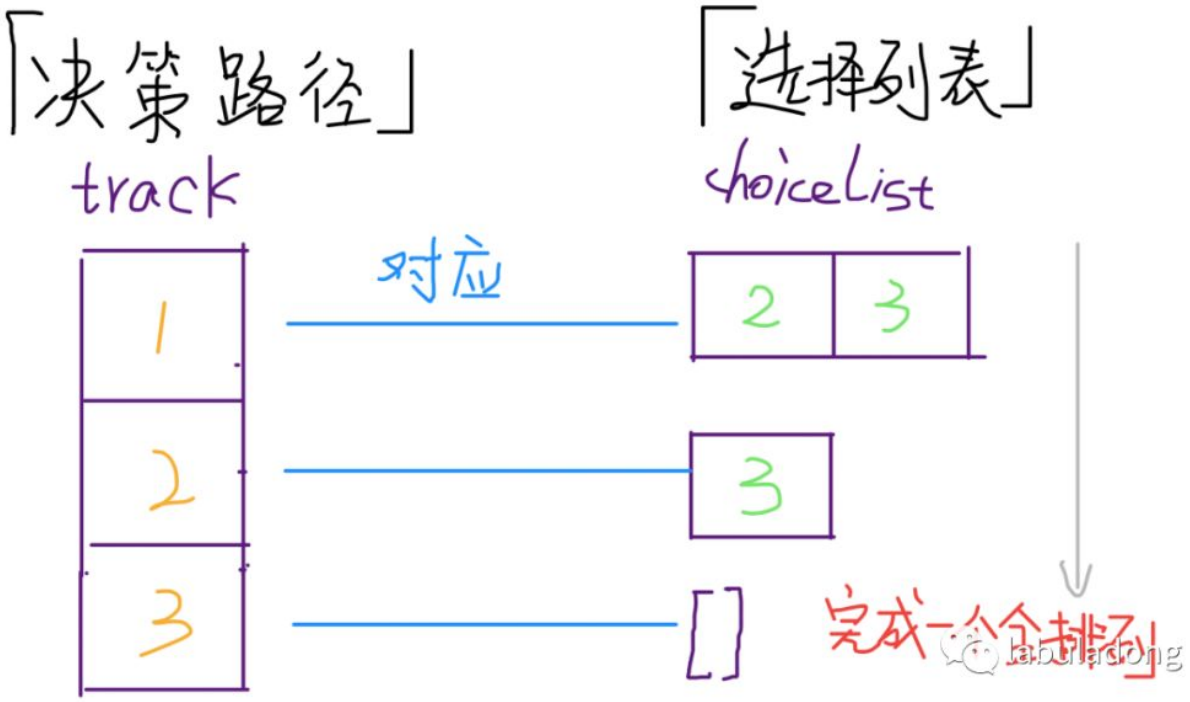


这下应该十分清楚了，现在我们可以针对全排列问题来具体化一下回溯算法模板：

然后，根据这个思路写代码，虽然不够漂亮，但是已经符合回溯算法的设计模式并且可以解决问题了：

PS：代码做成了图片形式，方便点开放大查看、保存，也便于左右滑动进行对照

```cpp
vector<vector<int>> permute(vector<int>& nums) {
    vector<vector<int>> ans;
    vector<int> track;
    backtrack(nums, track, ans);
    return ans;
}

// nums 数组可以理解为「选择列表」choiceList
void backtrack(
vector<int>& nums, vector<int>& track, vector<vector<int>>& ans) {
    if (0 == nums.size()) { // 选择列表已空，即完成一个全排列
        ans.push_back(track);
    } else {
        for (int i = 0; i < nums.size(); i++) {
            /* choose 过程 */
            int n = nums[i]; // 用变量存起来，等会 unchoose 要用
            track.push_back(n); // 加入决策路径的最后
            nums.erase(nums.begin() + i); // 擦除这个选择

            /* 进入下一步决策 */
            backtrack(nums, track, ans);

            /* unchoose 过程 */
            nums.insert(nums.begin() + i, n); // 再插回去
            track.pop_back(); // 移除这个选择
        }
    }
}
```

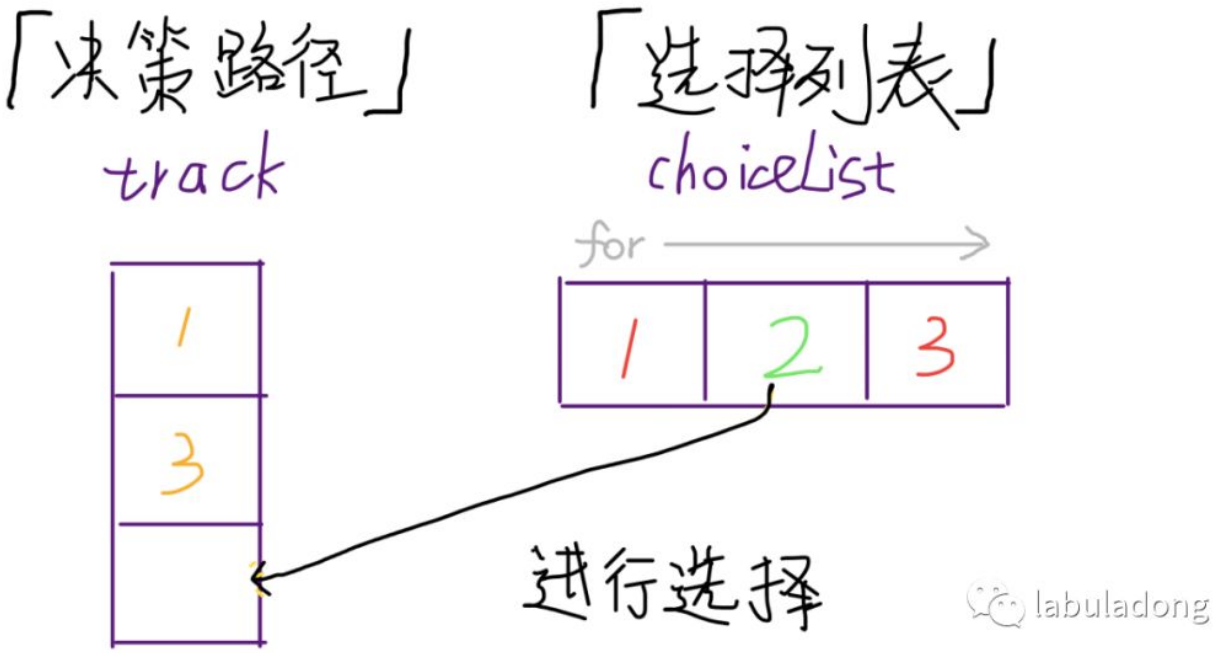


当然，对于数组来说，这样从数组中间擦除数据会产生数据搬移的时间复杂度，不过这属于编程细节，不是算法的问题，这里旨在突出 choose 和 unchoose 的操作。如果想优化也很简单，比如用一个额外的布尔数组记录某个元素是否能被选择；或者我们可以让索引 i 从大变小，即 for 循环从后往前遍历，这样每次擦除的元素都是最后一个，不会有数据搬移的消耗。

以下的 Java 代码是一种常用写法，避免了直接更改「选择列表」nums，而且看起来更简洁：

```java
public List<List<Integer>> permute(int[] nums) {
    List<List<Integer>> ans = new ArrayList<>();
    List<List<Integer>> track = new ArrayList<>()
    backtrack(nums, track, ans);
    return ans;
}

private void backtrack(
 int[] nums, List<Integer> track, List<List<Integer>> ans){
    if (track.size() == nums.length){
        ans.add(new ArrayList<>(track));
    } else{
        for(int i = 0; i < nums.length; i++){
            // 决策路径中已经存在的元素不能再次选择了
            if(track.contains(nums[i])) continue;
            /* choose 过程 */
            track.add(nums[i]); // 加到最后
            /* 进入下一步决策 */
            backtrack(nums, track, ans);
            /* unchoose 过程 */
            track.remove(track.size() - 1); // 移出最后
        }
    }
}
```

labuladong



labuladong

当然，还有更有技巧的代码，但那是细节问题，反而容易掩盖算法的结构。我们这里注重算法的设计，只要你明白了回溯算法就是遍历决策树，任何奇技淫巧随便你。

现在，你已经有能力解决经典问题：N 皇后问题（N Queens Problem）了。

二、N 皇后问题

这个问题大家应该都有耳闻：国际象棋的皇后可以横着，竖着，斜着进行攻击；给一个 N × N 的棋盘和 N 个皇后，如何放置这些皇后，才能使得任意两个都不能互相攻击到？

LeetCode 上也有这道题，难度 Hard，我直接截图吧：

n 皇后问题研究的是如何将 *n* 个皇后放置在 *n*×*n* 的棋盘上，并且使皇后彼此之间不能相互攻击。

One solution to the eight queens puzzle

上图为 8 皇后问题的一种解法。

给定一个整数 *n*，返回所有不同的 *n* 皇后问题的解决方案。

每一种解法包含一个明确的 *n* 皇后问题的棋子放置方案，该方案中 `'Q'` 和 `'.'` 分别代表了皇后和空位。

示例:

```
输入: 4
输出: [
 [".Q..",  // 解法 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // 解法 2
  "Q...",
  "...Q",
  ".Q.."]
]
解释: 4 皇后问题存在两个不同的解法。
```

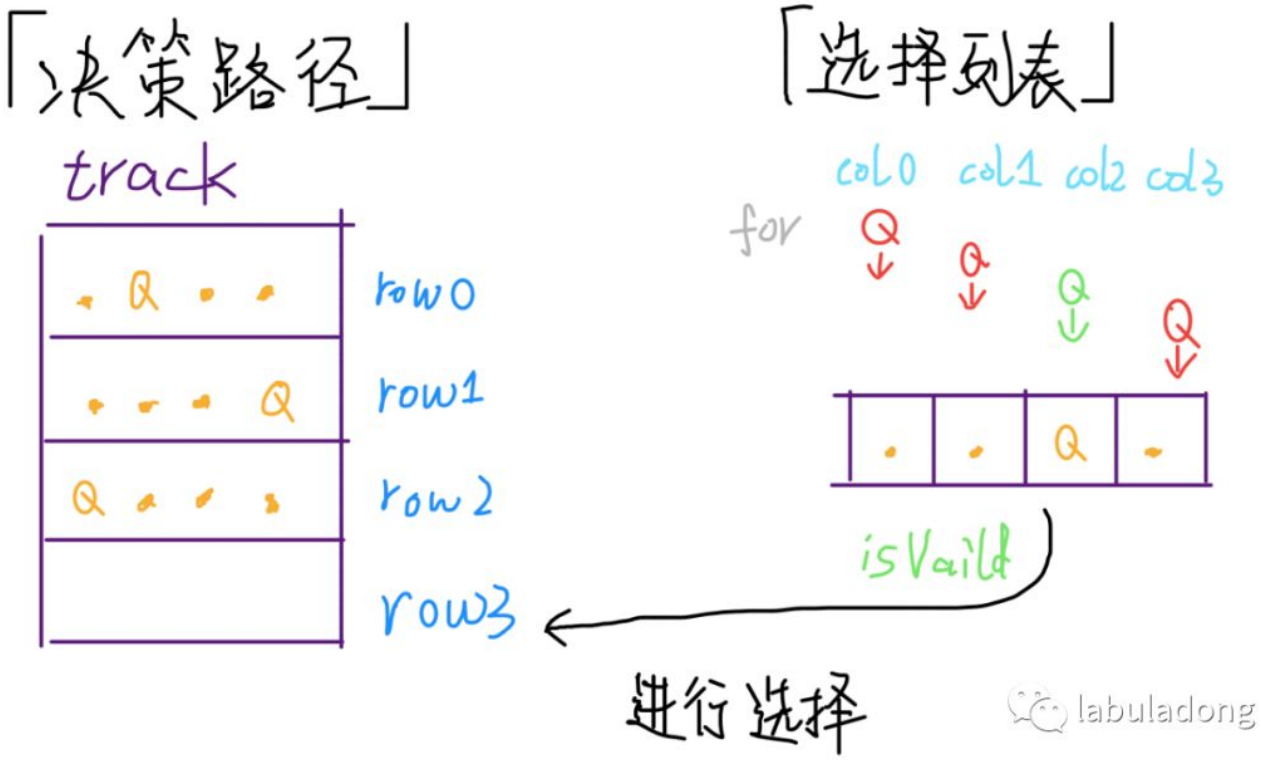
(截图来自 www.leetcode-cn.com)

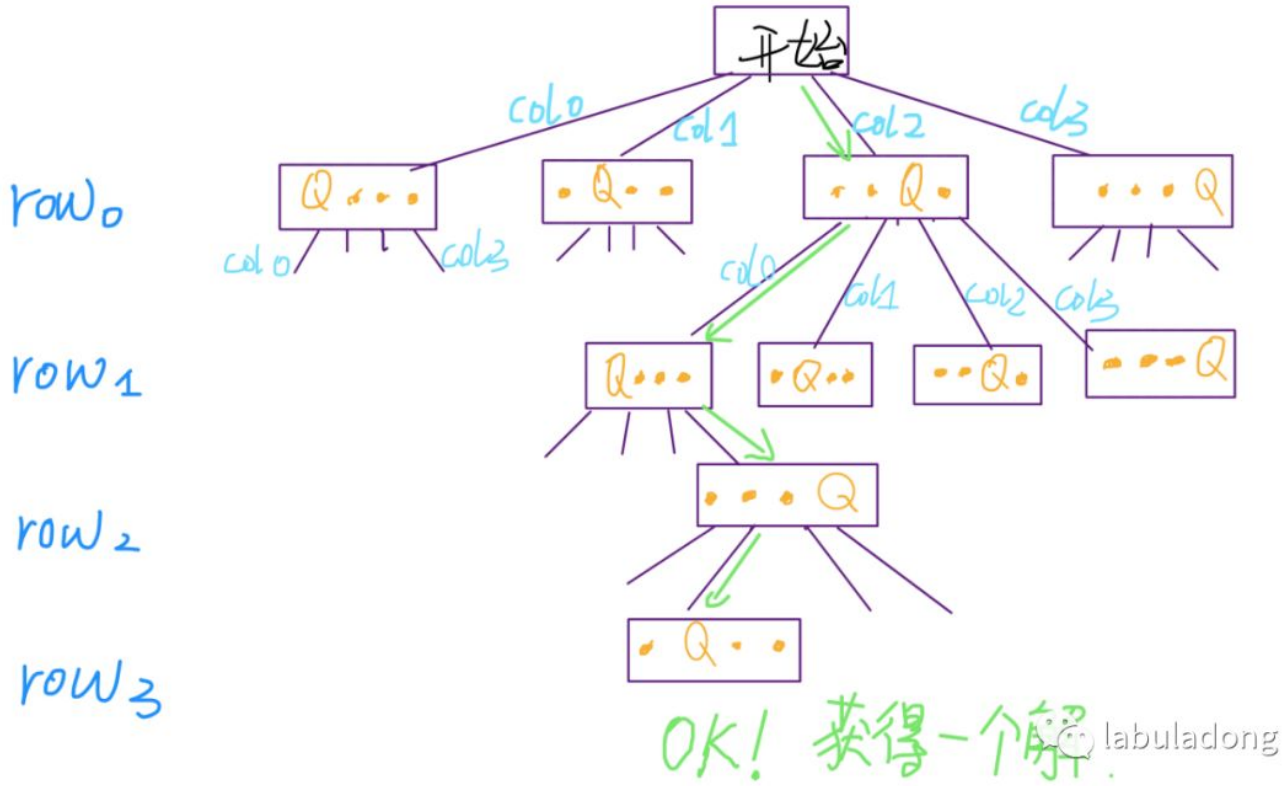
直接看代码，你会发现和上面的 Java 代码是完全相同的模板：

```
vector<vector<string>> solveNQueens(int n) {
    vector<string> board(n, string(n, '.'));
    vector<vector<string>> ans;
    backtrack(0, board, ans);
    return ans;
}

void backtrack(
  int row, vector<string>& board, vector<vector<string>>& ans) {
    if (row == board.size()) {
        ans.push_back(board);
    } else {
        // every column is a choice
        for (int col = 0; col < n; col++) {
            // 如果选择这个位置会被攻击，跳过
            if (!isValid(board, row, col)) continue;
            /* choose 过程 */
            board[row][col] = 'Q';
            /* 进行下一行的选择 */
            backtrack(row + 1, board, ans);
            /* unchoose 过程 */
            board[row][col] = '.';
        }
    }
}

// 判断 board[row][col] 是否可以放置 Q
bool isValid(vector<string>& board, int row, int col) {
    // 检查正上方
    for (int i = 0; i < n; i++)
        if (board[i][col] == 'Q') return false;
    // 检查右斜上方
    for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++)
        if (board[i][j] == 'Q') return false;
    // 检查左斜上方
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--)
        if (board[i][j] == 'Q') return false;
    // 不用检查下方，因为下方还没放置皇后
    return true;
}
```


「决策路径」：就是棋盘 board，每一步决策即 board 的每一行。row 变量记录着当前决策路径走到哪一步了。

「选择列表」：对于给定的一行（即决策的每一步），每一列都可能放置一个 'Q'，这就是所有的选择。代码中的 for 循环就在穷举「选择列表」判断是否可以放置皇后。





N 皇后问题不过如此，只要掌握了回溯算法模板，学会了把决策问题抽象成树的遍历，就能看出这种问题只不过是暴力试答案而已。其实再想想，解数独问题是不是也和 N 皇后问题差不多呢？

三、时间复杂度分析

分析一下回溯算法的时间复杂度吧。递归树的复杂度都是这样分析：总时间 = 递归树的节点总数 × 每个递归节点需要的时间。

全排列问题，节点总数等于 n + n*(n-1) + n*(n-2) ... * n!，总之不超过 O(n*n!)。

对于 Java 代码的那个解法，处理每个节点需要 O(n) 的时间，因为 track.contains(nums[i]) 这个操作要扫描数组。

所以全排列问题总时间不超过 O(n^2 * n!)。

N 皇后问题，节点总数为 n + n^2 + n^3 + ... + n^n，不超过 O(n^(n+1))。

处理每个节点需要向上扫描棋盘以免皇后互相攻击，需要 O(n) 时间。

所以 N 皇后问题总时间不超过 O(n^(n+2))。

可见，回溯算法的复杂度是极其高的，甚至比指数级还高，因为树形结构注定了复杂度爆炸的结局。

你可能会问，之前动态规划一文中讲到的优化方法不是专门优化树形结构的时间复杂度的吗？不是能降维打击吗？

但这里无法做任何优化。再强调一遍，计算机做事的策略就是穷举。动态规划的一大特征：重叠子问题，可以让那类问题“聪明地穷举”。但回顾一下回溯算法的决策树，根本不存在重叠子问题，优化无从谈起。

就好比让你找出数组中最大的元素，你起码得把数组全部遍历一遍吧，还能再优化吗？不可能的。

四、最后总结

无论如何，能看到这里，给你鼓掌。现在的你，掌握了回溯算法模板，学会了树形搜索的抽象思维，递归功力大涨，解决了两道难度较大的经典问题，并且再次深入体会了计算机做事的逻辑。

「选择列表」+「决策路径」结合树的遍历，简单的组合却能解决一大类问题，这就是算法框架的力量。让我们最后重温一下算法的魅力吧！

```
def backtrack(choiceList, track, answer):
  if track is OK:
    answer.add(track)
  else:
    for choice in choiceList:
      if choice is not OK: continue
      # choose 过程
      backtrack(choiceList, track, answer)
      # unchoose 过程
```

PS：前一阵微信升级，把整体字体调小了，我其实觉得眼睛很不习惯。最近我一直在摸索排版方式，尤其是代码的插入：代码字体大了要左右滑动不好查看，字体小了又看起来不舒服。

这次我把一部分较宽的代码排版做了优化，甚至尝试将部分代码块直接做成图片上传，方便读者点开放大查看以及保存，自认为这样的阅读体验最佳。你觉得这个办法如何？可以在留言板发表看法。

点击这里进入留言板 ~

你的点赞是对我的鼓励。