一套代码解决 Combination Sum 系列问题（LeetCode 39/40/216）

原创 nettee 面向大象编程 3月23日

来自专辑

LeetCode 例题精讲

本文讲解的问题为 Combination Sum 系列问题，共四道：

- 39. Combination Sum（Medium）
- 40. Combination Sum II（Medium）
- 216. Combination Sum III（Medium）
- 377. Combination Sum IV（Medium）

我们的系列文章已经有三期回溯法的内容了。前面的两篇文章中，我们详细讲解了回溯法的子集问题、排列问题、组合问题这些经典问题，讲解了回溯法的候选集合概念以及剪枝方法：

- **08 排列组合问题：回溯法的候选集合**
- **09 排列组合问题再探：回溯法的去重策略**

回溯法的基本方法讲解就到此为止了。本篇文章是习题篇，以 LeetCode 上的 Combination Sum 系列问题为例，用一套代码解决系列问题，对每一题根据题目条件的不同做代码的微调。在回溯法问题中，很多细微条件的不同就会导致代码思路上出现变化。对于 Combination Sum 系列问题，我们要处理的条件有：

- 有放回 vs 无放回
- 有重复元素 vs 无重复元素
- 是否限制组合数量

下面将一一讲解如何微调代码来处理这些条件。

Combination Sum 通用解法

我们先写出一套通用的 Combination Sum 问题解法，后面再根据具体的题目做代码上的调整。

标准版 Combination Sum 问题（LeetCode 上的系列每一道都和它有细微差异）：

> 给定一个数组 `candidates` 和一个目标数 `target`，找出 `candidates` 中所有可以使数字之和为 `target` 的组合。
>
> 注意：
>
> - 数组 `candidates` 中无重复元素。
> - 所有数字（包括 target）都是正整数。
> - 组合是无序的，不同顺序的排列应视为同一组合。

Combination Sum 问题和组合（Combination）问题非常相似。掌握了组合问题的解法（可以回顾这篇文章），就可以很轻松地写出 Combination Sum 的代码。在回溯函数中添加一个参数 `target`，每次选择了一个元素 `x`，就将 `target - x` 传入下一个回溯函数中。当 `target` 为 0 时，我们就找到了一个和为 `target` 的组合。以下是题解代码：

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    Deque<Integer> current = new ArrayDeque<>();
    List<List<Integer>> res = new ArrayList<>();
    backtrack(candidates, 0, target, current, res);
    return res;
}

// 候选集合 candidates[m..N)
void backtrack(int[] candidates, int m, int target, Deque<Integer> current, List<List<Integer>> res
    if (target < 0) {
        return;
    } elseif (target == 0) {
        res.add(new ArrayList<>(current));
        return;
    }

    // 和 Combinations 问题类似，需要升序
    for (int i = m; i < candidates.length; i++) {
        // 选择数字 candidates[i]
        current.addLast(candidates[i]);
        // 元素 candidates[m..i) 均失效
        backtrack(candidates, i+1, target - candidates[i], current, res);
        // 撤销选择
        current.removeLast();
```

```
    }
}
```

需要注意的是，Combination Sum 并不限制组合中元素的个数，所以回溯法代码中并没有 k 这个参数。**请记住这段代码，后面的代码都将在这段代码的基础上进行微调。**

问题的各种变种

下面，我们来分别看看 LeetCode 上四道系列题目是如何在标准 Combination Sum 问题上做变化的，又该如何具体解决。

有放回 vs. 无放回

> 题目：**LeetCode 39 - Combination Sum**[1]（Medium）
>
> 变化之处：数组中的元素可以无限制重复选取。

所谓的可以重复选取，其实可以理解成**有放回**的抽样。

在标准版问题中，数组元素不可以重复选择，是无放回的。对于数组 `candidates` ，如果当前选择了元素 `candidates[i]` ，下一轮的候选集合就会变成 `candidates[i+1..N)` 。

对于有放回的变种问题，我们可以将当前元素 `candidates[i]` “放回”候选集合中，下一轮的候选集合变成 `candidates[i..N)` 。这样， `candidates[i]` 在后面仍然有可能被选中。

我们只需要在原先的代码中调整递归调用的参数，让下一轮的候选集合由 `candidates[i+1..N)` 变成 `candidates[i..N)` 。题解代码如下：

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    Deque<Integer> current = new ArrayDeque<>();
    List<List<Integer>> res = new ArrayList<>();
    backtrack(candidates, 0, target, current, res);
    return res;
}

// 候选集合 candidates[m..N)
void backtrack(int[] candidates, int m, int target, Deque<Integer> current, List<List<Integer>> res
```

```
    if (target < 0) {
        return;
    } elseif (target == 0) {
        res.add(new ArrayList<>(current));
        return;
    }

    for (int i = m; i < candidates.length; i++) {
        // 选择数字 candidates[i]
        current.addLast(candidates[i]);
        // 代码调整处：递归调用参数
        // 递归调用传递 i 而不是原先的 i+1
        // 这样 candidates[i] 选完后仍然在候选集合里，后续仍然可以再选
        backtrack(candidates, i, target - candidates[i], current, res);
        // 撤销选择
        current.removeLast();
    }
}
```

有重复元素 vs. 无重复元素

> 题目：**LeetCode 40 - Combination Sum II**[2]（Medium）
>
> 变化之处：数组中可能含有重复元素。

当数组中含有重复元素的时候，我们需要去除重复的结果。例如，数组为 $[1,2,2]$，`target` 为 3。如果把数组中的两个 2 记为 2_1 和 2_2 的话，$[1,2_1]$ 和 $[1,2_2]$ 就是两个重复的结果，只能保留一个。

我们在前一篇文章中已经讲解过了回溯法中有重复元素的问题，并给出了子集、排列、组合问题的去重策略（要回顾前一篇文章，请点击这里）。Combination Sum 问题的去重策略可以直接参考组合问题的去重策略：

1. 在回溯之前，预先将数组中的元素排序，保证相等的元素是相邻的；
2. 当有多个相等元素的时候，只能选择其中第一个。

我们只需要在原先的代码中调整候选元素的遍历。如果 `candidates[i]` 与前一个元素相等，说明不是相等元素中的第一个，则跳过该元素。题解代码如下：

```
public List<List<Integer>> combinationSum2(int[] candidates, int target) {
    Arrays.sort(candidates);
    Deque<Integer> current = new ArrayDeque<>();
    List<List<Integer>> res = new ArrayList<>();
    backtrack(candidates, 0, target, current, res);
    return res;
}

// 候选集合 candidates[m..N)
void backtrack(int[] candidates, int m, int target, Deque<Integer> current, List<List<Integer>> res
    if (target < 0) {
        return;
    } elseif (target == 0) {
        res.add(new ArrayList<>(current));
    }

    for (int i = m; i < candidates.length; i++) {
        // 代码调整处：候选集合遍历
        if (i > m && candidates[i] == candidates[i-1]) {
            // 如果 candidates[i] 与前一个元素相等，说明不是相等元素中的第一个，跳过。
            continue;
        }
        // 选择数字 candidates[i]
        current.addLast(candidates[i]);
        // 元素 candidates[m..i) 均失效
        backtrack(candidates, i+1, target - candidates[i], current, res);
        // 撤销选择
        current.removeLast();
    }
}
```

限制 k-combination

> **LeetCode 216 - Combination Sum III**[3]（Medium）
>
> 变化之处：限制结果只能是 k-combination，即由 k 个元素相加得到 `target` 。

前面几道题中，组合中元素的数量没有限制。而这道题限制了组合必须正好是 k 个数。这其实回归到了正常的 “n 中取 k” 的组合问题，我们需要在回溯函数中添加参数 k，当组合的大小达到 k，并且和为 target 的时候才得到一个结果。

这道题的另一个变化之处是没有输入数组，直接变成在 1 到 9 的数字中选择。这个变化比较好应对，只需要稍微调整递归参数即可。

题解代码如下：

```
public List<List<Integer>> combinationSum3(int k, int n) {
    Deque<Integer> current = new ArrayDeque<>();
    List<List<Integer>> res = new ArrayList<>();
    backtrack(k, 1, n, current, res);
    return res;
}

// 候选集合：整数 [m..9]
// 代码调整处：加入参数 k
void backtrack(int k, int m, int target, Deque<Integer> current, List<List<Integer>> res) {
    if (target < 0) {
        return;
    } elseif (target == 0) {
        // 代码调整处：已选集合达到 k 个元素才收集结果
        if (current.size() == k) {
            res.add(new ArrayList<>(current));
        }
        return;
    }
    if (current.size() > k) {
        return;
    }

    // 从候选集合中选择
    for (int i = m; i <= 9; i++) {
        // 选择数字 i
        current.addLast(i);
        // 数字 [m..i) 均失效
        backtrack(k, i+1, target - i, current, res);
        // 撤销选择
        current.removeLast();
    }
}
```

回溯法 vs. 动态规划

> 题目：**LeetCode 377 - Combination Sum IV**[4]（Medium）
>
> 变化之处：……我怎么看已经变成另一道题了呢？

这道题其实非常有迷惑性。我们重新读一下题：

> 给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数。
>
> 示例： `nums = [1, 2, 3]` ， `target = 4` 。所有可能的组合为：
>
> ```
> (1, 1, 1, 1)
> (1, 1, 2)
> (1, 2, 1)
> (1, 3)
> (2, 1, 1)
> (2, 2)
> (3, 1)
> ```
>
> 请注意，顺序不同的序列被视作不同的组合。因此输出为 7。

“顺序不同的序列被视作不同的组合”，这已经脱离了组合的概念，变成了排列！所以说，这道题实际上是 Permutation Sum 问题，而不是 Combination Sum。既然这样，我们就不能再用 Combination Sum 的代码去套了。

前面我们用一个套路做完了 Combination Sum 的三道题。这第四道题，就是 LeetCode 让你上钩的。这道题根本不是一个回溯法题目，用回溯法做的话，会超出时间限制（TLE）。所以说，我们做题不能形成思维定式，要仔细思考题目的条件。

明白这一点之后，我们会发现，这道题是一个不折不扣的动态规划题目。再定睛一看，这道题实际上就是换零钱问题嘛：

> **LeetCode 322 - Coin Change**[5]（Medium）
>
> 给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 `-1` 。你可以认为每种硬币的数量是无限的。
>
> 示例：

```
输入: coins = [1, 2, 5], amount = 11
输出: 3
解释: 11 = 5 + 5 + 1
```

只不过是“数组”变成了“硬币”， `target` 变成了 `amount` 而已。那么这道题的解法，参考换零钱问题即可。

总结

Combination Sum 系列问题是回溯法非常好的练习题，既能够巩固我们之前讲解的候选元素、剪枝的概念，又能体会题目的不同变种给代码带来的影响。回溯法问题千变万化，题目的细节条件稍有不同就变成了另一道题（甚至可能从回溯法变成了动态规划，例如系列的第四题）。因此我们在做题的时候，一定要推敲题目的细节。在面试中遇到回溯法题目，也要和面试官沟通好题目的各个条件再开始写代码。

另外预告一下，回溯法问题的讲解暂时告一段落。接下来会写几篇二叉树、DFS/BFS 相关文章，敬请期待~

参考资料

[1] LeetCode 39 - Combination Sum: *https://leetcode.com/problems/combination-sum/*

[2] LeetCode 40 - Combination Sum II: *https://leetcode.com/problems/combination-sum-ii/*

[3] LeetCode 216 - Combination Sum III: *https://leetcode.com/problems/combination-sum-iii/*

[4] LeetCode 377 - Combination Sum IV: *https://leetcode.com/problems/combination-sum-iv/*

[5] LeetCode 322 - Coin Change: *https://leetcode.com/problems/coin-change/*