0052. N 皇后 II

ITCharge 大约 4 分钟

- 标签：回溯
- 难度：困难

题目链接

- 0052. N 皇后 II - 力扣

题目大意

描述：给定一个整数 `n` 。

要求：返回「 `n` 皇后问题」不同解决方案的数量。

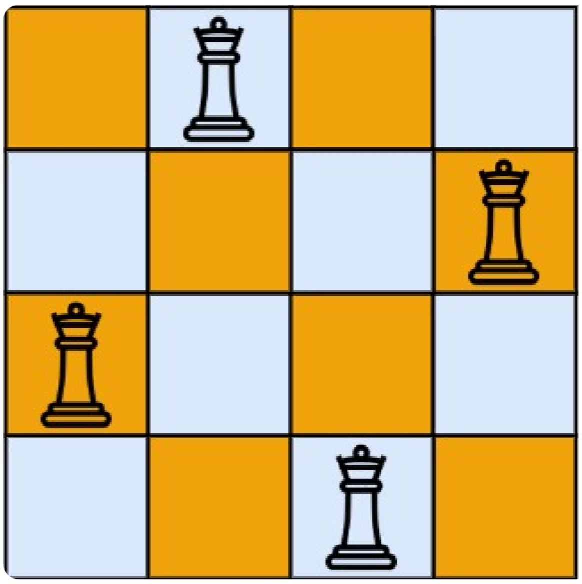
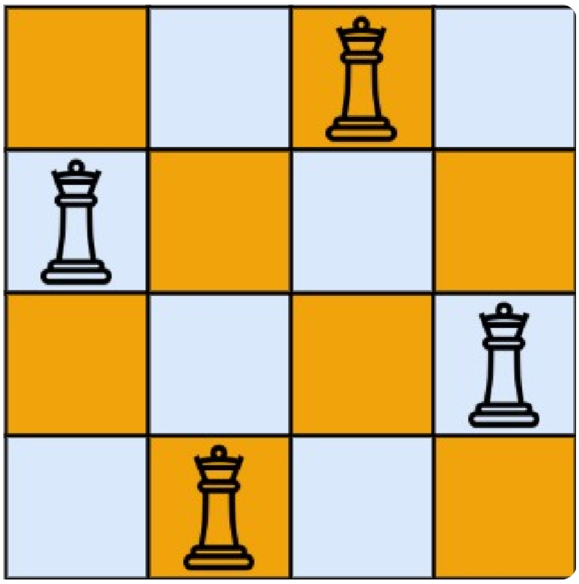
说明：

- **n 皇后问题**：将 `n` 个皇后放置在 `n` 的棋盘上，并且使得皇后彼此之间不能攻击。
- **皇后彼此不能相互攻击**：指的是任何两个皇后都不能处于同一条横线、纵线或者斜线上。
- $1 \le n \le 9$。

示例：

- 示例 1：

```py
输入：n = 4
输出：2
解释：如下图所示，4 皇后问题存在两个不同的解法。
```

解题思路

思路 1：回溯算法

和「51. N 皇后 - 力扣 」做法一致。区别在于「51. N 皇后 - 力扣 」需要返回所有解决方案，而这道题只需要得到所有解决方案的数量即可。下面来说一下这道题的解题思路。

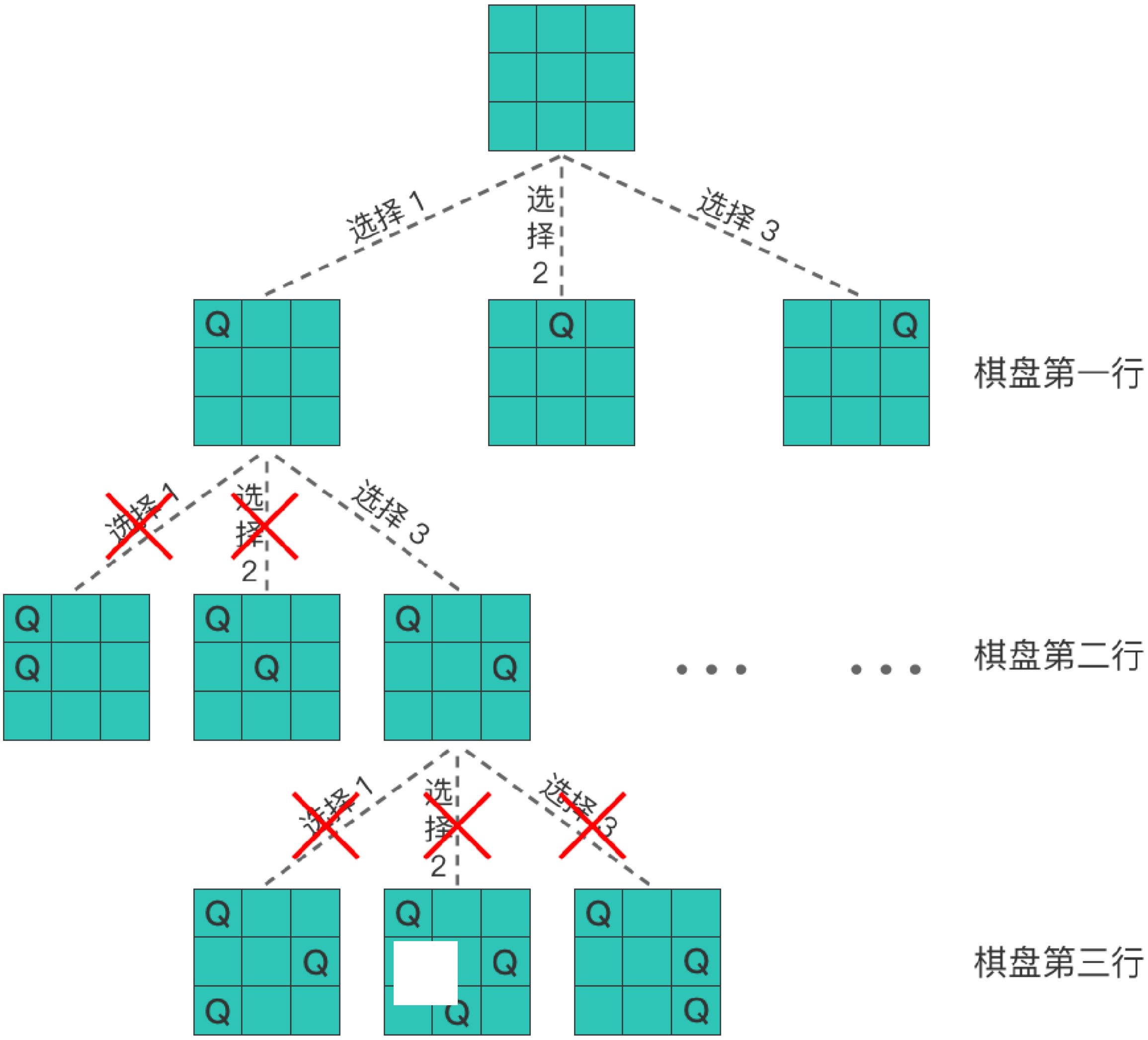
我们可以按照行序来放置皇后，也就是先放第一行，再放第二行 …… 一直放到最后一行。

对于 `n * n` 的棋盘来说，每一行有 `n` 列，也就有 `n` 种放法可供选择。我们可以尝试选择其中一列，查看是否与之前放置的皇后有冲突，如果没有冲突，则继续在下一行放置皇后。依次类推，直到放置完所有皇后，并且都不发生冲突时，就得到了一个合理的解。

并且在放置完之后，通过回溯的方式尝试其他可能的分支。

下面我们根据回溯算法三步走，写出对应的回溯算法。

1. **明确所有选择**：根据棋盘中当前行的所有列位置上是否选择放置皇后，画出决策树，如下图所示。



2. **明确终止条件**：
   - 当遍历到决策树的叶子节点时，就终止了。也就是在最后一行放置完皇后后时，递归终止。
3. **将决策树和终止条件翻译成代码：**
   1. 定义回溯函数：
      - 首先我们先使用一个 `n * n` 大小的二维矩阵 `chessboard` 来表示当前棋盘，`chessboard` 中的字符 `Q` 代表皇后，`.` 代表空位，初始都为 `.`。
      - 然后定义回溯函数 `backtrack(chessboard, row):` 函数的传入参数是 `chessboard`（棋盘数组）和 `row`（代表当前正在考虑放置第 `row` 行皇后），全局变量是 `ans`（所有可行方案的数量）。
      - `backtrack(chessboard, row):` 函数代表的含义是：在放置好第 `row` 行皇后的情况下，递归放置剩下行的皇后。

2. 书写回溯函数主体（给出选择元素、递归搜索、撤销选择部分）。

- 枚举出当前行所有的列。对于每一列位置：
    - 约束条件：定义一个判断方法，先判断一下当前位置是否与之前棋盘上放置的皇后发生冲突，如果不发生冲突则继续放置，否则则继续向后遍历判断。
    - 选择元素：选择 `row, col` 位置放置皇后，将其棋盘对应位置设置为 `Q` 。
    - 递归搜索：在该位置放置皇后的情况下，继续递归考虑下一行。
    - 撤销选择：将棋盘上 `row, col` 位置设置为 `.` 。

思路 1：代码

```py
class Solution:
    # 判断当前位置 row, col 是否与之前放置的皇后发生冲突
    def isValid(self, n: int, row: int, col: int, chessboard: List[List[str]]):
        for i in range(row):
            if chessboard[i][col] == 'Q':
                return False

        i, j = row - 1, col - 1
        while i >= 0 and j >= 0:
            if chessboard[i][j]    'Q':
                return False
            i -= 1
            j -= 1
        i, j = row - 1, col + 1
        while i >= 0 and j < n:
            if chessboard[i][j] == 'Q':
                return False
            i -= 1
            j += 1

        return True

    def totalNQueens(self, n: int) -> int:
        chessboard = [['.' for _ in range(n)] for _ in range(n)]    # 棋盘初始化

        ans = 0
        def backtrack(chessboard: List[List[str]], row: int):
# 正在考虑放置第 row 行的皇后
            if row == n:                                    # 遇到终止条件
                nonlocal ans
                ans += 1
```

```
                return

            for col in range(n):                                # 枚举可放置皇后的列
                if self.isValid(n, row, col, chessboard):       # 如果该位置与之前放置的皇后不发生冲突
                    chessboard[row][col] = 'Q'                  # 选择 row, col 位置放置皇后
                    backtrack(chessboard, row + 1)              # 递归放置 row + 1 行之后的皇后
                    chessboard[row][col] = '.'                  # 撤销选择 row, col 位置

        backtrack(chessboard, 0)

        return ans
```

思路 1：复杂度分析

- **时间复杂度**：$O(n!)$，其中 n 是皇后数量。
- **空间复杂度**：$O(n^2)$，其中 n 是皇后数量。递归调用层数不会超过 n，每个棋盘的空间复杂度为 $O(n^2)$，所以空间复杂度为 $O(n^2)$。