回溯-进阶篇

该部分题目相对于排列组合问题增加了难度，在模板基础上需要增加一个额外的判断方法，不是固定的循环而是新思路，或者需要剪枝操作。但还是万变不离模板，稍作改变即可。

在 LeetCode 中回溯法-进阶篇包含的题目有：

1. Palindrome Partitioning
2. Restore IP Addresses
3. Sudoku Solver
4. N-Queens
5. N-Queens II
6. Generate Parentheses
7. Word Search
8. Word Break II

1. Palindrome Partitioning，这道题是求一个字符串中回文子串的切割，并且输出切割结果，其实是Word Break II和Longest Palindromic Substring结合，该做的我们都做过了。首先我们根据Longest Palindromic Substring中的方法建立一个字典，得到字符串中的任意子串是不是回文串的字典。接下来就跟Word Break II一样，根据字典的结果进行切割，然后按照循环处理递归子问题的方法，如果当前的子串满足回文条件，就递归处理字符串剩下的子串。如果到达终点就返回当前结果。算法的复杂度跟Word Break II一样，取决于结果的数量，最坏情况是指数量级的。

```
// Solution 1 – Classic Model
public ArrayList<ArrayList<String>> partition(String s) {
    ArrayList<ArrayList<String>> res = new ArrayList<ArrayList<String>>();
    if(s == null || s.length() == 0){
        return res;
    }
    ArrayList<String> item = new ArrayList<String>();
    boolean[][] dict = getDict(s);
    helper(s, 0, item, res, dict);
    return res;
}
```

```
    private void helper(String s, int start, ArrayList<String> item, ArrayList<ArrayList<String>> res, boolean[][] dict){
        if(start == s.length()){
            res.add(new ArrayList<String>(item));
            return;
        }
        for(int i = start; i < s.length(); i++){
            if(dict[start][i]){
                item.add(s.substring(start, i + 1));
                helper(s, i + 1, item, res, dict);
                item.remove(item.size() - 1);
            }
        }
    }

    private boolean[][] getDict(String s){
        boolean[][] dict = new boolean[s.length()][s.length()];
        for(int i = s.length() - 1; i >= 0; i--){
            for(int j = i; j < s.length(); j++){
                if(s.charAt(i) == s.charAt(j) && (j - i <= 2 || dict[i + 1][j - 1])){
                    dict[i][j] = true;
                }
            }
        }
        return dict;
    }
```

做题时的感悟:

1. 思维要灵活，不要太死板。这里的 dict 不一定非要是 Hashset，像回文字符串中用 boolean[][]就可以，而且更加简洁。判断的时候，只要判断 if(dict[start][i])即可。

2. 再次提醒，如果要恢复现场的话，在加入结果集的时候一定要新建一个数组：res.add(new ArrayList<String>(item));

2. Restore IP Addresses，这道题的基本思路就是取出一个合法的数字，作为IP地址的一项，然后递归处理剩下的项。可以想象出一颗树，每个结点有三个可能的分支（因为范围是0-255，所以可以由一位两位或者三位组成）。并且这里树的层数不会超过四层，因为IP地址由四段组成，到了之后我们就没必要再递归下去了。这里除了上述的结束条件外，另一个就是

字符串读完了。可以看出不像平常的NP问题那样，时间复杂度取决于输入的规模，是指数量级的，这棵树的规模是固定的，因为他的分支是有限制的四段，所以这道题并不是NP问题。实现中需要一个判断数字是否为合法ip地址的一项的函数，首先要在0-255之间，其次前面字符不能是0。剩下的就是套路了，使用模板即可。

```
    // Solution 1 – Classic Model
    public ArrayList<String> restoreIpAddresses(String s) {
        ArrayList<String> res = new ArrayList<String>();
        if(s == null || s.length() < 4 || s.length() > 12){
            return res;
        }
        ArrayList<String> items = new ArrayList<String>();
        helper(s, 0, items, res);
        return res;
    }

    private void helper(String s, int start, ArrayList<String> items, ArrayList<String> res){
        if(items.size() == 4){
            if(start != s.length()){
                return;
            }
            StringBuilder IPAddress = new StringBuilder();
            for(String str : items){
                IPAddress.append(str).append(".");
            }
            IPAddress.deleteCharAt(IPAddress.length() - 1);
            res.add(IPAddress.toString());
            return;
        }
        for(int i = start; i < start + 3 && i < s.length(); i++){
            String str = s.substring(start, i + 1);
            if(isValid(str)){
                items.add(str);
                helper(s, i + 1, items, res);
                items.remove(items.size() - 1);
            }
        }
    }
```

```
private boolean isValid(String s){
    if(s == null || s.length() > 3){
        return false;
    }
    if(s.charAt(0) == '0' && s.length() > 1){
        return false;
    }
    int num = Integer.parseInt(s);
    return num >= 0 && num <= 255;
}
```

3. Sudoku Solver，这道题的思路就是对于每个格子，带入不同的9个数，然后判合法，如果成立就递归继续，结束后把数字设回空。依然套用模板。判合法可以用Valid Sudoku做为subroutine，但是其实在这里因为每次进入时已经保证之前的board不会冲突，所以不需要判断整个盘，只需要看当前加入的数字和之前是否冲突就可以，这样可以大大提高运行效率，毕竟判合法在程序中被多次调用。

```
// Solution 1 – Classic Model
public void solveSudoku(char[][] board) {
    if(board == null || board.length != 9 || board[0].length != 9){
        return;
    }
    helper(board, 0, 0);
}

private boolean helper(char[][] board, int i, int j){
    if(i == 9){
        return true;
    }
    if(j == 9){
        return helper(board, i + 1, 0);
    }
    if(board[i][j] == '.'){
        for(int k = 1; k <= 9; k++){
            board[i][j] = (char)(k + '0');
            if(isValid(board, i, j)){
                if(helper(board, i, j + 1)){
                    return true;
                }
```

```
                }
                board[i][j] = '.';
            }
        } else {
            return helper(board, i, j + 1);
        }
        return false;
    }

    private boolean isValid(char[][] board, int i, int j){
        for(int k = 0; k < 9; k++){
            if(k != i && board[k][j] == board[i][j]){
                return false;
            }
        }
        for(int k = 0; k < 9; k++){
            if(k != j && board[i][k] == board[i][j]){
                return false;
            }
        }
        for(int row = i / 3 * 3; row < i / 3 * 3 + 3; row++){
            for(int col = j / 3 * 3; col < j / 3 * 3 + 3; col++){
                if((row != i || col != j) && board[row][col] == board[i][j]){
                    return false;
                }
            }
        }
        return true;
    }
```

做题时的感悟:

1. 因为进入时已经保证之前的 board 不会冲突，所以不需要检查整个棋盘，只需要检查当前字符与之前是否冲突即可。分别检查当前行，当前列以及当前小 block 即可。

2. 注意，这里的 helper 方法返回了一个 boolean 值。首先，很多时候写代码返回一个 boolean 变量来反映函数的执行成功与否是可行的。而且，这道题中必须要返回一个 boolean 变量，因为如果找到了合法解，就可以直接 return true 以跳过回溯的过程，否则又会打乱棋盘。

4. N-Queens，该类题目主要思想就是一句话：用一个循环递归处理子问题。

这个问题中，在每一层递归函数中，我们用一个循环把一个皇后填入对应行的某一列中，如果当前棋盘合法，我们就递归处理下一行，找到正确的棋盘我们就存储到结果集里面。这种题目都是使用这个套路，就是用一个循环去枚举当前所有情况，然后把元素加入，递归，再把元素移除。这道题目中不用移除的原因是我们用一个一维数组去存皇后在对应行的哪一列，因为一行只能有一个皇后，如果二维数组，那么就需要把那一行那一列在递归结束后设回没有皇后，所以道理是一样的。

这道题最后一个细节就是怎么实现检查当前棋盘合法性的问题，因为除了刚加进来的那个皇后，前面都是合法的，我们只需要检查当前行和前面行是否冲突即可。检查是否同列很简单，检查对角线就是行的差和列的差的绝对值不要相等就可以。

代码如下：

```
// Solution 1 - Classic Model
public ArrayList<String[]> solveNQueens(int n) {
    ArrayList<String[]> res = new ArrayList<String[]>();
    int[] column = new int[n];
    helper(0, n, column, res);
    return res;
}

private void helper(int row, int n, int[] column, ArrayList<String[]> res){
    if(row == n){
        String[] board = new String[n];
        for(int i = 0; i < n; i++){
            StringBuilder item = new StringBuilder();
            for(int j = 0; j < n; j++){
                if(column[i] == j){
                    item.append("Q");
                } else {
                    item.append(".");
                }
            }
            board[i] = item.toString();
        }
        res.add(board);
        return;
    }
```

```
        for(int i = 0; i < n; i++){
            column[row] = i;
            if(isValid(row, column)){
                helper(row + 1, n, column, res);
            }
        }
    }

    private boolean isValid(int row, int[] column){
        for(int i = 0; i < row; i++){
            if(column[i] == column[row] || Math.abs(column[i] - column[row]) == row - i){
                return false;
            }
        }
        return true;
    }
```

做题时的感悟：

当row == n时，即找到合适结果并存储结果之后，我们要记得return；否则会继续进入下面的for循环，此时就会导致column数组越界。

5. N-Queens II，这道题跟N-Queens算法是完全一样的，只是把输出从原来的结果集变为返回结果数量而已。算法的时间复杂度仍然是指数量级的，空间复杂度是O(n)。

代码如下：

```
    public int totalNQueens(int n) {
        ArrayList<Integer> res = new ArrayList<Integer>();
        res.add(0);
        int[] column = new int[n];
        helper(0, n, column, res);
        return res.get(0);
    }

    private void helper(int row, int n, int[] column, ArrayList<Integer> res){
        if(row == n){
            res.set(0, res.get(0) + 1);
            return;
        }
```

```
            for(int i = 0; i < n; i++){
                column[row] = i;
                if(isValid(row, column)){
                    helper(row + 1, n, column, res);
                }
            }
        }

        private boolean isValid(int row, int[] column){
            for(int i = 0; i < row; i++){
                if(column[i] == column[row] || Math.abs(column[i] - column[row]) == row -
i){
                    return false;
                }
            }
            return true;
        }
```

做题时的感悟:

1. 因为java中所有的传参都是按值传递，所以如果你直接传int res在里面改变res的值是不会影响res在外面的值的，而我们这个res是希望每层递归都能知道最新的值，所以采用传进一个仅一个元素的ArrayList，这样res的值才能真正得到传递。

2. 这道题目中不用在回溯时移除加入的皇后的原因是我们用一个一维数组去存皇后在对应行的哪一列，因为一行只能有一个皇后，如果二维数组，那么就需要把那一行那一列在递归结束后设回没有皇后。

3. 判断是否符合条件的时候有2种情况不符合条件，第一，同列。第二，同左右对角线，可以使用row - i == Math.abs(col[row] - col[i])来判断，涵盖了2种对角线，非常方便。

6. Generate Parentheses，这道题其实是关于卡特兰数的，如果只是要输出结果有多少组，那么直接用卡特兰数的公式就可以。关于卡特兰数，请参见卡特兰数-维基百科，里面有些常见的例子，这个概念还是比较重要的，因为很多问题的原型其实都是卡特兰数，特别是其中这个公式尤为重要:

$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^{n} C_i \, C_{n-i} \quad \text{for } n \geq 0.$$

这个递推式的定义，很多这类问题都可以归结成这个表达式。这个题对于C的定义就是第一对括号中包含有几组括号。因为第一组括号中包含的括号对数量都不同，所以不会重复，接下来就是一个递归定义，里面又可以继续用更小的C去求组合可能性。

这道题一般来说是用递归的方法，因为可以归结为子问题去操作。在每次递归函数中记录左括号和右括号的剩余数量，然后有两种选择，一个是放一个左括号，另一种是放一个右括号。当然有一些否定条件，比如剩余的右括号不能比左括号少，或者左括号右括号数量都要大于0。正常结束条件是左右括号数量都为0。算法的复杂度是O(结果的数量)，因为卡特兰数并不是一个多项式量级的数字，所以算法也不是多项式复杂度的。这道题目主要考查的是递归思想的实现，当然如果可以看透背后是一个卡特兰数的模型，会更好一些。

```
// Solution 1 – Classic Model
public ArrayList<String> generateParenthesis(int n) {
    ArrayList<String> res = new ArrayList<String>();
    if(n <= 0){
        return res;
    }
    StringBuilder item = new StringBuilder();
    helper(n, n, item, res);
    return res;
}

private void helper(int left, int right, StringBuilder item, ArrayList<String> res){
    if(left == 0 && right == 0){
        res.add(item.toString());
        return;
    }
    if(left > 0){
        item.append('(');
        helper(left - 1, right, item, res);
        item.deleteCharAt(item.length() - 1);
    }
    if(right > 0 && left < right){
        item.append(')');
        helper(left, right - 1, item, res);
        item.deleteCharAt(item.length() - 1);
    }
}
```

做题时的感悟:

1. 卡特兰数的公式要熟记，组合数学中有很多的组合结构可以用卡特兰数来计算。leetcode中本题和unique binary search tree都可以用到相关知识。而如果结果是只要求数量，则可以直接用公式求解。

2. 卡特兰数可以解决的常见问题有:

（1）Cn表示长度2n的dyck word的个数。Dyck word是一个有n个X和n个Y组成的字串，且所有的前缀字串皆满足X的个数大于等于Y的个数。

（2）将上例的X换成左括号，Y换成右括号，Cn表示所有包含n组括号合法运算式的个数。

（3）Cn表示有n个节点组成不同构二叉树的方案数。

（4）Cn表示有2n+1个节点组成不同构满二叉树（full binary tree）的方案数。

3. 我们知道String在java中是不可以改变的。所以如果用String来进行下一次迭代时，相当于创建了一个新的对象，原来的String对象并不受影响，所以不用在回溯时删除加入的字符。

4. 一个更优的方法是用StringBuilder来进行迭代。因为StringBuilder对象是可以改变的，而且他们指向同一个对象，所以在回溯之后要清除掉最后加入的字符，即“保护现场”。也正是因为使用同一个对象，所以空间上效率更高，而用String要创建很多个不同的对象。

```
if(left == 0 && right == 0){
    res.add(item.toString());
    return;
}
if(left > 0){
    helper(left - 1, right, item.append("("), res);
    item.deleteCharAt(item.length() - 1);
}
if(right > 0 && left < right){
    helper(left, right - 1, item.append(")"), res);
    item.deleteCharAt(item.length() - 1);
}
```

Code Ganker：这是因为这道题我传进去的是一个item+"("的String，在java中生成一个新的String，然后传入到下一层函数，这里并没有改变item，所以不需要删除最后加入的字符。如果是改变item，然后传进去，那么出来的时候就需要把最后一个括号去掉。一般来说还是还原现场更好，因为那样一直是在一个变量上操作，空间上效率高很多。

5. 递归时间复杂度一般就是用画递归树来解决，如果这块不是很清楚的话，建议看看算法导论里面关于时间复杂度分析的那一章。

7. Word Search，这道题很容易感觉出来是图的题目，其实本质上还是做深度优先搜索。基本思路就是从某一个元素出发，往上下左右深度搜索是否有相等于word的字符串。这里注意每次从一个元素出发时要重置访问标记（也就是说虽然单次搜索字符不能重复使用，但是每次从一个新的元素出发，字符还是重新可以用的）。我们知道一次搜索的复杂度是O(E+V)，E是边的数量，V是顶点数量，在这个问题中他们都是O(m*n)量级的（因为一个顶点有固定上下左右四条边）。加上我们对每个顶点都要做一次搜索，所以总的时间复杂度最坏是O(m^2*n^2)，空间上就是要用一个数组来记录访问情况，所以是O(m*n)。

这道题其实还可以变一变，比如字符可以重复使用。准备的时候多联想还是比较好的，因为面试中常常会做完一道题会变一下问问，虽然不用重新写代码，但是想了解一下思路。

```
// Solution 1 – Classic Model
public boolean exist(char[][] board, String word) {
    if(word == null || word.length() == 0){
        return true;
    }
    if(board == null || board.length == 0 || board[0].length == 0){
        return false;
    }
    boolean[][] used = new boolean[board.length][board[0].length];
    for(int i = 0; i < board.length; i++){
        for(int j = 0; j < board[0].length; j++){
            if(helper(board, 0, i, j, word, used)){
                return true;
            }
        }
    }
    return false;
}

private boolean helper(char[][] board, int idx, int i, int j, String word, boolean[][] used){
    if(idx == word.length()){
        return true;
    }
    if(i < 0 || j < 0 || i >= board.length || j >= board[0].length || used[i][j] || board[i][j] != word.charAt(idx)){
        return false;
    }
```

```
        used[i][j] = true;
        boolean res = helper(board, idx + 1, i - 1, j, word, used) ||
                      helper(board, idx + 1, i + 1, j, word, used) ||
                      helper(board, idx + 1, i, j - 1, word, used) ||
                      helper(board, idx + 1, i, j + 1, word, used);
        used[i][j] = false;
        return res;
    }
```

做题时的感悟:

这里需要一个叫used的boolean[][]矩阵，为了在搜索一条路径的时候，不走到同一个结点，不然可能存在死循环问题。在一条路径搜索完之后，要记得“恢复现场”，因为单次搜索字符不能重复使用，但是每次从一个新的元素出发，字符还是重新可以用的。

8. Word Break II，这道题目要求跟Word Break比较类似，不过返回的结果不仅要知道能不能break，如果可以还要返回所有合法结果。一般来说这种要求会让动态规划的效果减弱很多，因为我们要在过程中记录下所有的合法结果，中间的操作会使得算法的复杂度不再是动态规划的两层循环，因为每次迭代中还需要不是constant的操作，最终复杂度会主要取决于结果的数量，而且还会占用大量的空间。所以这道题目我们直接用递归+剪枝解。

```
    // Solution 1 – Classic Model
    public ArrayList<String> wordBreak(String s, Set<String> dict) {
        ArrayList<String> res = new ArrayList<String>();
        if(s == null || s.length() == 0 || dict == null || dict.size() == 0){
            return res;
        }
        StringBuilder item = new StringBuilder();
        boolean[] possible = new boolean[s.length() + 1];
        Arrays.fill(possible, true);
        helper(0, s, dict, possible, item, res);
        return res;
    }

    private void helper(int start, String s, Set<String> dict, boolean[] possible,
StringBuilder item, ArrayList<String> res){
        if(start == s.length()){
            res.add(item.toString().substring(0, item.length() - 1));
```

```
            return;
        }
        for(int i = start; i < s.length(); i++){
            String piece = s.substring(start, i + 1);
            if(possible[i + 1] && dict.contains(piece)){
                item.append(piece).append(" ");
                int currentResultSize = res.size();
                helper(i + 1, s, dict, possible, item, res);
                if(res.size() == currentResultSize){
                    possible[i + 1] = false;
                }
                item.delete(item.length() - piece.length() - 1, item.length());
            }
        }
    }
```

做题时的感悟:

StringBuilder中的delete(i, j)方法和String中的substring(i, j)方法一样，都是删掉/截取包含i但不包含j的那一段字符串。