DFS回溯

```
result = []
def backtrack(路径, 选择列表):
    if 满足结束条件:
        result.add(路径)
        return

    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

全排列问题

全排列，做个判断即可了。

全排列中判断是否重复添加了，N皇后，判断是否合法即可；

a.Leetcode46 全排列

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]
]

> 结果为0，就是没有用new。
>
> 为了去掉重复的， 看容器里面有没有包含即可了。

```
class Solution {
    // 全部结果
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> permute(int[] nums) {
        // 列出全部的可能性
        if(nums.length==0){
            return res;
        }
        // 回溯
        List<Integer> list = new ArrayList<>();
        // 回溯,存储本次的临时值
        dfs(nums,list);
```

```
        return res;
    }

    // dfs算法
    public void dfs(int[] nums,List<Integer> list){
        // 符合结果
        if(list.size()==nums.length){
            // 需要重新new一下
            res.add(new ArrayList<>(list));
            return;
        }
        //状态
        for(int i=0;i<nums.length;i++){
            // 值只能用一次
            if(list.contains(nums[i])){
                continue;
            }

            // 选择
            list.add(nums[i]);
            // 回溯
            dfs(nums,list);

            list.remove(list.size()-1);
        }

    }
}
```

b.N皇后

```
class Solution {
    // 结果
    List<List<String>> res = new ArrayList<>();
    public List<List<String>> solveNQueens(int n) {
        // 类似全排列的问题
        // 生成一个棋盘
        char[][] chess = new char[n][n];
        // 默认填充为“.”
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                chess[i][j] = '.';
            }
        }

        // 全排列行 结果在chess中
        dfs(chess,0);
        return res;
    }

    // 回溯
    public void dfs(char[][] chess,int row){
        if(row==chess.length){
            // 加入结果
            ArrayList<String> list = new ArrayList<>();
            for(int i=0;i<chess.length;i++){
                list.add(new String(chess[i]));
```

```
                }
                res.add(new ArrayList<>(list));
                return;
            }

            // 继续遍历状态
            for(int col=0;col<chess.length;col++){
                // 删除不合法选择
                if(!valid(chess,row,col)){
                    continue;
                }
                chess[row][col] = 'Q';
                // 回溯
                dfs(chess,row+1);
                chess[row][col] = '.';
            }
        }

        // 检查
        public boolean valid(char[][] chess,int row,int col){
            int n = chess.length;
            // 检查列
            for(int i=0;i<n;i++){
                if(chess[i][col]=='Q'){
                    return false;
                }
            }
            // 检查左上
            for(int i=row-1,j=col-1;i>=0&&j>=0;i--,j--){
                if(chess[i][j]=='Q'){
                    return false;
                }
            }
            // 检查右上
            for(int i=row-1,j=col+1;i>=0&&j<n;i--,j++){
                if(chess[i][j]=='Q'){
                    return false;
                }
            }
            return true;

        }

    }
```

子集

a.Leetcode78 子集

给你一个整数数组 nums ， 数组中的元素 互不相同 。返回该数组所有可能的子集（幂集）。

解集 不能 包含重复的子集。你可以按 任意顺序 返回解集。

示例 1：

输入：nums = [1,2,3]
输出：[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

```
class Solution {
    // 结果
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> subsets(int[] nums) {
        // 临时结果
        List<Integer> list = new ArrayList<>();
        dfs(nums,0,list);
        return res;
    }

    // 回溯
    public void dfs(int[] nums,int index,List<Integer> list){
        // 符合结果
        res.add(new ArrayList<>(list));

        // 其余状态
        for(int i=index;i<nums.length;i++){
            list.add(nums[i]);
            dfs(nums,i+1,list);
            list.remove(list.size()-1);

        }

    }
}
```

组合(类似于子集问题)

a.Leetcode77 组合

给定两个整数 n 和 k，返回 1 ... n 中所有可能的 k 个数的组合。

示例:

输入: n = 4, k = 2
输出:
[
 [2,4],
 [3,4],
 [2,3],
 [1,2],
 [1,3],
 [1,4],
]

```
class Solution {
    // 组合就是子集限制k数量了
    // 结果
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> combine(int n, int k) {
```

```
        // 临时存储结果
        List<Integer> list = new ArrayList<>();
        // 组合的解法类似于子集
        dfs(1,n,list,k);
        return res;
    }

    //  从1...n中选出k个数
    public void dfs(int index,int n,List<Integer> list,int k){
        // 符合条件
        if(list.size()==k){
            res.add(new ArrayList<>(list));
            return;
        }
        // 其它状态
        for(int i=index;i<=n;i++){
            list.add(i);
            dfs(i+1,n,list,k);
            list.remove(list.size()-1);
        }

    }
}
```

应用

a.Leetcode037 解数独

> 不同于N皇后，只需要往上放即可，只需要回溯行即可。
>
> 这个需要回溯行和列，以及for循环数字

```
class Solution {
    public void solveSudoku(char[][] board) {
        // 回溯
        dfs(board,0,0);
    }
    // i表示开始行 j表示开始列
    public boolean dfs(char[][] board,int i,int j){
        int row = 9;int col = 9;
        // 如果列遍历完成
        if(j==col){
            // 继续遍历行 列重新赋值
            return dfs(board,i+1,0);
            // 返回
        }
        // 递归截止条件
        if(i==row){
            return true;
```

```
        }
        // 如果遇到了已经放好的数字
        if(board[i][j]!='.'){
            // 继续下一个
            return dfs(board,i,j+1);
        }

        // 开始拿出状态
        for(char c='1';c<='9';c++){
            //  不合法 继续下一个数字
            if(!isValid(board,i,j,c)){
                continue;
            }
            // 放置
            board[i][j] = c;
            // 一个接即可
            if(dfs(board,i,j+1)){
                return true;
            }
            board[i][j] = '.';

        }
        // 穷举完依然找不到
        return false;
    }
    // 判断是否合法
    public boolean isValid(char[][] board,int r,int c,char n ){
        for(int i=0;i<9;i++){
            // 判断行
            if(board[r][i]==n){
                return false;
            }
            // 判断列
            if(board[i][c]==n){
                return false;
            }
            // 判断3*3的方格
            if(board[(r/3)*3+i/3][(c/3)*3+i%3]==n){
                return false;
            }
        }
        return true;

    }
}
```

b.括号生成

括号问题可以简单分成两类，一类是括号的合法性判断，主要是借助【栈】这种数据，而对于括号的生成，一般都要利用 回溯递归的思想。

数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 有效的 括号组合。

示例 1：

输入：n = 3
输出：["((()))","(()())","(())()","()(())","()()()"]
示例 2：

输入：n = 1
输出：["()"]

```
class Solution {
    public List<String> generateParenthesis(int n) {
        // 回溯生成所有括号，不过先给左右括号
        List<String> res = new ArrayList<>();
        // 每一个结果的路径
        StringBuilder path = new StringBuilder();
        // left从n开始 right从n开始
        dfs(n,n,res,path);
        return res;
    }

    // 回溯
    public void dfs(int left,int right,List<String> res,StringBuilder path){
        // 左括号剩下的多 不合法
        if(right<left){
            return;
        }
        if(left<0 || right<0){
            return;
        }

        if(left==0&&right==0){
            //添加
            res.add(path.toString());
            return;
        }


        // 记录状态
        path.append("(");
        dfs(left-1,right,res,path);
        path.deleteCharAt(path.length()-1);

        path.append(")");
        dfs(left,right-1,res,path);
        path.deleteCharAt(path.length()-1);
    }
}
```

Leetcode140 单词拆分II

给定一个非空字符串 s 和一个包含非空单词列表的字典 wordDict，在字符串中增加空格来构建一个句子，使得句子中所有的单词都在词典中。返回所有这些可能的句子。

说明：

分隔时可以重复使用字典中的单词。
你可以假设字典中没有重复的单词。
示例 1：

输入:
s = "catsanddog"
wordDict = ["cat", "cats", "and", "sand", "dog"]
输出:
[
  "cats and dog",
  "cat sand dog"
]

> 回溯，对其的单词列表进行状态选择

```
class Solution {
    // 单词拆分
    // 结果存储
    List<String> res = new ArrayList<>();
    public List<String> wordBreak(String s, List<String> wordDict) {
        // 每一次的临时结果
        List<String> path = new ArrayList<>();
        // 将其转换为数组
        char[] arr = s.toCharArray();
        //  dfs算法
        dfs(arr,0,wordDict,path);
        return res;
    }
    //  dfs回溯算法
    public void dfs(char[] arr,int index, List<String> wordDict,List<String>
path){
        // 如果遍历到最后了
        if(index==arr.length){
            // 将结果添加进来
            StringBuilder temp = new StringBuilder();
            // 对其遍历
            for(String word:path){
                temp.append(word);
                temp.append(" ");
            }
            res.add(temp.toString().substring(0,temp.length()-1));
        }
        // 递归结束条件
        if(index>arr.length){
            return;
        }
        // 选择状态
        for(String word:wordDict){
            // 当前单词的长度
            int len = word.length();
            // 判断是否相等
            if(index+len<=arr.length&&new String(arr,index,len).equals(word)){
                // 符合条件相等
                path.add(word);
                dfs(arr,index+len,wordDict,path);
                path.remove(path.size()-1);
```

```
            }
        }


    }
}
```

Leetcode093 复原IP地址

给定一个只包含数字的字符串，用以表示一个 IP 地址，返回所有可能从 s 获得的 有效 IP 地址 。你可以按任何顺序返回答案。

有效 IP 地址 正好由四个整数（每个整数位于 0 到 255 之间组成，且不能含有前导 0），整数之间用 '.' 分隔。

例如："0.1.2.201" 和 "192.168.1.1" 是 有效 IP 地址，但是 "0.011.255.245"、"192.168.1.312" 和 "192.168@1.1" 是 无效 IP 地址。

示例 1：

输入：s = "25525511135"
输出：["255.255.11.135","255.255.111.35"]

```
class Solution {
    public List<String> restoreIpAddresses(String s) {
        // 暴力破解每个数的开始和结束
        StringBuilder path = new StringBuilder();
        // 结果
        List<String> res = new ArrayList<>();
        // 对其四个for循环
        for(int a=1;a<4;a++){
            for(int b=1;b<4;b++){
                for(int c=1;c<4;c++){
                    for(int d=1;d<4;d++){
                        if(a+b+c+d==s.length()){
                            int s1 = Integer.parseInt(s.substring(0,a));
                            int s2 = Integer.parseInt(s.substring(a,a+b));
                            int s3 = Integer.parseInt(s.substring(a+b,a+b+c));
                            int s4 = Integer.parseInt(s.substring(a+b+c));
                            //判断是否
                            if(s1<=255&&s2<=255&&s3<=255&&s4<=255){
                                //添加点

 path.append(s1).append(".").append(s2).append(".").append(s3).append(".").appen
d(s4);
                                if(path.length()==s.length()+3){
                                    //添加
                                    res.add(path.toString());
                                }
                                // 清空
                                path.delete(0,path.length());
                            }
                        }
                    }
                }
            }
```

```
        }
        return res;

    }
}
```

> 回溯算法

```
class Solution {
    List<String> res = new ArrayList<>();
    public List<String> restoreIpAddresses(String s) {
        // 临时存储结果
        int[] arr = new int[4];
        // 一个是s的开端 一个是arr的开端
        dfs(s,arr,0,0);
        return res;
    }
    // 回溯
    public void dfs(String s,int[] arr,int s_start,int arr_start){
        // 如果 都遍历到最后了
        if(arr_start==arr.length){
            if(s_start==s.length()){
                // 开始转换
                StringBuilder temp = new StringBuilder();
                //
                for(int i=0;i<arr.length;i++){
                    temp.append(arr[i]);
                    if(i!=arr.length-1){
                        temp.append(".");
                    }
                }
                res.add(temp.toString());
            }
            return;
        }
        // 其他截至条件
        if(s_start==s.length()){
            return;
        }

        // 如果是0的话
        if(s.charAt(s_start)=='0'){
            arr[arr_start] = 0;
            dfs(s,arr,s_start+1,arr_start+1);
        }
        // 遍历
        // 数字
        int digit = 0;
        for(int i=s_start;i<s.length();i++){
            digit = digit*10 + (s.charAt(i)-'0');
            // 符合条件
            if(digit>0&&digit<=0xFF){
                arr[arr_start] = digit;
                dfs(s,arr,i+1,arr_start+1);
            }else{
                break;
            }
```

```
        }

    }
}
```