12 回溯详解

起手式：
回溯法，就是试探法，按照优选条件去向前搜索，以达到目标。

但是在搜索到某一步时，发现原先这样并不能满足条件，就回退一步重新选择，这种走不通就退回再走的技术成为回溯法。

在做回溯法的题目的时候，有添加状态或元素就一定有与之对应的回退状态和元素。若是寻找成功，回退以查看有没有其他满足条件的解；如果寻找不成功，回退以查看其它情况。

例题： 从1，… ，n中取出k个数，要求不重复。

n=4, k=2, 1,2 1,3 1,4 2,3 2,4 3,4

题目框架：

```
public class Solution {
    public List<List<Integer>> combine(int n, int k) {

    }
}
```

我们很容易想到，先建一个全部变量

`public List<List<Integer>> result = new ArrayList<>();` 然后我们可以新建一个回溯函数的形式，以迭代调用：

`backtracking(int n, int k, int start,List<Integer> list)` 其中 n代表总数n， k代表还要取出多少个数，start代表起始取数位置， list用来保存已知的解。

这些肯定都是要用到的吧，然后来看看具体的回溯函数：

```
    public  List<List<Integer>> result = new ArrayList<>();
    public  List<List<Integer>> combine(int n, int k) {
        List<Integer> list = new ArrayList<>();
        backtracking(n, k, 1, list);
        return result;
    }

    public  void backtracking(int n, int k, int start,List<Integer> list){
        if(k<0){
            return;
```

```
        }else if(k==0){
            result.add(new ArrayList<>(list));
        }else{
            for (int i = start; i <= n; i++) {
                list.add(i);                        // 添加元素
                backtracking(n,k-1,i+1,list);
                list.remove(list.size()-1);         // 回退
            }
        }
    }
```

具体思路：

1.对于n=4，k=2，1,2,3,4中选2个数字，我们可以做如下尝试，加入先选择1，那我们只需要再选择一个数字，注意这时候k=1了（此时只需要选择1个数字啦）。当然，我们也可以先选择2,3 或者4，通俗化一点，我们可以选择（1-n）的所有数字，这个是可以用一个循环来描述？每次选择一个加入我们的链表list中，下一次只要再选择k-1个数字。那什么时候结束呢？当然是k<0的时候啦，这时候都选完了。

2.加入了一个start变量，它是i的起点。为什么要加入它呢？比如我们第一次加入了1，下一次搜索的时候还能再搜索1了么？肯定不可以啊！我们必须从他的下一个数字开始，也就是2 、3或者4啦。所以start就是一个开始标记这个很重要。

3.加上：list.remove(list.size()-1);他的作用就是每次清除一个空位 让后续元素加入。寻找成功，最后一个元素要退位，寻找不到，方法不可行，那么我们回退，也要移除最后一个元素。

起手式变形：
给定一个数组{1，2，3}，得出所有的组合方式[],1,2,3,12,13,23,123。
同样可以用回溯法解决，求出组合在暴力求解的时候很有用，用于枚举出所有可能结果，再一一判断。

```
    public ArrayList<ArrayList<Integer>> combine(int[] nums){
        ArrayList<ArrayList<Integer>> result = new ArrayList<>();
        ArrayList<Integer> list = new ArrayList<>();
        backtracing(result,list,0,nums);
        return result;
    }

    public void backtracing(ArrayList<ArrayList<Integer>> result,ArrayList<
Integer> list,int start, int[] nums){
        result.add(new ArrayList<>(list));
        for (int i = start; i < nums.length; i++) {
```

```
            list.add(nums[i]);
            backtracing(result,list,i+1,nums);
            list.remove(list.size()-1);
        }
    }
```

进阶式：
分割字符串：

给一个字符串,你可以选择在一个字符或两个相邻字符之后拆分字符串,使字符串由仅一个字符或两个字符组成,输出所有可能的结果

样例：
给一个字符串"123"
返回[["1","2","3"],["12","3"],["1","23"]]

分析：我们可以设置一个起始位置start，每次我们都从start位置开始向后划分一个或者两个字符出去，这样就有很多种划分路径，当一种路径走到末端的时候，之前划分出去的需要回退以再次以不同形式划分。

```
    public static  List<List<String>> result = new ArrayList<>();    // 首先新建一个 result
    public static List<List<String>> splitString(String s) {
        List<String> list = new ArrayList<>();
        backtraceing(s, 0, list);                                    // 回溯函数
        return result;
    }
    public static void backtraceing(String s, int start, List<String> list){ // start标注每次的起始位置
        if(start>s.length()){
            return;
        }else if(start==s.length()){
            result.add(new ArrayList<>(list));
        }else{
            for (int i = start; i<start+2 && i < s.length(); i++) {   // i<start+2:  每次划分一个或者两个字符
                String subString = s.substring(start,i+1);
                list.add(subString);
                backtraceing(s,i+1,list);
                list.remove(list.size()-1);
            }
```

```
        }
    }
```

高阶式：
例：矩阵中的路径

题目描述
请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子。 例如 a b c e s f c s a d e e 矩阵中包含一条字符串"bcced"的路径，但是矩阵中不包含"abcb"路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。

```
public class Solution {
    public boolean hasPath(char[] matrix, int rows, int cols, char[] str)
    {

    }
}
```

思路：1. 在矩阵中任选一个格子作为路径的起点。如果路径上的第i个字符不是ch，那么这个格子不可能处在路径上的 第i个位置。如果路径上的第i个字符正好是ch，那么往相邻的格子寻找路径上的第i+1个字符。除在矩阵边界上的格子之外，其他格子都有4个相邻的格子。重复这个过程直到路径上的所有字符都在矩阵中找到相应的位置。

2. 找下一个节点时，可能是刚刚搜索过的位置，所以需要新建一个visited数组记录下一个节点是否被访问过。

参考代码：

```
    public boolean hasPath(char[] matrix, int rows, int cols, char[] str)
    {
        if(matrix==null || matrix.length==0 || str==null || str.length==0 |
| rows<=0 || cols<=0 ||
                rows*cols < str.length){
            return false;
        }
        boolean[] visited = new boolean[rows*cols];
        char[] path = null;
        int start = 0;                          //表示str的起始位置
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if(isbacktracing(matrix, i, j, rows, cols, str, visited, st
```

```
art)){   // 从第i，j个位置是否能走通
                    return true;
                }
            }
        }
        return false;

    }
    public boolean isbacktracing(char[] matrix,int i, int j,int rows,int co
ls,char[] str,boolean[] visited,int start){
        if(start >= str.length){
            return true;
        }else {
            int index = i*cols + j;     //  记录当前访问的节点
            if(i<0 || i>=rows || j<0 || j>=cols || matrix[index] != str[sta
rt] || visited[index]==true){ //判断是否合适或者是否已被访问
                return false;
            }
            visited[index] = true;      //  标记已读
            boolean judge = isbacktracing(matrix,i+1,j,rows,cols,str,visite
d,start+1)||

isbacktracing(matrix,i,j+1,rows,cols,str,visited,start+1)||
                    isbacktracing(matrix,i-
1,j,rows,cols,str,visited,start+1)||
                    isbacktracing(matrix,i,j-
1,rows,cols,str,visited,start+1);
            if(judge){
                return true;
            }
            visited[index] = false;     // 回退
            return false;               // 搜索不到，返回false
        }
    }
```