面试题 08.12. 八皇后

```
var result  [][]string
func solveNQueens(n int) [][]string {
    result = make([][]string, 0)
    board := make([][]byte, n)
    for i := 0; i < n; i++ {
        board[i] = make([]byte, n)
        for j := 0; j < n; j++ {
            board[i][j] = '.'
        }
    }
    backtrack(0, board, n)
    return result
}

//row: 阶段
//board: 路径，记录已经做出的决策
//可选列表： 通过 board 推导出来， 没有显示记录
func backtrack(row int, board [][]byte, n int) {
    //结束条件,得到可行解
    if row == n {
        snapshot := make([]string, len(board))
        for i := 0; i < n; i++ {
            snapshot[i] = string(board[i])
        }
        result = append(result, snapshot)
        return
    }
    for col := 0; col < n; col++ { //每一行都有 n 种放法
        if isOk(board, n, row, col) { //可选列表
            board[row][col] = 'Q' //做选择，第 row 行的棋子放在 col 列
            backtrack(row+1, board, n) //考察下一行
            board[row][col] = '.' //恢复选择
        }
    }
}

//判断 row 行 column 列放置是否合适
func isOk(board [][]byte, n, row, col int) bool{
    //检查是否有冲突
    for i := 0; i < n; i++ {
        if board[i][col] == 'Q' {
            return false
        }
    }
    //检查右上对角线是否有冲突
    i := row-1
    j := col+1
    for i >= 0 && j < n {
        if board[i][j] == 'Q' {
            return false
        }
        i--
        j++
    }
```

```
    //检查左上对角线是否有冲突
    i = row-1
    j = col-1
    for i >= 0 && j >= 0 {
        if board[i][j] == 'Q' {
            return false
        }
        i--
        j--
    }
    return true
}
```

37. 解数独

```
var rows [][]bool
var cols [][]bool
var blocks [][][]bool
var solved bool
func solveSudoku(board [][]byte) {
    initParam()
    for i := 0; i < 9; i++ {
        for j := 0; j < 9; j++ {
            if board[i][j] != '.' {
                num := board[i][j] - '0'
                rows[i][num] = true
                cols[j][num] = true
                blocks[i/3][j/3][num] = true
            }
        }
    }
    backtrack(0, 0, board)
}

func initParam() {
    rows = make([][]bool, 9)
    for i := 0; i < len(rows);i ++ {
        rows[i] = make([]bool, 10)
    }
    cols = make([][]bool, 9)
    for i := 0; i < len(cols);i ++ {
        cols[i] = make([]bool, 10)
    }
    blocks = make([][][]bool, 3)
    for i := 0; i < len(blocks); i++ {
        blocks[i] = make([][]bool, 3)
        for j := 0; j < len(blocks); j++{
            blocks[i][j] = make([]bool, 10)
        }
    }
    solved = false
}

func backtrack(row, col int, board [][]byte) {
    if row == 9 {
        solved = true
        return
    }
    if board[row][col] != '.' {
        nextRow := row
```

```
            nextCol := col+1
            if col == 8 {
                nextRow = row+1
                nextCol = 0
            }
            backtrack(nextRow, nextCol, board)
            if solved {return}
        } else {
            for num := 1; num <= 9; num++ {
                if !rows[row][num] && !cols[col][num] && !blocks[row/3][col/3][num] {
                    board[row][col] = byte(num+'0')
                    rows[row][num] = true
                    cols[col][num] = true
                    blocks[row/3][col/3][num] = true
                    nextRow := row
                    nextCol := col+1
                    if col == 8 {
                        nextRow = row+1
                        nextCol = 0
                    }
                    backtrack(nextRow, nextCol, board)
                    if solved {return}
                    board[row][col] = '.'
                    rows[row][num] = false
                    cols[col][num] = false
                    blocks[row/3][col/3][num] = false
                }
            }
        }
    }
```

17. 电话号码的字母组合

```
var result []string
func letterCombinations(digits string) []string {
    result = make([]string, 0)
    if len(digits) == 0 {return []string{}}
    mappings := make([]string, 10)
    mappings[2] = "abc"
    mappings[3] = "def"
    mappings[4] = "ghi"
    mappings[5] = "jkl"
    mappings[6] = "mno"
    mappings[7] = "pqrs"
    mappings[8] = "tuv"
    mappings[9] = "wxyz"
    path := make([]byte, len(digits))
    backtrack(mappings, digits, 0, path)
    return result
}

func backtrack(mappings []string, digits string, k int, path []byte) {
    if k == len(digits) {
        result = append(result, string(path))
        return
    }
    mapping := mappings[digits[k]-'0']
    for i := 0; i < len(mapping); i++ {
        path[k] = mapping[i]
        backtrack(mappings, digits, k+1, path)
```

```
    }
}
```

77. 组合

```
var result [][]int
func combine(n int, k int) [][]int {
    result = make([][]int, 0)
    backtrack(n, k, 1, []int{})
    return result
}

func backtrack(n, k, step int, path []int) {
    if len(path) == k {
        snapshot := make([]int, len(path))
        copy(snapshot, path)
        result = append(result, snapshot)
        return
    }
    if step == n+1 {
        return
    }
    backtrack(n, k, step+1, path)
    path = append(path, step)
    backtrack(n, k, step+1, path)
    path = path[:len(path)-1]
}
```

78. 子集

```
var result [][]int
func subsets(nums []int) [][]int {
    result = make([][]int, 0)
    backtrack(nums, 0, []int{})
    return result
}

func backtrack(nums []int, k int, path []int) {
    if k == len(nums) {
        snapshot := make([]int, len(path))
        copy(snapshot, path)
        result = append(result, snapshot)
        return
    }
    backtrack(nums, k+1, path)
    path = append(path, nums[k])
    backtrack(nums, k+1, path)
    path = path[:len(path)-1]
}
```

90. 子集 II

```
var result [][]int
func subsetsWithDup(nums []int) [][]int {
    result = make([][]int, 0)
    hm := make(map[int]int, 0)
    for i := 0; i < len(nums); i++ {
        count := 1
        if _, ok := hm[nums[i]]; ok {
```

```
            count += hm[nums[i]]
        }
        hm[nums[i]] = count
    }
    n := len(hm)
    uniqueNums := make([]int, n)
    counts := make([]int, n)
    k := 0
    for i := 0; i < len(nums); i++ {
        if _, ok := hm[nums[i]]; ok {
            uniqueNums[k] = nums[i]
            counts[k] = hm[nums[i]]
            k++
            delete(hm, nums[i])
        }
    }
    backtrack(uniqueNums, counts, 0, []int{})
    return result
}

func backtrack(uniqueNums, counts []int, k int, path []int) {
    if k == len(uniqueNums){
        snapshot := make([]int, len(path))
        copy(snapshot, path)
        result = append(result, snapshot)
        return
    }
    for count := 0; count <= counts[k]; count++ {
        for i := 0; i < count; i++ {
            path = append(path, uniqueNums[k])
        }
        backtrack(uniqueNums, counts, k+1, path)
        for i := 0; i < count; i++ {
            path = path[:len(path)-1]
        }
    }
}
```

46. 全排列

```
var result [][]int
func permute(nums []int) [][]int {
    result = make([][]int, 0)
    path := make([]int, 0)
    backtrack(nums, 0, path)
    return result
}

func backtrack(nums []int, k int, path []int) {
   if k == len(nums) {
        snapshot := make([]int, len(path))
        copy(snapshot, path)
        result = append(result, snapshot)
        return
   }
   for i := 0; i < len(nums); i++ {
        if contain(path, nums[i]) {
            continue
        }
        path = append(path, nums[i])
        backtrack(nums, k+1, path)
```

```
        path = path[:len(path)-1]
    }
}

func contain(path []int, num int) bool{
    for _, p := range path {
        if p == num {
            return true
        }
    }
    return false
}
```

47. 全排列 II

```
var result [][]int
func permuteUnique(nums []int) [][]int {
    result = make([][]int, 0)
    hm := make(map[int]int, 0)
    for i := 0; i < len(nums); i++ {
        count := 1
        if _, ok := hm[nums[i]]; ok {
            count += hm[nums[i]]
        }
        hm[nums[i]] = count
    }
    n := len(hm)
    uniqueNums := make([]int, n)
    counts := make([]int, n)
    k := 0
    for i := 0; i < len(nums); i++ {
        if _, ok := hm[nums[i]]; ok {
            uniqueNums[k] = nums[i]
            counts[k] = hm[nums[i]]
            k++
            delete(hm, nums[i])
        }
    }
    backtrack(uniqueNums, counts, 0, []int{}, len(nums))
    return result
}

func backtrack(uniqueNums, counts []int, k int, path []int, n int) {
    if k == n {
        snapshot := make([]int, len(path))
        copy(snapshot, path)
        result = append(result, snapshot)
        return
    }
    for i := 0; i < len(uniqueNums); i++ {
        if counts[i] == 0 {continue}
        path = append(path, uniqueNums[i])
        counts[i]--
        backtrack(uniqueNums, counts, k+1, path, n)
        path = path[:len(path)-1]
        counts[i]++
    }
}
```

39. 组合总和

```go
var result [][]int
func combinationSum(candidates []int, target int) [][]int {
    result = make([][]int, 0)
    backtrack(candidates, 0, target, []int{})
    return result
}

func backtrack(candidates []int, k, left int, path []int) {
    if left == 0 {
        snapshot := make([]int, len(path))
        copy(snapshot, path)
        result = append(result, snapshot)
        return
    }
    if k == len(candidates) {
        return
    }
    for i := 0; i <= left/candidates[k]; i++ {
        for j := 0; j < i; j++ {
            path = append(path, candidates[k])
        }
        backtrack(candidates, k+1, left-i*candidates[k], path)
        for j := 0; j < i; j++ {
            path = path[:len(path)-1]
        }
    }
}
```

40. 组合总和 II

```go
var result [][]int
func combinationSum2(candidates []int, target int) [][]int {
    result = make([][]int, 0)
    hashTable := make(map[int]int, 0)
    for i := 0; i < len(candidates); i++ {
        if _, ok := hashTable[candidates[i]]; !ok {
            hashTable[candidates[i]] = 1
        } else {
            hashTable[candidates[i]] =  hashTable[candidates[i]]+1
        }
    }
    nums := make([]int, 0)
    counts := make([]int, 0)
    for i := 0; i < len(candidates); i++ {
        if _, ok := hashTable[candidates[i]]; ok {
            nums = append(nums, candidates[i])
            counts = append(counts, hashTable[candidates[i]])
            delete(hashTable, candidates[i])
        }
    }
    backtrack(nums, counts, 0, target, []int{})
    return result
}

func backtrack(nums, counts []int, k, left int, path []int) {
    if left == 0 {
        snapshot := make([]int, len(path))
        copy(snapshot, path)
        result = append(result, snapshot)
        return
```

```
    }
    if left < 0 || k == len(nums) {
        return
    }
    for count := 0; count <= counts[k]; count++ {
        for i := 0; i < count; i++ {
            path = append(path, nums[k])
        }
        backtrack(nums, counts, k+1, left-count*nums[k], path)
        for i := 0; i < count; i++ {
            path = path[:len(path)-1]
        }
    }
}
```

216. 组合总和 III

```
var result [][]int
func combinationSum3(k int, n int) [][]int {
    result = make([][]int, 0)
    backtrack(k, n, 1, 0, []int{})
    return result
}

func backtrack(k, n, step, sum int, path []int) {
    if sum == n && len(path) == k {
        snapshot := make([]int, len(path))
        copy(snapshot, path)
        result = append(result, snapshot)
        return
    }
    if sum > n || len(path) > k || step > 9 {
        return
    }
    backtrack(k, n, step+1, sum, path)
    path = append(path, step)
    backtrack(k, n, step+1, sum+step, path)
    path = path[:len(path)-1]
}
```

131. 分割回文串

```
var result [][]string
func partition(s string) [][]string {
    result = make([][]string, 0)
    backtrack(s, 0, []string{})
    return result
}

func backtrack(s string, k int, path []string) {
    if k == len(s) {
        snapshot := make([]string, len(path))
        copy(snapshot, path)
        result = append(result, snapshot)
        return
    }
    for end := k; end < len(s); end++ {
        if ispalidrome(s, k, end) {
            path = append(path, s[k:end+1])
            backtrack(s, end+1, path)
```

```
            path = path[:len(path)-1]
        }
    }
}

func ispalidrome(s string, p int, r int) bool{
    i := p
    j := r
    for i <= j {
        if s[i] != s[j] {return false}
        i++
        j--
    }
    return true
}
```

93. 复原 IP 地址

```
var result []string
func restoreIpAddresses(s string) []string {
    result = make([]string, 0)
    backtrack(s, 0, 0, []int{})
    return result
}

func backtrack(s string, k ,step int, path []int) {
    if k == len(s) && step == 4 {
        sb := strings.Builder{}
        for i := 0; i < 3; i++ {
            sb.Write([]byte(fmt.Sprintf("%d", path[i])))
            sb.WriteByte(byte('.'))
        }
        sb.Write([]byte(fmt.Sprintf("%d", path[3])))
        result = append(result, sb.String())
    }
    if step > 4 {
        return
    }
    if k == len(s) {
        return
    }
    val := 0
    //1 位数
    if k < len(s) {
        val = val * 10 + int(s[k]-'0')
        path = append(path, val)
        backtrack(s, k+1, step+1, path)
        path = path[:len(path)-1]
    }
    if s[k] == '0' {
        return
    }
    //2 位数
    if k+1 < len(s) {
        val = val * 10 + int(s[k+1]-'0')
        path = append(path, val)
        backtrack(s, k+2, step+1, path)
        path = path[:len(path)-1]
    }
```

```
    //3 位数
    if k+2 < len(s) {
        val = val * 10 + int(s[k+2]-'0')
        if val <= 255 {
            path = append(path, val)
            backtrack(s, k+3, step+1, path)
            path = path[:len(path)-1]
        }
    }
}
```

22. 括号生成

```
var result []string
func generateParenthesis(n int) []string {
    result = make([]string, 0)
    path := make([]byte, 2*n)
    backtrack(n, 0, 0, 0, path)
    return result
}

func backtrack(n, leftUsed, rightUsed, k int, path []byte) {
    if k == 2*n {
        result = append(result, string(path))
        return
    }
    if leftUsed < n {
        path[k] = '('
        backtrack(n, leftUsed+1, rightUsed, k+1, path)
    }
    if leftUsed > rightUsed {
        path[k] = ')'
        backtrack(n, leftUsed, rightUsed+1, k+1, path)
    }
}
```