八皇后、回溯与位运算

补充知识点及其应用 (2/2)

八皇后与回溯算法

DFS的拓展

八皇后问题

八皇后问题，是一个古老而著名的问题，是回溯算法的典型案例。该问题是国际西洋棋棋手马克斯·贝瑟尔于1848年提出：在8×8格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。 高斯认为有76种方案。1854年在柏林的象棋杂志上不同的作者发表了40种不同的解，后来有人用图论的方法解出92种结果。

回溯法

回溯法（backtracking）是暴力搜索法中的一种

对于某些计算问题而言，回溯法是一种可以找出所有（或一部分）解的一般性算法，尤其适用于约束补偿问题（在解决约束满足问题时，我们逐步构造更多的候选解，并且在确定某一部分候选解不可能补全成正确解之后放弃继续搜索这个部分候选解本身及其可以拓展出的子候选解，转而测试其他的部分候选解）

回溯法采用**试错**的思想，它尝试分步的去解决一个问题。在分步解决问题的过程中，当它通过尝试发现，现有的分步答案不能得到有效的正确的解答的时候，它将取消上一步甚至是上几步的计算，再通过其它的可能的分步解答再次尝试寻找问题的答案。回溯法通常用最简单的递归方法来实现，在反复重复上述的步骤后可能出现两种情况：

- 找到一个可能存在的正确的答案
- 在尝试了所有可能的分步方法后宣告该问题没有答案

在最坏的情况下，回溯法会导致一次复杂度为指数时间的计算

回溯法: 框架

回溯法的实现可以参考以下的框架:

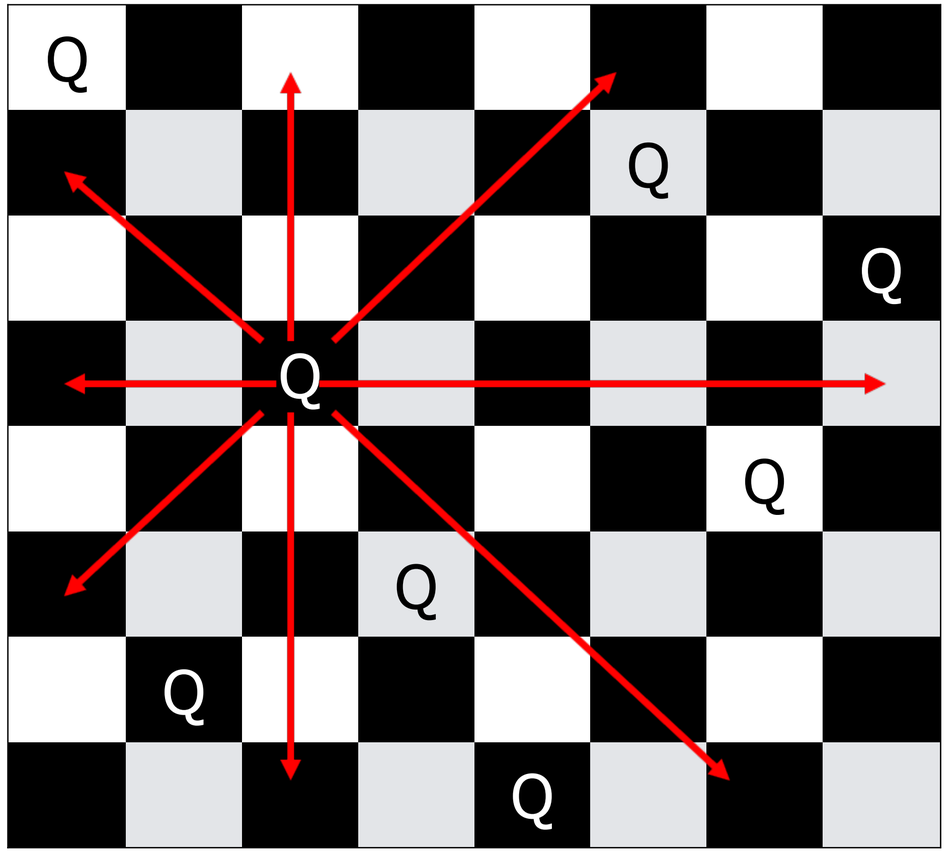
```
Explore(choices):
  if there are no more choices to make:
    stop.
  else:
    for each available choice C:
      Choose C.
      Explore the remaining choices.
      Un-choose C, if necessary.  // backtrack!
```

八皇后问题: 回溯法求解

问题: 在8×8格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法

思考以下三个问题:

1. 如何定义 “choices”?
2. 如何 “Choose” & “Un-choose”?
3. 如何确定是否需要停止?



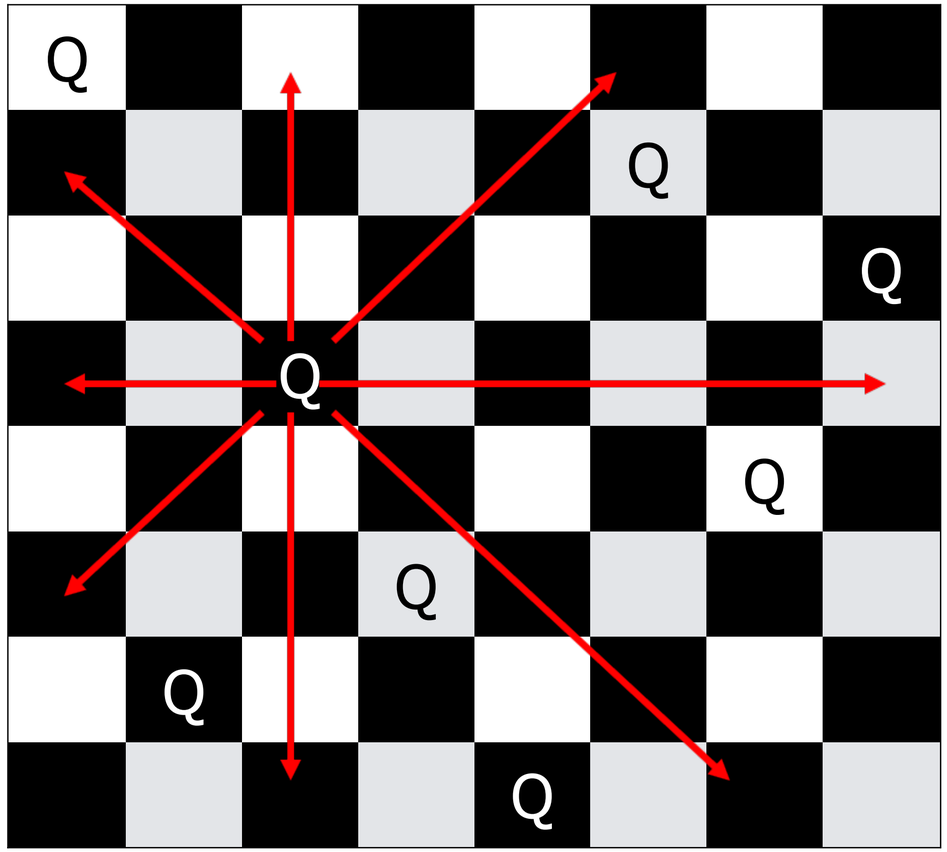

```
Explore(choices):
  if there are no more choices to make:
    stop.
  else:
    for each available choice C:
      Choose C.
      Explore the remaining choices.
      Un-choose C, if necessary.  // backtrack!
```

八皇后问题: 回溯法求解

问题: 在8×8格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法

回溯策略:

1. 在第 1 行第一个可以放置的格子放置 1 个皇后
2. 考察下一行，选择第一个可以放置的格子放置皇后 (不与已放置的皇后冲突)
3. 重复以上过程直到
    1. 问题解决
    2. 无法继续
        - 当无法继续时，移除上一行的皇后（该摆法导致当前行不存在可以选择的格子），直到当前行出现能选择的格子



八皇后问题: 回溯法求解

问题: 在8×8格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法

回溯策略:

1. 在第 1 行第一个可以放置的格子放置 1 个皇后
2. 考察下一行，选择第一个可以放置的格子放置皇后 (不与已放置的皇后冲突)
3. 重复以上过程直到
    1. 问题解决
    2. 无法继续
        - 当无法继续时，移除上一行的皇后（该摆法导致当前行不存在可以选择的格子），直到当前行出现能选择的格子

```
dfs(x):
    if x == 8: //如果x等于8，说明每行的皇后都放置完毕
        //将棋盘内容保存下来
        return
    for y from 0 to 8:
        if [x,y]这个位置是有效的:
            将棋盘[x,y]位置设置为Q
            dfs(x+1) 继续尝试下一行
            将棋盘[x,y]位置还原
```

N 皇后问题: 回溯法求解

问题: 在N×N格的国际象棋上摆放N个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法

LeetCode 51. N皇后 困难
LeetCode 52. N皇后 II 困难

回溯策略:

1. 在第 1 行第一个可以放置的格子放置 1 个皇后
2. 考察下一行，选择第一个可以放置的格子放置皇后 (不与已放置的皇后冲突)
3. 重复以上过程直到
    1. 问题解决
    2. 无法继续
        - 当无法继续时，移除上一行的皇后（该摆法导致当前行不存在可以选择的格子），直到当前行出现能选择的格子

```
dfs(x):
    if x == N: //如果x等于N，说明每行的皇后都放置完毕
        //将棋盘内容保存下来
        return
    for y from 0 to N:
        if [x,y]这个位置是有效的:
            将棋盘[x,y]位置设置为Q
            dfs(x+1) 继续尝试下一行
            将棋盘[x,y]位置还原
```

N 皇后问题: 回溯法求解

```
boolean check(char[][] board, int x, int y, int n) {
        // 检查竖着的一列是否有皇后
        for (int i = 0; i < x; i++) {
            if (board[i][y] == 'Q') {
                return false;
            }
        }
        // 检查左上到右下的斜边否有皇后
        int i = x - 1, j = y - 1;
        while (i >= 0 && j >= 0) {
            if (board[i][j] == 'Q') {
                return false;
            }
            i--;
            j--;
        }
        // 检查左下到右上的斜边否有皇后
        i = x - 1;
        j = y + 1;
        while (i >= 0 && j < n) {
            if (board[i][j] == 'Q') {
                return false;
            }
            i--;
            j++;
        }
        return true;
    }
}
```

```
List<List<String>> NQueens(int n) {
    // 生成 N*N 的棋盘
    char[][] board = new char[n][n];
    // 填充棋盘，每个格子默认“.” 表示没有放置皇后
    for (int i = 0; i < n; i++) {
        Arrays.fill(board[i], '.');
    }
    List<List<String>> result = new ArrayList<List<String>>();
    dfs(board, 0, n, result);
    return result;
}

void dfs(char[][] board, int x, int n, List<List<String>> result) {
    // x从0开始计算，当 x==n 时所有行的皇后都放置完毕，此时记录结果
    if (x == n) {
        List<String> ans = new ArrayList<String>();
        for(int i = 0; i < n; i++) {
            StringBuilder row = new StringBuilder();
            for (int j = 0; j < n; j++) {
                if(board[i][j] == '.') {
                    row.append(".");
                } else {
                    row.append("Q");
                }
            }
            ans.add(row.toString());
        }
        result.add(ans);
        return;
    }
    // 遍历每一列
    for (int y = 0; y < n; y++) {
        // 检查[x,y]这一坐标否可以放置皇后
        // 如果满足条件，就放置皇后，并继续检查下一行
        if(check(board, x, y, n)) {
            board[x][y] = 'Q';
            dfs(board, x + 1, n, result);
            board[x][y] = '.'; // 回溯!
        }
    }
}
```

N 皇后问题: 回溯法求解(优化)

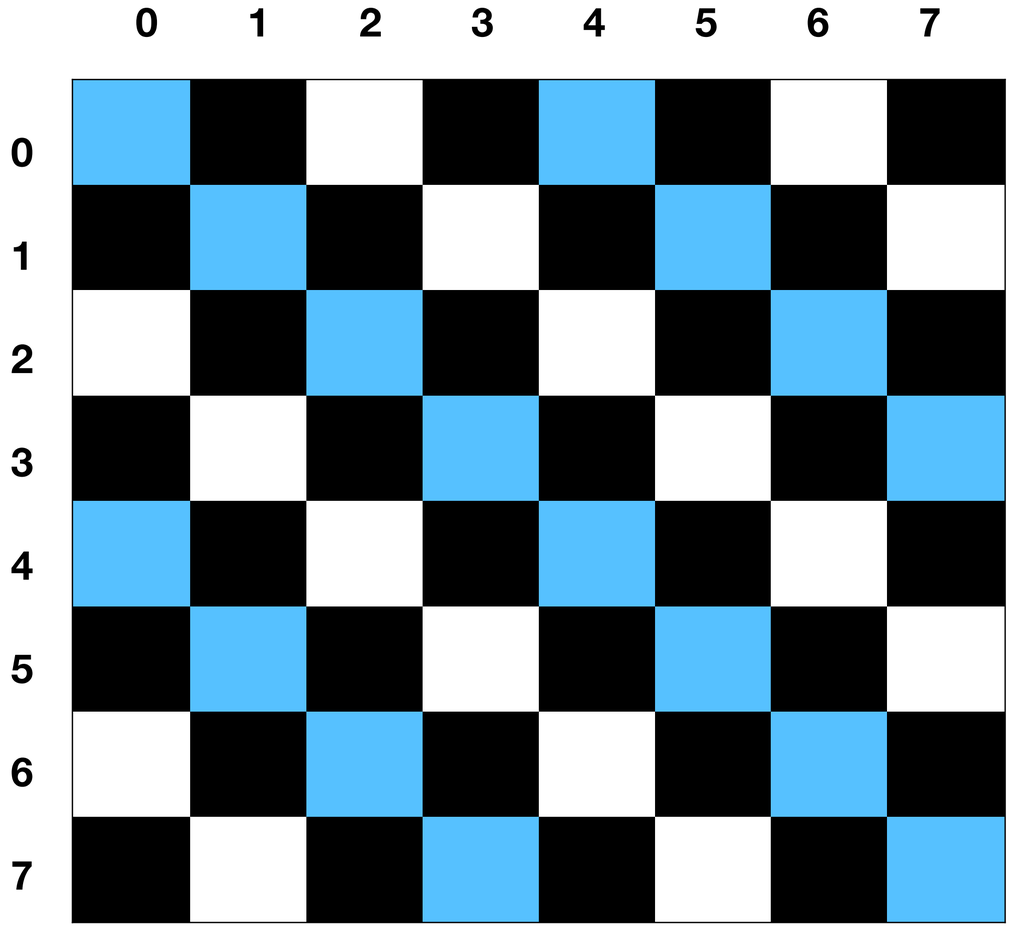
```
boolean check(char[][] board, int x, int y, int n) {
        // 检查竖着的一列是否有皇后
        for (int i = 0; i < x; i++) {
            if (board[i][y] == 'Q') {
                return false;
            }
        }
        // 检查左上到右下的斜边否有皇后
        int i = x - 1, j = y - 1;
        while (i >= 0 && j >= 0) {
            if (board[i][j] == 'Q') {
                return false;
            }
            i--;
            j--;
        }
        // 检查左下到右上的斜边否有皇后
        i = x - 1;
        j = y + 1;
        while (i >= 0 && j < n) {
            if (board[i][j] == 'Q') {
                return false;
            }
            i--;
            j++;
        }
        return true;
    }
}
```

使用三个集合 columns、diagonals1 和 diagonals2 分别记录每一列以及两个方向的每条斜线上是否有皇后

N 皇后问题: 回溯法求解(优化)

使用三个集合 columns、diagonals1 和 diagonals2
分别记录每一列以及两个方向的每条斜线上是否有皇后

diagonals1 方向



斜线上 行下标 - 列下标 = 4

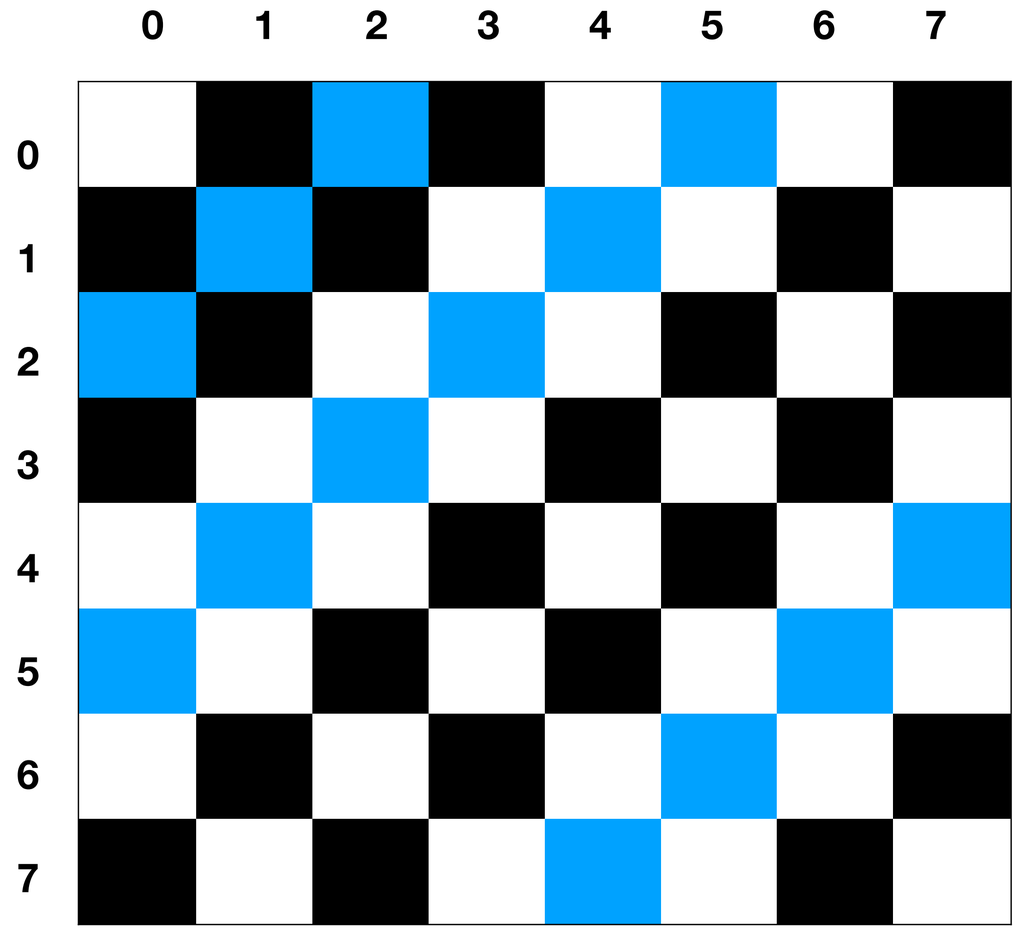
斜线上 行下标 - 列下标 = -4

斜线上 行下标 - 列下标 = 0

N 皇后问题: 回溯法求解(优化)

使用三个集合 columns、diagonals1 和 diagonals2
分别记录每一列以及两个方向的每条斜线上是否有皇后

diagonals2 方向



斜线上 行下标 + 列下标 = 2

斜线上 行下标 + 列下标 = 5

斜线上 行下标 + 列下标 = 11

N 皇后问题: 回溯法求解(优化)

```
List<List<String>> NQueens(int n) {
    // 创建一个 N 行的数组，下标 i 对应 N*N 棋盘格子第 i 行的皇后位置
    int[] board = new int[n];
    List<List<String>> result = new ArrayList<List<String>>();
    // 三个集合，分别判断某一列，左斜线(左上到右下的斜线)，右斜线(左下到右上的斜线)
    Set<Integer> columns = new HashSet<Integer>();
    Set<Integer> diagonals1 = new HashSet<Integer>();
    Set<Integer> diagonals2 = new HashSet<Integer>();
    dfs(result, n, 0, board, columns, diagonals1, diagonals2);
    return result;
}
```

时间复杂度：O(N!)，其中 N 是皇后数量。

空间复杂度：O(N)，其中 N 是皇后数量。空间复杂度主要取决于递归调用层数、记录每行放置的皇后的列下标的数组以及三个集合，递归调用层数不会超过 N，数组的长度为 N，每个集合的元素个数都不会超过 N。

```
void dfs(List<List<String>> result, int n, int x, int[] board,
    Set<Integer> columns, Set<Integer> diagonals1, Set<Integer> diagonals2) {
        if (x == n) {
            // 如果 x == n 说明所有的皇后都摆放完了
            // 将 board 数组中保存的结果拼接起来
            List<String> ans = new ArrayList<String>();
            for (int k = 0; k < n; k++) {
                char[] row = new char[n];
                Arrays.fill(row, '.');
                row[board[k]] = 'Q';
                ans.add(new String(row));
            }
            result.add(ans);
            return;
        }
        // 遍历一行中的每一列，并检查竖线、左斜线、右斜线是否有皇后
        for (int y = 0; y < n; y++) {
            if (columns.contains(y)) {
                continue;
            }
            if (diagonals1.contains(x - y)) {
                continue;
            }
            if (diagonals2.contains(x + y)) {
                continue;
            }
            // 如果检查通过，设置这一行的皇后位置，并将竖线、左斜线、右斜线的值放入集合中，继续下一行递归
            // 当下一层的所有递归遍历完后，回到本轮需要将之前集合、board数组中保存的结果都清空
            board[x] = y;
            columns.add(y);
            diagonals1.add(x - y);
            diagonals2.add(x + y);
            dfs(result, n, x + 1, board, columns, diagonals1, diagonals2);
            board[x] = -1;
            columns.remove(y);
            diagonals1.remove(x - y);
            diagonals2.remove(x + y);
        }
    }
```

回溯法应用: 全排列

LeetCode 46. 全排列 中等

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

```java
class Solution {
    List<List<Integer>> result;

    public List<List<Integer>> permute(int[] nums) {
        this.result = new ArrayList<>();
        dfs(nums, 0);
        return result;
    }

    void dfs(int[] nums, int index) {
        if (index == nums.length - 1) {
            List<Integer> permutation = new ArrayList<>();
            for (int num : nums)
                permutation.add(num);
            result.add(permutation);
        } else {
            for (int i = index; i < nums.length; i++) {
                swap(nums, index, i);
                dfs(nums, index + 1);
                swap(nums, index, i); // backtracking
            }
        }
    }

    void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

回溯法应用: 全排列

LeetCode 39. 组合总和 中等

给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的数字可以无限制重复被选取。

说明：
所有数字（包括 target）都是正整数。
解集不能包含重复的组合。

示例：

```
输入: candidates = [2,3,6,7], target = 7
输出:
[
  [7],
  [2,2,3]
]
```

```java
class Solution {
    public List<List<Integer>> combinationSum(int[] candidates, int
target) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        List<Integer> combine = new ArrayList<Integer>();
        dfs(candidates, target, result, combine, 0);
        return result;
    }

    public void dfs(int[] candidates, int target, List<List<Integer>>
result, List<Integer> combine, int idx) {
        if (idx == candidates.length) {
            return;
        }
        if (target == 0) {
            result.add(new ArrayList<Integer>(combine));
            return;
        }
        // 直接跳过
        dfs(candidates, target, result, combine, idx + 1);
        // 选择当前数
        if (target - candidates[idx] >= 0) {
            combine.add(candidates[idx]);
            dfs(candidates, target - candidates[idx], result, combine,
idx);
            combine.remove(combine.size() - 1);
        }
    }
}
```

位运算

二进制数的一元和二元操作

位运算（&、|、^、~、>>、<<）

符号	描述	运算规则
&	与	两个位都为1时，结果才为1
|	或	两个位都为0时，结果才为0
^	异或	两个位相同为0，相异为1
~	取反	0变1，1变0
<<	左移	各二进位全部左移若干位，高位丢弃，低位补0
>>	右移	各二进位全部右移若干位，对无符号数，高位补0; 有符号数，各编译器处理方法不一样，有的补符号位（算术右移），有的补0（逻辑右移）

参考: cs.cmu.edu/~213 *Computer Systems: A Programmer's Perspective, Third Edition*

位运算

- **按位与运算符（&）**
  - 清零
    - X & 0 = 0
  - 取一个数的指定位
    - 取低四位：X & 0000 1111
  - 判断奇偶
    - X & 1
- **按位或运算符（|）**
  - 常用来对一个数据的某些位设置为1
    - 设置低四位：X | 0000 1111
- **取反运算符（~）**
  - 使一个数的最低位为零
    - a & ~1。~1的值为 1111 1111 1111 1110
- **左移运算符（<<）**
  - 左移一位，相当于数乘以2
- **右移运算符（>>）**
  - 右移一位，相当于操作数除以2
- **异或运算符（^）**
  - 翻转指定位
  - 与0相异或值不变
  - 交换两个数

位运算: 异或 xor

定义: 参加运算的两个数据，按二进制位进行**异或**运算

运算规则: `0^0=0  0^1=1  1^0=1  1^1=0`

异或的几条性质:

1 交换律 `a ^ b == b ^ a`

2 结合律 `(a ^ b) ^ c == a ^ (b ^ c)`

3 对于任何数x，都有 `x ^ x = 0, x ^ 0 = x`

4 自反性: `a ^ b ^ b = a ^ 0 = a`

位运算: 异或 xor

异或运算的用途

翻转指定位

比如将数 X=1010 1110 的低4位进行翻转，只需要另找一个数Y，令Y的低4位为1，其余位为0，即 Y=0000 1111，然后将X与Y进行异或运算（X^Y=1010 0001）即可得到。

与0相异或值不变

例如：1010 1110 ^ 0000 0000 = 1010 1110

交换两个数

```
void Swap(int &a, int &b){
    if (a != b){
        a ^= b;
        b ^= a;
        a ^= b;
    }
}
```

位运算: 例

LeetCode 136. 只出现一次的数字 简单

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

示例：

输入： [2,2,1]
输出： 1

```
class Solution {
    public int singleNumber(int[] nums) {
        int single = 0;
        for (int num : nums) {
            single ^= num;
        }
        return single;
    }
}
```

位运算: 例

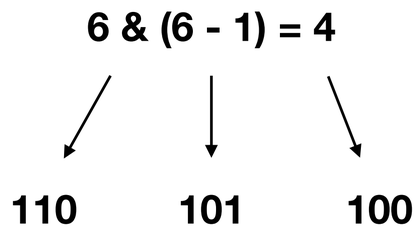
LeetCode 191. 位1的个数 简单

编写一个函数，输入是一个无符号整数（以二进制串的形式），返回其二进制表达式中数字位数为 '1' 的个数（也被称为汉明重量）。

示例：

输入： 00000000000000000000000000001011
输出： 3
解释： 输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。

```java
public class Solution {
    public int hammingWeight(int n) {
        int ret = 0;
        while (n != 0) {
            n &= n - 1;
            ret++;
        }
        return ret;
    }
}
```



位运算: 例

LeetCode 260. 只出现一次的数字III 中等

给定一个整数数组 nums，其中恰好有两个元素只出现一次，其余所有元素均出现两次。 找出只出现一次的那两个元素。你可以按 任意顺序 返回答案。

示例：

输入： nums = [1,2,1,3,2,5]
输出： [3,5]
解释： [5, 3] 也是有效的答案。

思路

1. 先对所有数字进行一次异或，得到两个出现一次的数字的异或值
2. 在异或结果中找到任意为 1 的位
3. 根据这一位对所有的数字进行分组
4. 在每个组内进行异或操作，得到两个数字

```java
class Solution {
    public int[] singleNumber(int[] nums) {
        int ret = 0;
        for (int n : nums) {
            ret ^= n;
        }
        int div = 1;
        while ((div & ret) == 0) {
            div <<= 1;
        }
        int a = 0, b = 0;
        for (int n : nums) {
            if ((div & n) != 0) {
                a ^= n;
            } else {
                b ^= n;
            }
        }
        return new int[]{a, b};
    }
}
```