```
/* 参考博客：http://blog.csdn.net/sunao2002002/article/details/47089053 */
1.时间：O（N）；空间：O（1）    ->错误，数据例子[1,0]
/* 累乘相关的题目特别要注意，数组元素中是否存在 0，
    如果还涉及除法，首先要判断被除数是否是 0 */
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        if (nums.empty()) return std::vector<int>();

        std::vector<int> result;
        int zeroCount = 0;    /* 记录 nums 中 0 的个数 */
        int product = 1; //除 0 外，所有元素的乘积
        for (int i = 0; i < nums.size(); ++i){
            if (nums[i] != 0){
                product *= nums[i];
            } else{
                zeroCount++;
            }
        }
        /* 如果 0 的个数 >= 2，那么将 nums 全置 0 */
        if (zeroCount >= 2){
            nums = std::vector<int>(nums.size(), 0);
```

```
            return nums;
        }

        for (int i = 0; i < nums.size(); ++i){
            nums[i] = nums[i] == 0 ? product : (product / nums[i]);
        }

        return nums;
    }
};
```

2.时间：O（N）；空间：O（1）　->使用除法...

```
/*  累乘相关的题目特别要注意，数组元素中是否存在 0，
    如果还涉及除法，首先要判断被除数是否是 0 */
class Solution {
    /*  分析：1.如果数组有两个及以上的 0，那么结果数组的每一项都为 0
        2.如果数组中只有一个 0，那么结果数组中，0 对应的位置，为数组中其它元素的乘积，其余位置均为 0
        3.如果数组中没有 0，那么结果数组每一项均为数组所有元素的乘积/当前元素

        需要扫描两次，算法时间复杂度为 O（N）；
        第一次：计算 0 的个数，以及非 0 元素的乘积
        第二次：计算每一位的元素
```

```
    */
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        if (nums.empty()) return std::vector<int>();

        std::vector<int> result;
        int zeroCount = 0;    /* 记录 nums 中 0 的个数 */
        int product = 1; //除 0 外，所有元素的乘积
        for (int i = 0; i < nums.size(); ++i){
            if (nums[i] != 0){
                product *= nums[i];
            } else{
                zeroCount++;
            }
        }
        /* 如果 0 的个数 >= 2，那么将 nums 全置 0 */
        /*if (zeroCount >= 2){
            nums = std::vector<int>(nums.size(), 0);
            return nums;
        }*/

        for (int i = 0; i < nums.size(); ++i){
```

```
                if (nums[i] != 0){
                    if (zeroCount != 0) nums[i] = 0;
                    else nums[i] = product / nums[i];
                } else{
                    if (zeroCount == 1) nums[i] = product; /* nums 内有 2 个及以上的 0
*/
                    else nums[i] = 0;
                }
            }

            return nums;
        }
};
```

3.时间：O（N）；空间：O（N）　->使用额外的线性空间

```
/*  累乘相关的题目特别要注意，数组元素中是否存在 0，
    如果还涉及除法，首先要判断被除数是否是 0 */
class Solution {
    /*   1.分别计算两个数组，left_product，right_product，left_product[i]为 nums[i]
左边的元素的乘积，
            right_product[i]为 nums[i]右边元素的乘积；
        2.nums[i] = left_product[i] * right_product[i]
    */
```

```
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        if (nums.empty()) return std::vector<int>();

        std::vector<int> left_product(nums.size(), 1);   /* 记录位置 i 左边的元素的乘积 */
        std::vector<int> right_product(nums.size(), 1); /* 记录位置 i 右边的元素的乘积 */
        for (int i = 1; i < nums.size(); ++i)
            left_product[i] = left_product[i - 1] * nums[i - 1];
        for (int i = nums.size() - 2; i >= 0; --i)
            right_product[i] = right_product[i + 1] * nums[i + 1];

        for (int i = 0; i < nums.size(); ++i){
            nums[i] = left_product[i] * right_product[i];
        }

        return nums;
    }
};
```

4.时间：O（N）；空间：O（1）

```
/* 累乘相关的题目特别要注意，数组元素中是否存在 0，
```

```
    如果还涉及除法，首先要判断被除数是否是 0 */
class Solution {
    /*   1.返回结果的空间不计入额外空间；
         2.返回的数组 result,result[i]表示 i 左边的元素的乘积
         3.从右向左遍历[n-2->0]，用变量 right_product 记录 i 右边的元素的乘积，那么
result[i] = result[i]（区间[0~i-1]乘积）*right_product;
         这个类似与动态规划中的"滚动数组"，以降低空间消耗
    */
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        if (nums.empty()) return std::vector<int>();

        std::vector<int> result(nums.size(), 1);
        /* 初始化 result 作为左边元素乘积 */
        for (int i = 1; i < nums.size(); ++i){
            result[i] = result[i - 1] * nums[i - 1];
        }
        int right_product = 1;
        /* 反向计算结果 */
        for (int i = nums.size() - 2; i >= 0; --i){
            right_product *= nums[i + 1];
            result[i] *= right_product;
```

```
            }


            return result;

        }

};
```

总结：1.由简单->复杂，逐步优化时间和空间效率；

2.思考边界数据，并验证程序；