在这一小节，大家需要关注到的是“从具体问题中抽象算法模型”这个能力。
直白点说，有一类题目，它们（看上去）来者不善：题干不仅天马行空，有时候还又臭又长，导致你读了五分钟很可能也只读出了一个屁——这类题目其实就是在考察你把具体问题抽象为算法模型的能力。
遇到它，你除了不要慌、不要怕之外，最重要的是不要被题目牵着鼻子走。你得拉拢它，收买它，把它往你已经掌握的那些知识点上靠——很多时候，同学们缺少的并不是知识储备，而是【建立题目与知识点之间的关联】的能力。

岛屿数量问题

> 题目描述：给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。
> 岛屿总是被水包围，并且每座岛屿只能由水平方向或竖直方向上相邻的陆地连接形成。
> 此外，你可以假设该网格的四条边均被水包围。

> 示例 1:
> 输入:
> 11110
> 11010
> 11000
> 00000
> 输出: 1

> 示例 2:
> 输入:
> 11000
> 11000
> 00100
> 00011
> 输出: 3
> 解释: 每座岛屿只能由水平和/或竖直方向上相邻的陆地连接而成。

命题关键字：模拟、DFS

思路分析

这道题好就好在它题目不长，但是题干这通描述有可能会让一部分同学直接失去耐心——岛屿？水？“网格”？？？

啥啥啥？这都是啥？

其实，只要同学能够耐住性子读下去，就会发现所谓“网格”不过是二维数组，而“岛屿”和“水”这样的具体概念，题目也已经贴心地帮我们抽象为了“1”和“0”这样简单的数字。因此，我们拿到这道题，首先要做的就是把题目中这些干扰性的概念“翻译”成简单直接的算法语言：

> 已知一个二维数组，定义“相互连接的1”为一个块（这里的相互连接，意思就是1和1之间可以不经过0就相互抵达），求符合条件的块的数量。

翻译到这个程度之后，我们不难找出“相互连接”这个关键词作为我们做题的抓手，进而去形成一个初步的思路——若当前所在位置是1，从1出发，可以抵达的**所有**1都和它算作同一个岛屿。注意这里我把“所有”这个词标了粗体，已经读了25节算法小册的你，请和我一起大声喊出下面这句话：

> 看到“所有”，必须想到“枚举”！看到“枚举”，必须回忆起 DFS 和 BFS ！

喜欢递归的我，选择用 DFS 来做~~~

在明确了 DFS 的大方向之后，结合题意，我们可以提取出以下关键问题：

1. 如何实现对不同岛屿的统计?
2. 已经计算过的岛屿如何排除?

下面我一一回答这两个问题：

1. 岛屿的统计思路：从起点出发，遵循“不撞水面（也就是0）不回头”的原则，枚举当前可以触及的所有1。当枚举无法继续进行时，说明当前这座岛屿被遍历完毕，记为一个。**也就是说每完成一次 DFS ，就累加一个岛屿**。
2. 避免重复计算的方法：每遍历过一个1，就把它置为0，后续再次路过时就会自动忽略它啦~~

回答完这俩问题，代码也算基本写完了（如果以上描述仍然无法帮你建立清晰的思路，不妨去代码注释里找一下答案~）：

编码实现

```
/**
 * @param {character[][]} grid
 * @return {number}
 */
// 入参是二维数组
const numIslands = function(grid) {
  const moveX = [0, 1, 0, -1]
  const moveY = [1, 0, -1, 0]
  // 处理二维数组的边界情况
  if(!grid || grid.length === 0 || grid[0].length === 0) {
      return 0
  }
  // 初始化岛屿数量
```

```
    let count = 0
    // 缓存二维数组的行数和列数
    let row = grid.length, column = grid[0].length
    // 以行和列为线索，尝试“逐个”遍历二位数组中的坑位
    for(let i=0; i<row; i++) {
        for(let j=0; j<column; j++) {
            if(grid[i][j] === '1') {
                // 每遇到1，就进入dfs，探索岛屿边界
                dfs(grid, i, j)
                // 每完成一个 dfs，就累加一个岛屿
                count++
            }
        }
    }
    return count

    // 编写探索岛屿边界的逻辑
    function dfs(grid, i, j) {
        // 如果试图探索的范围已经越界，则return
        if(i<0 || i>=grid.length || j<0 || j>=grid[0].length || grid[i][j]
            return
        }
        // 遍历过的坑位都置0，防止反复遍历
        grid[i][j] = '0'
        // 遍历完当前的1，继续去寻找下一个1
        for(let k=0; k<4; k++) {
            dfs(grid, i+moveX[k], j+moveY[k])
        }
    }
}
```

编码复盘

对初学此类问题的同学来说，这道题里有一个值得关注的做题技巧，就是对 `moveX` 和 `moveY` 两个数组的设定：

```
const moveX = [0, 1, 0, -1]
const moveY = [1, 0, -1, 0]
```

结合代码的上下文可以看出，我们借助这两个数组，可以完成对当前格子的“垂直”和“水平”两个方向上的相邻格子的检查：

```
for(let k=0; k<4; k++) {
  dfs(grid, i+moveX[k], j+moveY[k])
}
```

后续我们遇到的一些题目，一旦和这道题一样，强调了“水平”、“垂直”方向上的相邻关系，我们就可以无脑复用这个套路啦~

“扫地机器人”问题

> 题目描述：房间（用格栅表示）中有一个扫地机器人。格栅中的每一个格子有空和障碍物两种可能。
> 扫地机器人提供4个API，可以向前进，向左转或者向右转。每次转弯90度。
> 当扫地机器人试图进入障碍物格子时，它的碰撞传感器会探测出障碍物，使它停留在原地。
> 请利用提供的4个API编写让机器人清理整个房间的算法。

```
interface Robot {
  // 若下一个方格为空，则返回true，并移动至该方格
  // 若下一个方格为障碍物，则返回false，并停留在原地
  boolean move();

  // 在调用turnLeft/turnRight后机器人会停留在原位置
  // 每次转弯90度
  void turnLeft();
  void turnRight();

  // 清理所在方格
  void clean();
}
```

> 示例:
> 输入:
> room = [
> [1,1,1,1,1,0,1,1],
> [1,1,1,1,1,0,1,1],
> [1,0,1,1,1,1,1,1],
> [0,0,0,1,0,0,0,0],
> [1,1,1,1,1,1,1,1]
>],
> row = 1,
> col = 3
> 解析: 房间格栅用0或1填充。0表示障碍物，1表示可以通过。 机器人从row=1，col=3的初始位置出发。在左上角的一行以下，三列以右。

> 注意:
> 输入只用于初始化房间和机器人的位置。你需要“盲解”这个问题。换而言之，你必须在对房间和机器人位置一无所知的情况下，只使用4个给出的API解决问题。
> 扫地机器人的初始位置一定是空地。
> 扫地机器人的初始方向向上。
> 所有可抵达的格子都是相连的，亦即所有标记为1的格子机器人都可以抵达。
> 可以假定格栅的四周都被墙包围。

命题关键字：模拟、DFS

思路分析

这道题很明显属于我们开篇提到过的“又臭又长”系列。但笔者相信，每一个做过“岛屿数量”的同学，在面对这道题的时候，至少心里是不会怂的。 毕竟，它们也长得太像了：

```
room = [
  [1,1,1,1,1,0,1,1],
  [1,1,1,1,1,0,1,1],
  [1,0,1,1,1,1,1,1],
  [0,0,0,1,0,0,0,0],
  [1,1,1,1,1,1,1,1]
]
```

变化的题干，不变的1&0二维数组，嘿嘿嘿。

现在我们回头研究一下题干。
前面咱们说过，对于这种场景感比较强的具体问题，最要紧的是从冗长的描述中去提取出确切的算法问题。因此大家最好先尝试自己先翻译一下题干，想想它到底想让你干嘛。

这里我先给出自己做这道题时的脑回路，大家不妨观察一下这个思考的过程，**你会发现这些思路其实都不是从天而降的，而是来源于对已经做过的题目的吸收和反思**：

整体思路

这道题涉及到对二维数组网格的枚举，可能与岛屿数量问题的基本思路一致（将 `DFS` 作为优先方法来考虑）。虽然我不知道对不对，但是沿着这个思路往下多分析几步总是好的：

1. 机器人从初始位置出发，检查上下左右四个方向是否有障碍物，进而决定是否进入对应方向的格子完成清扫。
2. 因为题目强调了“所有可抵达的格子都是相连的，亦即所有标记为1的格子机器人都可以抵达”，所以我们借助 `DFS` 尝试枚举所有可抵达的格子是完全没有问题的。 `DFS` 的主要递归逻辑其实就是步骤1。
3. 当某一个方向已经“撞到南墙”后，机器人应该逐渐回溯到上一个位置，尝试新的方向。
4. 最后，由于递归边界其实就是障碍物/已经清扫过的格子。所以别忘了对已经清扫过的格子做个标记。

整个思路捋下来，没有逻辑上的硬伤。下面我试着对具体问题进行分析，看看实现起来有没有什么困难。

机器人的前进规则分析

题目的复杂之处在于强调了“上下左右”的概念，它要求我们先旋转、再判断、最后根据判断结果决定是否需要前进。也就是说，我们不仅需要考虑机器人的前进坐标，还需要考虑机器人的旋转角度。其实无论旋转也好、前进也罢，本质上都是让它“自己动”。大家记住，“自己动”往往和循环有关。比如说上一道题里，我们就是用一个固定 `k=4` 的循环来完成了上下左右四个方向的前进：

```
for(let k=0; k<4; k++) {
  dfs(grid, i+moveX[k], j+moveY[k])
}
```

在这道题里，我们同样可以用类似的循环来实现四个方向的旋转+前进。
明确了循环结构的设计，现在继续来分析循环体。
既然题目已经把步骤拆成了旋转和前进两步，那么我们就有必要把旋转角度和前进坐标之间的关系对应起来。假设机器人现在所在的格子坐标是 `(i, j)` ，它的旋转角度以及对应的前进坐标之间就有以下关系（结合题意我们把“上”这个初始方向记为0度）：

```
（定义逻辑）
// 初始化角度为 0 度
let dir = 0
...

（判断逻辑）
// 将角度和前进坐标对应起来
```

```
    switch(dir) {
        // 0度的前进坐标，是当前坐标向上走一步
        case 0:
            x = i - 1
            break
        // 90度（顺时针）的前进坐标，是当前坐标向右走一步
        case 90:
            y = j + 1
            break
        // 180度（顺时针）的前进坐标，是当前坐标向下走一步
        case 180:
            x = i + 1
            break
        // 270度（顺时针）的前进坐标，是当前坐标向左走一步
        case 270:
            y = j - 1
            break
        default:
            break
    }
    ...
    （叠加逻辑）
    // 注意这里我给机器人的规则就是每次顺时针转一个方向，所以是 turnRight
    robot.turnRight()
    // turnRight的同时，dir需要跟着旋转90度
    dir += 90
    // 这里取模是为了保证dir在[0, 360]的范围内变化
    dir %= 360
```

如何优雅地对已处理过的格子做标记

请思考一下，在这道题里，是否还可以沿用“岛屿数量”问题中直接修改二维数组的思路？说实话，没试过，也不建议——就这道题来说，题给的四个 `API` 都不是我们自己实现的，一旦改了全局的 `room` 变量，谁知道会影响哪个 `API` 呢。保险起见，我们应该优先考虑不污染 `room` 变量的实现方法，这里我借助的是 `Set` 数据结构：

```
（以下是定义逻辑）
//初始化一个 set 结构来存储清扫过的坐标
const boxSet =  new Set()
...

（以下是判断逻辑）
// 标识当前格子的坐标
let box = i + '+' + j
// 如果已经打扫过，那么跳过
if(boxSet.has(box)) return
// 打扫当前这个格子
robot.clean()
// 记住这个格子
boxSet.add(box)
```

OK，分析完这三个大问题，我们的代码也基本写完了：

编码实现

```
/**
 * @param {Robot} robot
 * @return {void}
 */
const cleanRoom = function(robot) {
    // 初始化一个 set 结构来存储清扫过的坐标
    const boxSet =  new Set()
    // 初始化机器人的朝向
    let dir = 0
    // 进入 dfs
    dfs(robot, boxSet, 0, 0, 0)

    // 定义 dfs
    function dfs(robot, boxSet, i, j, dir) {
        // 记录当前格子的坐标
        let box = i + '+' + j
        // 如果已经打扫过，那么跳过
```

```
if(boxSet.has(box)) return
// 打扫当前这个格子
robot.clean()
// 记住这个格子
boxSet.add(box)

// 四个方向试探
for(let k=0;k<4;k++) {
    // 如果接下来前进的目标方向不是障碍物（也就意味着可以打扫）
    if(robot.move()) {
        // 从当前格子出发，试探上右左下
        let x = i, y = j
        // 处理角度和坐标的对应关系
        switch(dir) {
            case 0:
                x = i - 1
                break
            case 90:
                y = j + 1
                break
            case 180:
                x = i + 1
                break
            case 270:
                y = j - 1
                break
            default:
                break
        }
        dfs(robot, boxSet, x, y, dir)
        // 一个方向的dfs结束了，意味着撞到了南墙，此时我们需要回溯到上一个
        robot.turnLeft()
        robot.turnLeft()
        robot.move()
        robot.turnRight()
        robot.turnRight()
    }
```

```
                // 转向
                robot.turnRight()
                dir += 90
                dir %= 360

            }
        }
    }
```

编码复盘

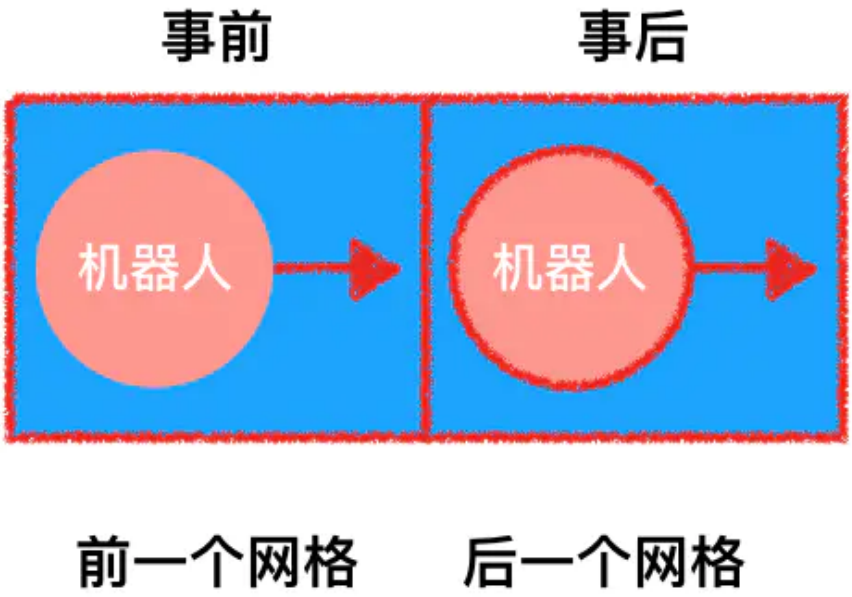
这里有一段逻辑可能会让初学题目的同学蒙圈：

```
dfs(robot, boxSet, x, y, dir)
robot.turnLeft()
robot.turnLeft()
robot.move()
robot.turnRight()
robot.turnRight()
```

这是在干啥?
结合一下代码的上下文，这里我给机器人的设定是：

> 你在进入每一个格子后，都需要基于当前方向顺时针旋转四次

在这个前提下，机器人在 `(x,y)` 这个格子工作完之后，它的朝向一定是和刚进入 `(x,y)` 时的朝向是一样的，区别在于在原来的基础上多走了一个格子：



此时后一个网格的机器人要想退回“事前”的状态，它必须先旋转 180 度，然后前进一步，再旋转 180 度。而“旋转 180 度”这个动作，可以通过连续两次 `turnLeft` 或者 `turnRight` 来完成。这里我为了写代码好看，各用了一次（羞）。

“合并区间”问题

> 题目描述：给出一个区间的集合，请合并所有重叠的区间。

> 示例 1:
> 输入: [[1,3],[2,6],[8,10],[15,18]]
> 输出: [[1,6],[8,10],[15,18]]
> 解释: 区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].

> 示例 2:
> 输入: [[1,4],[4,5]]
> 输出: [[1,5]]
> 解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。

命题关键字：数学问题、数组

思路分析

做完两道应用题，大家放松一下，换换口味，现在我们来一起解决一个并没有许多套路的数学问题。这个题里，你什么都可以忽略，但是请一定抓住“区间”二字，并记住下面这样一个规律：

> 对于区间类问题，尝试以区间内的第一个元素为索引进行排序，往往可以帮助我们找到问题的突破点

不信我们来看看这道题，题中给了我们这样一个例子：

```
[[1,3],[2,6],[8,10],[15,18]]
```

这个例子就是一个排序过的区间，当区间排序后，区间与区间之间的重叠关系会变得非常有迹可循:

```
[1, 3]
  [2, 6]
            [8, 10]
                      [15, 18]
```

可以看出，对于有序区间，我们其实可以从头开始，**逐个合并首尾有交集的区间**——比如上面区间关系图中的 `[1, 3]` 和 `[2, 6]`，由于前一个区间的尾部（3）和下一个区间的头部（2）是有交错关系的（这个交错关系用数学语言表达出来就是 `前一个的尾部 >= 下一个的头部` ），因此我们可以毫不犹豫地把它们合并为一个区间：

```
[1, 3] + [2, 6] ==> [1, 6]
```

遵循这个合并规则，我们可以编码如下：

编码实现

```
/**
 * @param {number[][]} intervals
 * @return {number[][]}
 */
const merge = function(intervals) {
```

```
    // 定义结果数组
    const res = []
    // 缓存区间个数
    const len = intervals.length
    // 将所有区间按照第一个元素大小排序
    intervals.sort(function(a, b) {
        return a[0] - b[0]
    })
    // 处理区间的边界情况
    if(!intervals || !intervals.length) {
        return []
    }
    // 将第一个区间（起始元素最小的区间）推入结果数组（初始化）
    res.push(intervals[0])
    // 按照顺序，逐个遍历所有区间
    for(let i=1; i<len; i++) {
        // 取结果数组中的最后一个元素，作为当前对比的参考
        prev = res[res.length-1]
        // 若满足交错关系（前一个的尾部 >= 下一个的头部）
        if(prev[1] >= intervals[i][0]) {
            prev[1] = Math.max(prev[1], intervals[i][1])
        } else {
            res.push(intervals[i])
        }
    }
    return res
}
```