```swift
import Foundation
// 剑指 Offer 12. 矩阵中的路径
// 给定一个 m x n 二维字符网格 board 和一个字符串单词 word 。
// 如果 word 存在于网格中，返回 true ；否则，返回 false 。
//
// 单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单
// 元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。
//
// 例如，在下面的 3×4 的矩阵中包含单词 "ABCCED"（单词中的字母已标出）。
// 示例 1：
// 输入：board = [["A","B","C","E"],
//               ["S","F","C","S"],
//               ["A","D","E","E"]],
//      word = "ABCCED"
// 输出：true
//
// 示例 2：
// 输入：board = [["a","b"],
//               ["c","d"]],
//      word = "abcd"
// 输出：false
//
// 提示：
// m == board.length
// n = board[i].length
// 1 <= m, n <= 6
// 1 <= word.length <= 15
// board 和 word 仅由大小写英文字母组成
class Solution {
    func existSolution1(_ board: [[Character]], _ word: String) -> Bool {
        if board.isEmpty { return false }
        if (board.first == nil) || (board.first!.isEmpty) { return false }
        var _board = board
        for i in 0..<_board.count {
            for j in 0..<_board[0].count {
                if(dfs(i, j, 0, &_board, word)) { return true }
            }
        }
        return false
    }

    func dfs(_ i: Int, _ j: Int, _ k: Int, _ board: inout [[Character]], _
     word: String) -> Bool {
        let startIndex = word.startIndex
        /// 如果`board[i][j] == word[k]`，则表明当前找到了对应的数，
        /// 就继续执行（标记找过，继续`dfs`下上右左）
        if (i < 0 || i >= board.count ||
            j < 0 || j >= board[0].count ||
            board[i][j] != word[word.index(startIndex, offsetBy: k)]) {
```

```
        return false
    }

    if k == word.count - 1 { return true }
    /// 访问过的标记空字符串，“ ”是空格 '\0'是空字符串，不一样的！
    /// 比如当前为A，没有标记找过，且A是word中对应元素，则此时应该找A下一个元素，
    /// 假设是B，在dfs（B）的时候还是-->要搜索B左边的元素（假设A在B左边），
    /// 所以就是ABA（凭空多出一个A，A用了2次，不可以），如果标记为空字符串->
    /// 就不会有这样的问题，因为他们值不相等AB != ABA。
    board[i][j] = "\0";
    /// 顺序是 下 上 右 左；上面找到了对应索引的值所以k+1
    let res = dfs(i+1, j, k+1, &board, word) || dfs(i-1, j, k+1, &board,
     word) || dfs(i, j+1, k+1, &board, word) || dfs(i, j-1, k+1, &board,
     word)

    /// 还原找过的元素，因为之后可能还会访问到（不同路径）
    board[i][j] = word[word.index(startIndex, offsetBy: k)]
    return res
}

func exist(_ board: [[Character]], _ word: String) -> Bool {
    let maxY = board.count
    let chars: [Character] = Array(word)

    guard let maxX = board.first?.count, maxY > 0, maxX > 0 else { return
     false }

    var visited: [[Bool]] = Array(repeating: Array(repeating: false,
     count: maxX), count: maxY)

    var pathLength = 0

    for y in 0..<maxY {
        for x in 0..<maxX {
            if hasPathCore(board, chars, maxX: maxX, maxY: maxY,
             pathLength: &pathLength, x: x, y: y, visted: &visited) {
                return true
            }
        }
    }

    return false
}

func hasPathCore(_ board: [[Character]],
                 _ chars: [Character],
                 maxX: Int, maxY: Int,
                 pathLength: inout Int,
                 x: Int, y: Int,
                 visted: inout [[Bool]]) -> Bool {
```

```
        if pathLength == chars.count { return true }

        var result = false
        if x >= 0, y >= 0,
            x < maxX, y < maxY,
            visted[y][x] == false,
            board[y][x] == chars[pathLength] {
            pathLength += 1
            visted[y][x] = true

            result =
                hasPathCore(board, chars, maxX: maxX, maxY: maxY, pathLength:
                 &pathLength, x: x - 1, y: y, visted: &visted) ||
                hasPathCore(board, chars, maxX: maxX, maxY: maxY, pathLength:
                 &pathLength, x: x + 1, y: y, visted: &visted) ||
                hasPathCore(board, chars, maxX: maxX, maxY: maxY, pathLength:
                 &pathLength, x: x, y: y - 1, visted: &visted) ||
                hasPathCore(board, chars, maxX: maxX, maxY: maxY, pathLength:
                 &pathLength, x: x, y: y + 1, visted: &visted)

            if result == false {
                pathLength -= 1
                visted[y][x] = false
            }
        }
        return result
    }
}
```