Flood Fill算法

构建框架

图片等可以抽象为一个二维矩阵，然后从某个点开始向四周开始扩散，直到无法扩散为止。 矩阵可以抽象为图片，这就转化成图的遍历问题，类似一个N叉树的遍历问题。

```
// (x,y)为坐标
int fill(int x, int y){
  fill(x-1,y); // 向上
  fill(x+1,y); // 向下
  fill(x, y-1); // 向左
  fill(x, y+1); // 向右
}
```

这个框架可以视为深度优先遍历，或者四叉树遍历问题。 例：leetcode颜色填充

编写函数，实现许多图片编辑软件都支持的「颜色填充」功能。

待填充的图像用二维数组 image 表示，元素为初始颜色值。初始坐标点的行坐标为 sr 列坐标为 sc。需要填充的新颜色为 newColor 。

「周围区域」是指颜色相同且在上、下、左、右四个方向上存在相连情况的若干元素。

请用新颜色填充初始坐标点的周围区域，并返回填充后的图像。

示例:

输入:
image = [[1,1,1],[1,1,0],[1,0,1]]
sr = 1, sc = 1, newColor = 2
输出: [[2,2,2],[2,2,0],[2,0,1]]
解释:
初始坐标点位于图像的正中间，坐标 (sr,sc)=(1,1) 。
初始坐标点周围区域上所有符合条件的像素点的颜色都被更改成 2 。
注意，右下角的像素没有更改为 2 ，因为它不属于初始坐标点的周围区域。

提示:

    image 和 image[0] 的长度均在范围 [1, 50] 内。
    初始坐标点 (sr,sc) 满足 0 <= sr < image.length 和 0 <= sc < image[0].length 。
    image[i][j] 和 newColor 表示的颜色值在范围 [0, 65535] 内。

解法：

```
class Solution {
public:
  std::vector<std::vector<int>> floodfill(std::vector<std::vector<int>>
&board,
                                          int
sr,
                                          int
sc,
                                          int newColor) {
    int oldColor = board[sr][sc];
    int row      = board.size();
    int col      = 0;

    if (row > 0) {
      col = board[0].size();
    }

    fillImage(board, row, col, sr, sc, oldColor, newColor);

    return board;
  }

private:
  void fillImage(std::vector<std::vector<int>> &board,
                 int                           row,
                 int                           col,
                 int                           sr,
                 int                           sc,
                 int                           oldColor,
                 int                           newColor) {
    // 越界
    if (!isInArea(sr, sc, row, col)) {
      return;
    }

    // 当前值不为oldColor
    if (oldColor != board[sr][sc]) {
      return;
    }
    board[sr][sc] = -1; // 替代visited[][]
    fillImage(board, row, col, sr - 1, sc, oldColor, newColor);
    fillImage(board, row, col, sr + 1, sc, oldColor, newColor);
    fillImage(board, row, col, sr, sc - 1, oldColor, newColor);
    fillImage(board, row, col, sr, sc + 1, oldColor, newColor);

    board[sr][sc] = newColor;
  }

  bool isInArea(int sr, int sc, int row, int col) {
    return sr >= 0 && sr < row && sc >= 0 && sc < col;
```

```
    }
};
```

扩展： 如何只填充边界部分，不填充内部。

```
int fill(std::vector<std::vector<int>> &image, int x, int y,int
orginColor, int newColor){
  // 出界
  if(!isInArea(image,x,y)) {
    return 0;
  }

  // 已探索过 orginColor区域
  if(visited[x][y]) {
    return 1;
  }

  // 遇到其他颜色
  if (orginColor != image[x][y]){
    return 0;
  }

  visited[x][y] = 1;

  int surround = fill(image, x-1, y, orginColor, newColor) +
                 fill(image, x+1, y, orginColor, newColor) +
                 fill(image, x, y-1, orginColor, newColor) +
                 fill(image, x, y+1, orginColor, newColor);

  // 边界点
  if(surround < 4 ) {
    image[x][y] = newColor;
  }

  return 1;
}
```

抽象出框架：

```
int fill(std::vector<std::vector<int>> &image, int x, int y,int
orginColor, int newColor){
  // 出界
  if(!isInArea(image,x,y)) {
    return 0;
  }

  // 已探索过 orginColor区域
  if(visited[x][y]) {
    return 1;
```

```
    }

    // 遇到其他颜色
    if (orginColor != image[x][y]){
      return 0;
    }

    visited[x][y] = 1;

    // 替换为其他对边界点的操作
    int surround = fill(image, x-1, y, orginColor, newColor) +
                   fill(image, x+1, y, orginColor, newColor) +
                   fill(image, x, y-1, orginColor, newColor) +
                   fill(image, x, y+1, orginColor, newColor);

    // 边界点
    if(surround < 4 ) {
      image[x][y] = newColor;
    }

    return 1;
  }
```