文 [Leetcode] Game of Life 生命游戏

算法 leetcode java **ethannnli** 2015年10月05日发布

Game of Life I

> According to the Wikipedia's article: "The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."
>
> Given a board with m by n cells, each cell has an initial state live (1) or dead (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):
>
> Any live cell with fewer than two live neighbors dies, as if caused by under-population. Any live cell with two or three live neighbors lives on to the next generation. Any live cell with more than three live neighbors dies, as if by over-population.. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction. Write a function to compute the next state (after one update) of the board given its current state.
>
> Follow up: Could you solve it in-place? Remember that the board needs to be updated at the same time: You cannot update some cells first and then use their updated values to update other cells. In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches the border of the array. How would you address these problems?

编解码法

复杂度

时间 O(NN) 空间 O(1)

思路

最简单的方法是再建一个矩阵保存，不过当inplace解时，如果我们直接根据每个点周围的存活数量来修改当前值，由于矩阵是顺序遍历的，这样会影响到下一个点的计算。如何在修改值的同时又保证下一个点的计算不会被影响呢？实际上我们只要将值稍作编码就行了，因为题目给出的是一个int矩阵，大有空间可以利用。这里我们假设对于某个点，值的含义为

```
0 : 上一轮是0，这一轮过后还是0
1 : 上一轮是1，这一轮过后还是1
2 : 上一轮是1，这一轮过后变为0
3 : 上一轮是0，这一轮过后变为1
```

这样，对于一个节点来说，如果它周边的点是1或者2，就说明那个点上一轮是活的。最后，在遍历一遍数组，把我们编码再解回去，即0和2都变回0，1和3都变回1，就行了。

注意

- 注意编码方式，1和3都是这一轮过后为1，这样就可以用一个模2操作来直接解码了
- 我实现的时候并没有预先建立一个对应周围8个点的数组，因为实际复杂度是一样，多加几个数组反而程序可读性下降

代码

```
public class Solution {
    public void gameOfLife(int[][] board) {
        int m = board.length, n = board[0].length;
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                int lives = 0;
                // 判断上边
                if(i > 0){
                    lives += board[i - 1][j] == 1 || board[i - 1][j] == 2 ? 1 : 0;
                }
                // 判断左边
```

```
                if(j > 0){
                    lives += board[i][j - 1] == 1 || board[i][j - 1] == 2 ? 1 : 0;
                }
                // 判断下边
                if(i < m - 1){
                    lives += board[i + 1][j] == 1 || board[i + 1][j] == 2 ? 1 : 0;
                }
                // 判断右边
                if(j < n - 1){
                    lives += board[i][j + 1] == 1 || board[i][j + 1] == 2 ? 1 : 0;
                }
                // 判断左上角
                if(i > 0 && j > 0){
                    lives += board[i - 1][j - 1] == 1 || board[i - 1][j - 1] == 2 ? 1 : 0;
                }
                //判断右下角
```

另一种编码方式是位操作，将下轮该cell要变的值存入bit2中，然后还原的时候右移就行了。

```
public void solveInplaceBit(int[][] board){
        int m = board.length, n = board[0].length;
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                int lives = 0;
                // 累加上下左右及四个角还有自身的值
                for(int y = Math.max(i - 1, 0); y <= Math.min(i + 1, m - 1); y++){
                    for(int x = Math.max(j - 1, 0); x <= Math.min(j + 1, n - 1); x++){
                        // 累加bit1的值
                        lives += board[y][x] & 1;
                    }
                }
                // 如果自己是活的，周边有两个活的，lives是3
                // 如果自己是死的，周边有三个活的，lives是3
                // 如果自己是活的，周边有三个活的，lives减自己是3
                if(lives == 3 || lives - board[i][j] == 3){
                    board[i][j] |= 2;
                }
            }
        }
        // 右移就是新的值
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                board[i][j] >>>= 1;
            }
        }
}
```

表优化法

复杂度

时间 O(NN) 空间 O(512)

思路

上面的方法实测都比较慢，对于5000*5000的矩阵计算时间都在600-1000ms，甚至比简单的用buffer的方法慢，我们再介绍一个能将速度提高一倍的方法。一般来说，优化程序有这么几个思路：

- 尽量减少嵌套的循环
- 减少对内存的读写操作

上个解法中，使用多个for循环的就比较慢，如果我们能够直接计算出该点的值而不用for循环就好了。这里我们可以用一个“环境”变量，表示该点所处的环境，这样我们根据它以及它周围八个点的值就可以直接算出它的环境值，而不需要用for循环来检查周围8个点。有人说，这不就只是把读取操作放到循环外面来了吗？其实这只是用了优化了第一点，减少循环，对于第二点我们也有优化，我们计算环境值这样计算，对于以n4为中心的点，其环境为

```
n8  n5  n2
n7  n4  n1
n6  n3  n0
```

则环境值 `environment = n8 * 256 + n7 * 128 + n6 * 64 + n5 * 32 + n4 * 16 + n3 * 8 + n2 * 4 + n1 * 2 + n0 * 1` ，这么做的好处是把每一个格子的死活信息都用一个bit来表示，更巧妙地是当我们计算以n1为中心的环境时，是可以复用这些信息的，我们不用再读取一遍 `n5, n4, n3, n2, n1, n0` 的值，直接将上一次的环境值模上64后再乘以8，就是可以将他们都向左平移一格，这时候再读取三个新的值 `a, b, c` 就行了。

```
n8  n5  n2  a
n7  n4  n1  b
n6  n3  n0  c
```

通过这种方法，我们将内存的读取次数从每个点九次，变成了每个点三次。另外我们还要预先制作一个表，来映射环境值和结果的关系。比如环境值为7时，说明 n2, n1, n0 都是活的，结果应该为1（下一轮活过来）。这里制作表的程序可以这么写：

```
int[] table = new int[512];
for(int i = 0; i < 512; i++){
    int lives = Integer.bitCount(i);
    if(lives == 3 || (lives - ((i & 16) > 0 ? 1 : 0) == 3)){
        table[i] = 1;
    }
}
```

代码

```
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0,
            1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0,
            0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0,
            0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,
            1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    int m = board.length, n = board[0].length;
    if(n == 0) return;
    int[][] buffer = new int[m][n];
    for(int i = 0; i < m; i++){
          // 每一行开始时，先计算初始的环境值（左边两列）
        int environment = (i - 1 >= 0 && board[i - 1][0] == 1? 4 : 0) + (board[i][0] == 1 ? 2 : 0) + (i + 1 < m && board[i + 1][0] == 1 ? 1 : 0);
            // 对该行的每一列，通过加入右边新的一列，来计算该点的环境值
        for(int j = 0; j < n; j++){
              // 将之前的环境值模64再乘以8，然后加上右边新的三列
            environment = (environment % 64) * 8 + (i - 1 >= 0 && j + 1 < n && board[i - 1][j + 1] == 1 ? 4 : 0) + (j + 1 < n && board[i][j + 1] ==
            buffer[i][j] = lookupTable[environment];
        }
    }
    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            board[i][j] = buffer[i][j];
        }
    }
}
```

后续 Follow Up

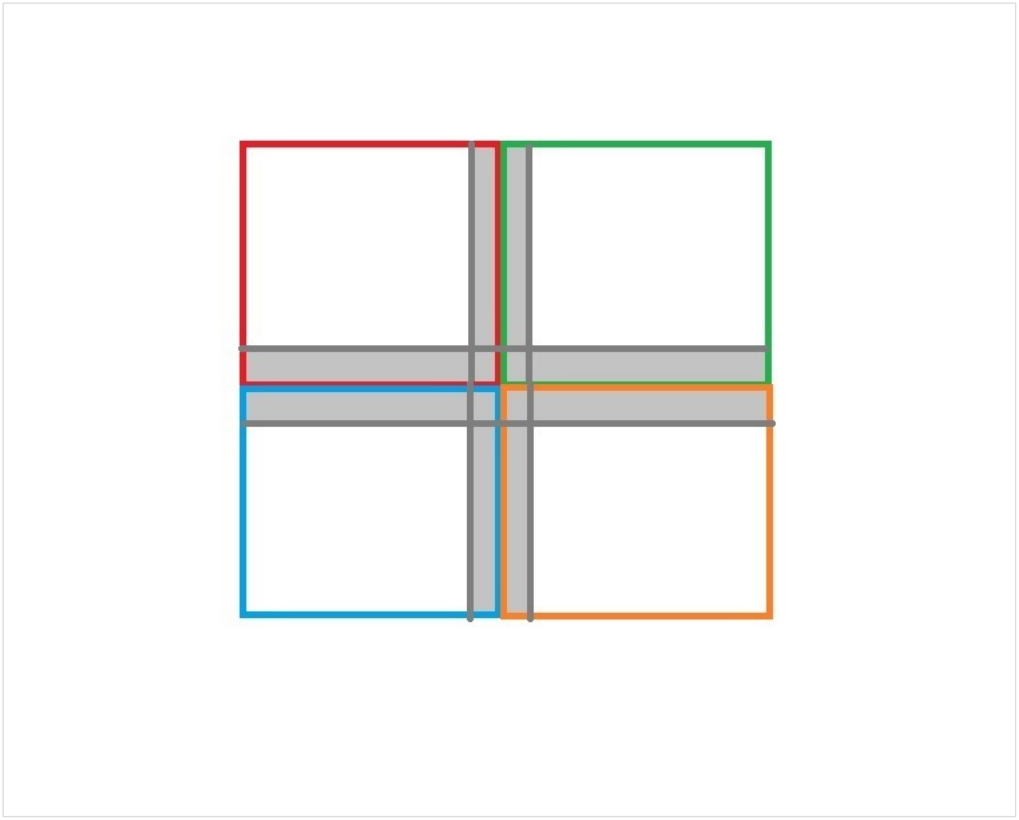
1. 如果循环矩阵如何解决？循环的意思是假设一个3x3的矩阵,则 a[0][0] 的左边是 a[0][1] ，其左上是 a[2][2]
   这样我们的坐标要多加一个数组长度，使用坐标时还要取模

```
public void solveInplaceCircular(int rounds, int[][] board){
    for(int round = 0; round < rounds; round++){
        int m = board.length, n = board[0].length;
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                int lives = 0;
                // 多加一个数组长度
                for(int y = i + m - 1; y <= i + m + 1; y++){
                    for(int x = j + n - 1; x <= j + n + 1; x++){
                        // 使用的时候要取模
                        lives += board[y % m][x % n] & 1;
                    }
                }
                if(lives == 3 || lives - board[i][j] == 3){
                    board[i][j] |= 2;
                }
            }
        }
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                board[i][j] >>>= 1;
            }
        }
    }
}
```

2. 如果矩阵很大如何优化？我们可以只记录存活节点的信息，存入一个live的list中，这里active代表着存活节点，或者存活节点的邻居。每次只计算这个list中节点和其邻居的情况。进一步优化的话，我们可以用一个active的list，只记录上次更新的节点，或者该节点的邻居。等计算完这个列表后，将产生更新的节点和它的邻居们存入一个新

列表中，再用这个新列表里节点的值来更新矩阵。下一轮时，就计算这个新列表，再产生一个新列表。

3. 如果多核的机器如何优化？



因为是多核，我们可以用线程来实现并行计算。如图，将矩阵分块后，每个线程只负责其所在的分块的计算，不过主线程每一轮都要更新一下这些分块的边缘，并提供给相邻分块。所以这里的开销就是主线程和子线程通信这个边缘信息的开销。如果线程变多分块变多，边缘信息也会变多，开销会增大。所以选取线程的数量是这个开销和并行计算能力的折衷。

4. 如果是多台机器如何优化？

同样的，我们可以用一个主机器负责处理边缘信息，而多个子机器处理每个分块的信息，因为是分布式的，我们的矩阵可以分块的存储在不同机器的内存中，这样矩阵就可以很大。而主机在每一轮开始时，将边缘信息通过网络发送给哥哥分块机器，然后分块机器计算好自己的分块后，把新自己内边缘信息反馈给主机器。下一轮，等主机器收集齐所有边缘后，就可以继续重复。

不过多台机器时还有一个更好的方法，就是使用Map Reduce。Map Reduce的简单版本是这样的，首先我们的Mapper读入一个file，这个file中每一行代表一个存活的节点的坐标，然后Mapper做出9个Key-Value对，对这个存活节点的邻居cell，分发出一个1。而对于节点自身，也要分发出一个1。这里Reducer是对应每个cell的，每个reducer累加自己cell得到了多少个1，就知道自己的cell周围有多少存活cell，就能知道该cell下一轮是否可以存活，如果可以存活则分发回mapper的文件中，等待下次读取，如果不能则舍弃。

如果要进一步优化Map Reduce，那我们主要优化的地方则是mapper和reducer通信的开销，因为对于每个存活节点，mapper都要向9个reducer发一次信息。我们可以在mapper中用一个哈希表，当mapper读取文件的某一行时，先不向9个reducer发送信息，而是以这9个cell作为key，将1累加入哈希表中。这样等mapper读完文件后，再把哈希表中的cell和该cell对应的累加1次数，分发给相应cell的reducer，这样就可以减少一些通信开销。相当于是现在mapper内做了一次累加。这种优化在只有一个mapper是无效的，因为这就等于直接在mapper中统计完了，但是如果多个mapper同时执行时，相当于在每个mapper里先统计一会，再交给reducer一起统计每个mapper的统计结果。

```
1: class Mapper:
2: method Map ():
3: hash = ∅
4: for line ∈ stdin:
5:     cell, state = Parse (line)
6:     hash[cell] += state
7:     for neighbor in Neighborhood (cell):
8:         hash[neighbor] += 2*state
9: for cell in hash:
10:     strip-number = cell.row / strip-length
11: Emit (cell, strip-number, hash[cell])
```

```
1: class Reducer:
2: method Reduce ():
3: H = 0; last-cell = None
4: for line ∈ stdin:
5:     strip-number, current-cell, in-value = Parse (line);
6:     if current-cell ≠ last-cell :
7:         if last-cell ≠ None:
8:             Emit (last-cell, state=F(E(H))
9:         H = 0; last-cell = current-cell
10:     H += in_value
11: Emit (last-cell, state=F(E(xi))
```

5. 如果整个图都会变，有没有更快的方法？
   参见Hashlife，大意是用哈希记录一下会重复循环的pattern

Game of Life II

> In Conway's Game of Life, cells in a grid are used to simulate biological cells. Each cell is considered to be either alive or dead. At each step of the simulation each cell's current status and number of living neighbors is used to determine the status of the cell during the following step of the simulation.
>
> In this one-dimensional version, there are N cells numbered 0 through N-1. The number of cells does not change at any point in the simulation. Each cell i is adjacent to cells i-1 and i+1. Here, the indices are taken modulo N meaning cells 0 and N-1 are also adjacent to eachother. At each step of the simulation, cells with exactly one living neighbor change their status (alive cells become dead, dead cells become alive).
>
> For example, if we represent dead cells with a '0' and living cells with a '1', consider the state with 8 cells: 01100101 Cells 0 and 6 have two living neighbors. Cells 1, 2, 3, and 4 have one living neighbor. Cells 5 and 7 have no living neighbors. Thus, at the next step of the simulation, the state would be: 00011101

编解码法

复杂度

时间 O(N) 空间 O()

思路

一维数组需要考虑的情况更少，要注意的是这里头和尾是相连的，所以在判断其左右两边时要取模。

代码

```
public void solveOneD(int[] board){
    int n = board.length;
    int[] buffer = new int[n];
    // 根据每个点左右邻居更新该节点情况。
    for(int i = 0; i < n; i++){
        int lives = board[(i + n + 1) % n] + board[(i + n - 1) % n];
        if(lives == 1){
            buffer[i] = (board[i] + 1) % 2;
        } else {
            buffer[i] = board[i];
        }
    }
    for(int i = 0; i < n; i++){
        board[i] = buffer[i];
    }
}
```

In Place 一维解法

```
public void solveOneD(int rounds, int[] board){
    int n = board.length;
    for(int i = 0; i < n; i++){
        int lives = board[(i + n + 1) % n] % 2 + board[(i + n - 1) % n] % 2;
        if(lives == 1){
            board[i] = board[i] % 2 + 2;
        } else {
            board[i] = board[i];
        }
    }
    for(int i = 0; i < n; i++){
        board[i] = board[i] >= 2 ? (board[i] + 1) % 2 : board[i] % 2;
    }
}
```

表优化法

复杂度

时间 O(N) 空间 O()

思路

和上题的表优化一个意思，不过这里用到了循环数组，并且规则不太一样。

代码

```
public void solveOneDWithTable(int[] board){
    int n = board.length;
    int[] lookupTable = {0, 1, 0, 1, 1, 0, 1, 0};
    int[] buffer = new int[n];
    int env = board[n - 1] * 2 + board[0] * 1;
    for(int i = 0; i < n; i++){
        env = (env % 4) * 2 + board[(i + n + 1) % n] * 1;
        buffer[i] = (lookupTable[env] + board[i]) % 2;
        System.out.println(env);
    }
    for(int i = 0; i < n; i++){
        board[i] = buffer[i];
    }
}
```

2015年10月05日发布 更多

系列文章

[Leetcode] Shortest Word Distance 最短单词间距 1 收藏， 6.6k 浏览

[Leetcode] Walls and Gates 墙与门 4.5k 浏览

[Leetcode] Minimum Size Subarray Sum 最短子串和 1.9k 浏览

相关收藏夹 换一组

数据结构
6 个条目 | 0 人关注

js数据结构
5 个条目 | 3 人关注

web_d
11 个条目 | 1 人关注

分享扩散：