生命游戏，高级编程风格

马克西米利安1986

2015 年 10 月 27 日 警察

“生命游戏”是人类进化的数学表示，其中一些模式随着时间的推移而发展。

下载 GameOfLife.zip

介绍

“生命游戏”是人类进化的数学表示，其中一些模式随着时间的推移而发展。每个模式通过时间的演化改变二维正交宇宙的状态。看到它非常酷，并且具有简单的规则，它具有令人难以置信的复杂行为。

规则

生命宇宙非常简单。方形网格包含细胞，这些细胞可能是死的，也可能是活的。每个单元格的行为与它的八个邻居交互，根据以下规则：

1. 任何具有少于两个活邻居的活细胞都会死亡，就好像是由于人口不足造成的。
2. 任何有两个或三个活邻居的活细胞都会传给下一代。
3. 任何拥有三个以上活邻居的活细胞都会死亡，就像人口过多一样。
4. 任何只有三个活邻居的死细胞都会变成活细胞，就像通过繁殖一样。

它从网格上的一个模式开始——在时间 t0 生成——并同时在所有单元格上应用规则。这个动作导致了一个新的模式（在时间 t1 生成），其中二维宇宙的状态已经改变。

之后，我们再次将规则应用于所有单元格等。

就是这样。没有其他规则。

方法

有不同的模式来解决这个问题。

一个不熟练的开发人员可能会直接背离规则，没有任何分析，实现模式 MVC，而不利用可能的并发编程风格。

大多数并发应用程序都是围绕任务的执行来组织的：将应用程序的工作划分为任务可以简化程序组织，通过提供自然事务边界促进错误恢复，并通过提供并行工作的自然结构以异步模式运行来促进并发。

设计空间的步骤

首先，我们必须分析识别可利用并发的问题，然后是并发架构。我们可以通过识别：

- 可能的任务
    - 与算法的某个逻辑部分相对应的指令序列。在实践中，这是一项需要作为作业完成的工作（面向设计的定义）
- 数据分解及其依赖
    - 实体中的映射任务
        - 激活负责执行任务的实体
        - 封装任务执行的控制逻辑

数据分解模式

概念类

问题很容易分解为可以相对独立操作的单元；就像一个divide-et-impera原则，它可以自然地分解为一组独立的任务：

- 任务必须足够独立，以便管理依赖关系只占用程序总执行时间的一小部分
- 独立性有利于并发，因为如果有足够的处理资源，独立的任务可以并行执行
- 选择好的任务边界，加上合理的任务执行策略，可以帮助实现这些目标。

更确切地说：

- 首先，识别数据单元，然后与任务相关，以便可以识别单元
- 在这种情况下，任务分解遵循数据分解
- 执行以异步模式进行：不同的任务帮助计算结果，然后，它们通过一个简单的监视器以被动模式同步

关于同步的警告：并发编程风格的毒害

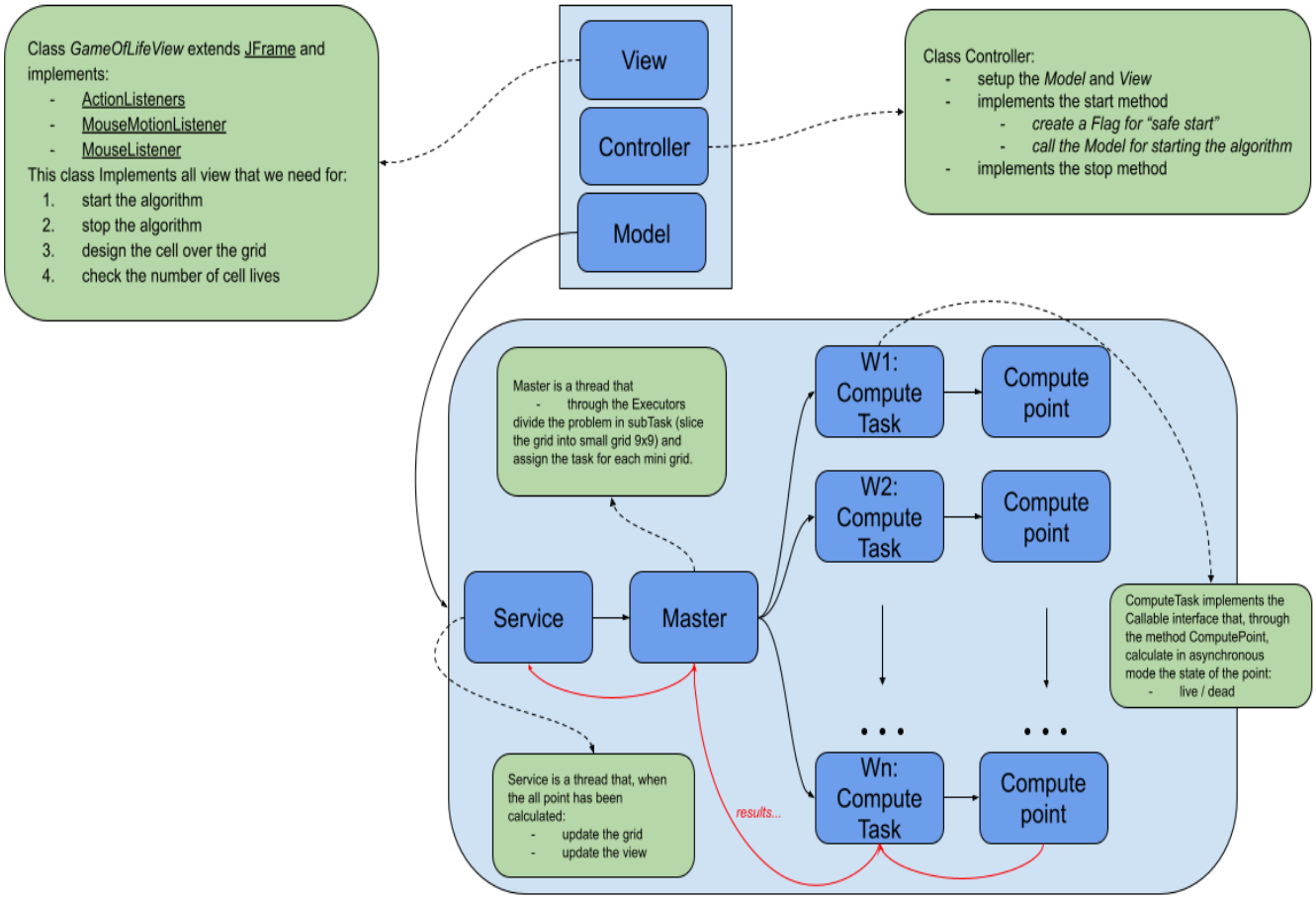
- 序列化/顺序化是表演的毒药
- 但对于正确性（安全属性）是必要的

我们不可能拥有生活中的一切，每个故事都有两张脸:-)

架构和模式

在这种情况下，我将使用 Master-Workers 模式。
该架构由一个主实体和一组动态的工作实体组成，通过适当的协调媒介进行交互，充当任务包

- 掌握
    - 将全局任务分解为子任务
    - 通过将子任务插入包中来分配子任务
    - 收集任务结果
- 工人
    - 从包里取出要做的任务，执行任务
    - 交流结果
        - 数据结构的监视器可用于聚合和同步结果



建筑学

它选择了一种MVC架构，其中：

- 看法：
  - 开始算法
  - 停止算法
  - 设计网格上的单元格
  - 检查细胞生命的数量
- 控制器：
  - 设置模型和视图
  - 实现 start 方法
  - 为"安全启动"创建一个标志
  - 调用模型以启动算法
  - 实现停止方法
- 模型：
  - 模型是所有项目的核心，它实现了所选择的策略。
  - 我们选择了Executor Framework来实现上面的策略：这个Executor基于生产者-消费者模式，其中提交任务的活动是生产者（生产要完成的工作单元），执行任务的线程是消费者（消耗这些工作单元）。

开始吧！

看法：

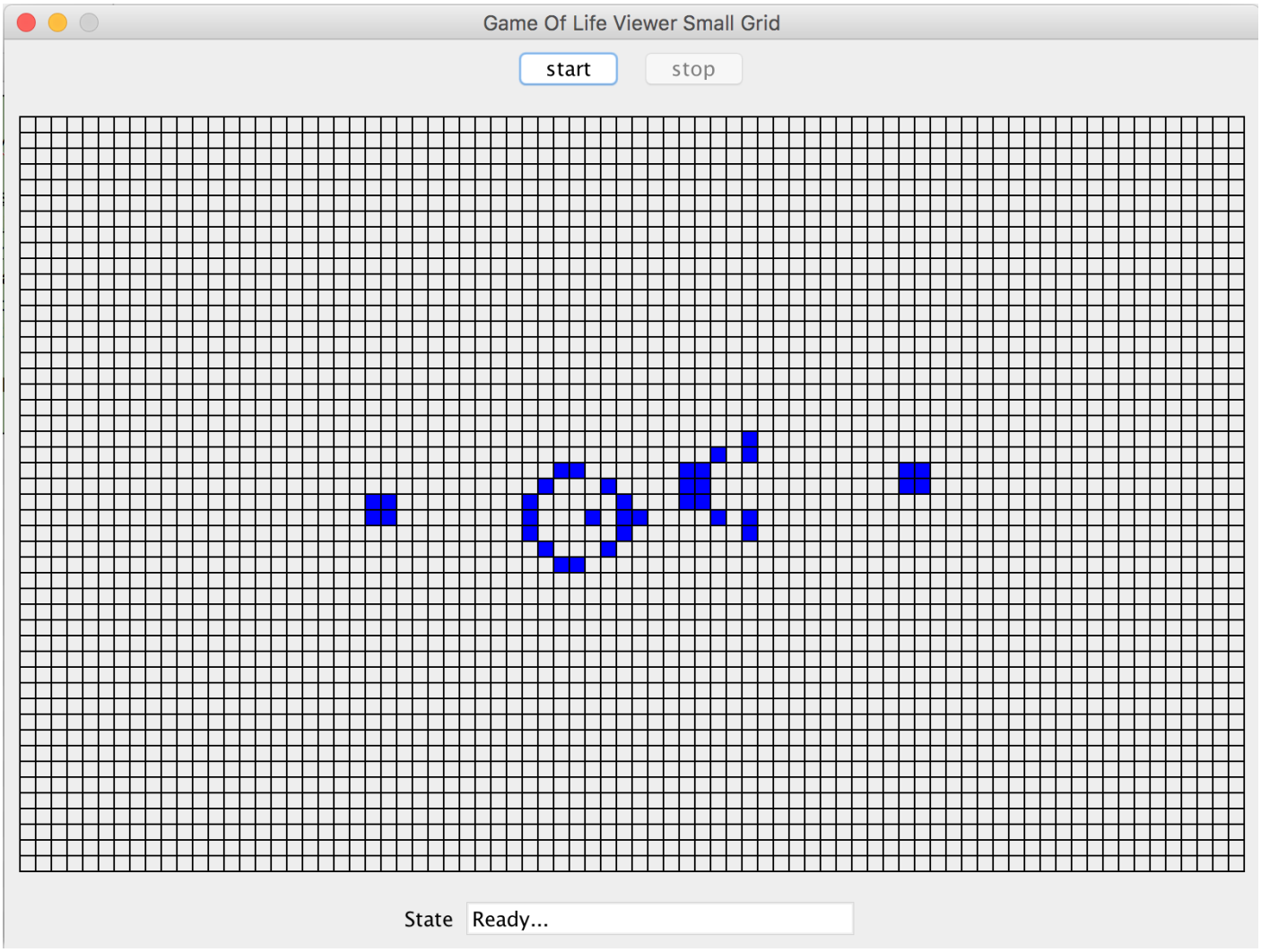
爪哇 缩小▲ 复制代码

```
public GameOfLifeView(Dimension windowsSize, Dimension matrixSize,int blockSize) {
    super("Game Of Life Viewer Small Grid");
    setSize(windowsSize);
```

```
    // Center the window in the middle of the screen
    setLocation(
                (Toolkit.getDefaultToolkit().getScreenSize().width - getWidth()) / 2,
                (Toolkit.getDefaultToolkit().getScreenSize().height - getHeight()) / 2
                );

    listeners = new ArrayList<inputlistener>();

    startButton = new JButton("start");
    stopButton = new JButton("stop");

    // When start the program, the Button "stop" is not enabled
    stopButton.setEnabled(false);

    JPanel controlPanel = new JPanel();

    controlPanel.add(startButton);
    controlPanel.add(stopButton);

    setPanel = new GameOfLifePanel(matrixSize, blockSize);
    setPanel.setSize(matrixSize);

    addMouseMotionListener(this);
    addMouseListener(this);

    JPanel infoPanel = new JPanel();
    state = new JTextField(20);
    state.setText("Ready...");
    state.setEditable(false);
    infoPanel.add(new JLabel("State"));
    infoPanel.add(state);
    JPanel cp = new JPanel();
    LayoutManager layout = new BorderLayout();
    cp.setLayout(layout);
    cp.add(BorderLayout.NORTH, controlPanel);

    // Add in the grid the Gosper Glider Gun like a test
    setPanel.CannoneAliantiGosper();

    cp.add(BorderLayout.CENTER, setPanel);
    cp.add(BorderLayout.SOUTH, infoPanel);
    setContentPane(cp);

    startButton.addActionListener(this);
    stopButton.addActionListener(this);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setResizable(false);
}
```

首先，我创建我插入的视图：
- 开始
按钮 - 停止按钮
- 网格（设置面板是实现网格的类）
- 检查单元标签 - 死或活 -

创建视图后，GUI 的结果应该是这样的：



控制器：

复制代码

```
public class Controller implements InputListener {

    private GameOfLifeView view;
    private GameOfLifeSet set;
    private Flag stopFlag;

    public Controller(GameOfLifeSet set, GameOfLifeView view){
        this.set = set;
        this.view = view;
    }

    @Override
    public void started(ArrayList< point > matrix){
        set.setupMatrix(matrix);
        stopFlag = new Flag();
        new QuadratureService(set,view,stopFlag, matrix).start();
    }

    @Override
    public void stopped() {
        stopFlag.set();
    }
}
```

类控制器非常简单：给定程序的结构，控制器仅限于链接模型和视图，为方法启动和停止创建监视器。

模型：

程序的模型是本文的核心，上面解释的架构在其中栩栩如生。
首先，我创建了一个类`QuadratureService`：

- 创建将所有单元格"存活"或"死亡"状态返回到网格的线程 Master。
- 接下来，它更新模型 (GameOfLifeSet) 和视图 (GameOfLifeView) 的网格

爪哇 缩小▲ 复制代码

```
/**
* Main thread to check the status of the algorithm Game of Life
*
* @param set
* @param view
* @param stopFlag
* @param matrix
*/
public QuadratureService(GameOfLifeSet set, GameOfLifeView view, Flag stopFlag, ArrayList<
point > matrix) {
    this.view = view;
    this.set = set;
    this.stopFlag = stopFlag;
    pointBoardT0 = new boolean[3][3];
}
@Override
public void run() {
    try {
        if (set.getMatrix().isEmpty()) {
            stopFlag.set();
            view.changeState("No point in Matrix!");
        }
        while (!stopFlag.isSet()) {
            Thread.sleep(50);
            //The numbers of core in your pc
            int poolSize = Runtime.getRuntime().availableProcessors() + 1;

            /*
             * I create master which calculates the state of the cells. the
             * result of all cells counted is saved in Variable result
             * (ArrayList < point >) that it is passed to both the Model that
             * the View to update the status of the structure Data and the
             * grid display screen
             */
            Master master = new Master(set, poolSize, stopFlag);
            ArrayList< point > result = master.compute();
            set.setupMatrix(result);
            view.setUpdated(result);

            if (stopFlag.isSet()) {
                view.changeState("Interrupted. Cell live: " + result.size());
            } else
                view.changeState(String.valueOf(result.size()));
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

Thread`Master`是一个 Executor，更准确地说是一个 `Executors.newFixedThreadPool(poolSize);`
But，我为什么要使用它?

- **新的固定线程池。**固定大小的线程池在提交任务时创建线程，直到最大池大小，然后尝试保持池大小不变（如果线程因意外异常而死亡，则添加新线程）。
- **新缓存线程池。**缓存线程池具有更大的灵活性，可以在池的当前大小超过处理需求时获取空闲线程，并在需求增加时添加新线程，但对池的大小没有限制。

为简单起见：我为每个存活的细胞创建了一个异步任务。从统计上看，存活的细胞数量非常高，所以我更喜欢使用方法 `newFixedThreadPool`。

爪哇 缩小▲ 复制代码

```
public Master(GameOfLifeSet set, int poolSize, Flag stopFlag) {
    this.executor = Executors.newFixedThreadPool(poolSize);
    // this.executor = Executors.newCachedThreadPool();
    this.set = set;
    this.stopFlag = stopFlag;
}

/**
 * The function returns the list of the living cells of the game
 *
 * @return
 * @throws InterruptedException
 */
public ArrayList< point > compute() throws InterruptedException {

    ArrayList< future < point > > pointT1 = new ArrayList< future < point > >();
    gameBoard = new boolean[set.getSizeX() + 1][set.getSizeY() + 1];
    for (Point current : set.getMatrix()) {
        if (current != null)
            gameBoard[current.x + 1][current.y + 1] = true;
    }
    for (int i = 1; i < gameBoard.length - 1; i++) {
        for (int j = 1; j < gameBoard[0].length - 1; j++) {
            try {
                pointBoardT0 = new boolean[3][3];
                // load points near instant T0
                pointBoardT0[0][0] = gameBoard[i - 1][j - 1];
                pointBoardT0[0][1] = gameBoard[i - 1][j];
                pointBoardT0[0][2] = gameBoard[i - 1][j + 1];
                pointBoardT0[1][0] = gameBoard[i][j - 1];
                // pointBoard[1][1] (me)
                pointBoardT0[1][2] = gameBoard[i][j + 1];
                pointBoardT0[2][0] = gameBoard[i + 1][j - 1];
                pointBoardT0[2][1] = gameBoard[i + 1][j];
                pointBoardT0[2][2] = gameBoard[i + 1][j + 1];

                /*
                 * If the point is evaluated "false" (white grid) then it is
                 * useless to create the res object type Future < point > as
                 * it will always be "false" and the grid does not change.
                 */
                int check = 0;
                for (boolean state[] : pointBoardT0) {
                    for (boolean b : state) {
                        if (b)
                            check++;
                    }
                }

                if (check != 0) {
                    /*
                     * I create the object of res futures; that is a
                     * "promise" of Point ("live" or. "Dead")
                     */
                    Future< point > res = executor.submit(new ComputeTask(
                            set, gameBoard[i][j], pointBoardT0, i, j,
                            stopFlag));
                    // The result I insert it in the list point T1 (at time T1)
                    // Like a monitor barrier
                    pointT1.add(res);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

我们深入下去！此时，程序为每个活动单元创建一个异步任务。这意味着并发性最大化。

爪哇 复制代码

```
@Override
public Point call() {
    Point pointT1 = null;
    try {
        pointT1 = result.computePoint(pointT0, pointBoard, i, j);
        if (stopFlag.isSet()) {
            // log("task interrupted");
        } else {
            // log("task completed");
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }

    return pointT1;
}
```

最后，computePoint计算细胞是活的还是死的，执行生命游戏的规则。

爪哇 缩小▲ 复制代码

```
public Point computePoint(boolean pointT0, boolean[][] pointBoard, int i, int j) {
    int surrounding = 0;
    if (pointBoard[0][0]) surrounding++;
    if (pointBoard[0][1]) surrounding++;
    if (pointBoard[0][2]) surrounding++;
    if (pointBoard[1][0]) surrounding++;
    // pointBoard[1][1]
    if (pointBoard[1][2]) surrounding++;
    if (pointBoard[2][0]) surrounding++;
    if (pointBoard[2][1]) surrounding++;
    if (pointBoard[2][2]) surrounding++;

    if (pointT0) {
        // A cell m [i, j] that in state s (t) is live and has
        // two or three neighbouring cells live,
        // in state s (t + 1) remains live ("survives")

        if ((surrounding == 2) || (surrounding == 3))
            return new Point(i - 1, j - 1);
    } else {
        // A cell m [i, j] that in state s (t) is dead and has
        // Three neighbouring cells live,
        // In state s (t + 1) become live
        if (surrounding == 3)
            return new Point(i - 1, j - 1);
    }
    /**
     * 1) a cell m [i, j] that in state s (t) is live and has zero or more a
     * neighbouring cell live (and other dead), in state s (t + 1) becomes
     * Dead ("die of loneliness")
     *
     * 2) a cell m [i, j] that in state s (t) is live and has four or more
     * neighbouring cells live in the state s (t + 1) becomes dead ("dies
     * over-population ")
     */
    return null;
}
```

关掉

一旦任务并行化，并发性最大化，算法应该如何关闭?
我们已经看到了如何创建一个 Executor 但没有看到如何关闭一个。由于 Executor 异步处理任务，因此在任何给定时间之前提交的任务的状态都不会立即明显。

一些可能已经完成，一些可能正在运行，而另一些可能正在排队等待执行。
为了关闭应用程序，有一个范围：

- 优雅关机：
    - 完成你已经开始的工作，但不要接受任何新工作
- 突然关机
    - 关闭机房和其间各点的电源。

对于这个应用程序，我使用了：

- 监视器 `Flag`：与用户一起检查按钮开始/停止是否出现在视图中
- 优雅关闭：启动有序关闭，其中执行先前提交的任务，不会接受新任务。

爪哇 复制代码

```
public class Flag {

    private boolean isSet;

    public Flag(){
        isSet = false;
    }

    public synchronized void set(){
        isSet = true;
    }

    public synchronized boolean isSet(){
        return isSet;
    }

    public synchronized void reset(){
        isSet = false;
    }
}
```

好奇心

目前就读于普林斯顿大学的英国数学家约翰康威在 1960 年代后期发明了生命游戏。他选择了产生最不可预测行为的规则。
如果您想了解更多关于生命游戏的历史背景，请访问 John Conway最近 关于生命游戏起源的采访。

结论

围绕任务执行构建应用程序可以简化开发并促进并发。Executor 框架允许您将任务提交与执行策略解耦，并支持多种执行策略。
为了最大限度地利用将应用程序分解为任务的好处，您必须确定合理的任务边界，您可以在分析问题的各个阶段完成这些工作。
在某些应用程序中，明显的任务边界运行良好，而在其他应用程序中，可能需要进行一些分析以发现更细粒度的可利用并行性，例如本案例研究。

参考书目

- http://www.theguardian.com/science/alexs-adventures-in-numberland/2014/dec/15/the-game-of-life-a-beginners-guide
- https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
- Java 并发实践 - Goetz 等人，Addison Wesley，2006 年

执照

分享

关于作者

马克西米利安1986
博洛尼亚学生大学
意大利
手表该会员
博洛尼亚大学学生

评论和讨论

添加评论或问题 | 电子邮件提醒 | Search Comments

第一 页上一页 下一页

想法
PIEBALDconsult 31-Oct-15 10:21

缺少下载文件
BillW33 29-Oct-15 5:04

Re: 缺少下载文件
BillW33 29-Oct-15 22:15

刷新 1

一般 新闻 建议 问题 错误 答案 笑话 赞美 咆哮 管理员

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。