中国科学 : 信息科学

SCIENTIA SINICA Informationis

课程报告

Developing a Java Game from Scratch

陈奕诺

学号：191220013 院系：计算机科学与技术系

E-mail: 1053650507@qq.com

摘要 本文主要是对于基于JAVA设计的吃豆人游戏的简单介绍，包括对于吃豆人游戏基本的游戏逻辑、设计理念，部分技术方法的介绍以及在游戏的设计过程的一些简单介绍。同时作为高级JAVA程序设计这一课程的实验部分，在最后提出了一些对课程的小感想。

关键词 JAVA，网络，游戏，面向对象，课程感想

1 游戏概况

本次游戏设计主要是基于之前的课程实验，设计一个ruguelike的对战游戏，并实现多人网络游戏要求。基于上述要求，结合之前的实验，我在原有的葫芦娃迷宫游戏的基础上，借用原有的对象设计和部分代码，设计出了一个简单的吃豆人（PACMAN）游戏，游戏的大部分设置与原有的经典游戏基本保持一致，通关要求仍然为在不被幽灵杀死的情况下吃完所有的豆子。其中开始界面如图1所示，基础游戏界面如图2 所示。采用面向对象的设计，更加符合实际的游戏逻辑，使得游戏逻辑设计更加清晰，同时降低代码的耦合度，方便编程和进行代码管理。

2 具体游戏介绍

本次游戏主要由四部分组成，分别是开始一个新游戏，继续旧游戏，回放游戏之前的游戏过程和网络对战。接下来我们进行逐个说明。

2.1 开始新游戏

2.1.1 基本介绍

由开始菜单，键盘输入“A”进入。这部分的游戏逻辑其实很简单，主要就是一个单机的吃豆人游戏，玩家通过”w””s””a””d”四个按键来控制吃豆人移动，在不被幽灵杀死的情况下吃完所有的豆子则成功，跳转到成功界面，游戏结束，否则跳转到失败界面，游戏失败。值得注意的是在本游戏中，有两种道具，分别是魔法豆（樱桃）和爱心，其中，吃到爱心，可以加一条生命，吃到魔法豆会进入为期10s的魔法状态。在魔法状态下，吃豆人与幽灵相遇，会杀死幽灵，而非魔法转状态下，则会被杀死。进入魔法状态的效果如图3所示。

图 1　菜单

Figure 1　Menu

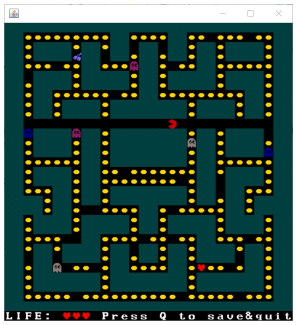

图 2　基础游戏界面

Figure 2　Basic Game

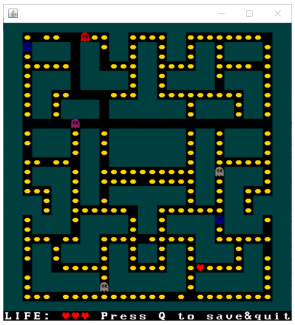

图 3　魔法模式

Figure 3　Magic Mode

2.1.2　基本实现原理

为了简化游戏实现，游戏的基础逻辑是建立在JAVA的实验JW04上进行的，尤其是对于游戏界面的设计，沿用了之前提供的ASCIIPanel和JFrame作为图形化工具，同样的我们也可以沿用之前的WorldScreen的一系列对于键盘监听等的操作。为了沿用JW04的一些设计，我更改了world的一些设计，主要是在原来的单个容器的基础上，添加了一个容器tiles_props，用来存放道具（包括魔法豆、普通豆子、生命恢复豆），从而实现代码的复用。为了将游戏界面向真实的PACMAN贴近，我通过网络查询到了相关的图片，通过将图片改成位图，调整大小，结合PHOTO SHOP我们可以得到属于自己的cp437的位图，从而进行游戏设计。

在游戏设计上，主要是将WorldScreen作为游戏的一个管理者，所有的界面切换和游戏初始化的设置都由其完成，并且能够调动游戏所有的对象，同时WorldScreen 还负责将外部的信息（Main键盘监听传入的KEYEVENT）经过一个简单的处理后传递给具体的实现对象。

如果说WorldScreen是一个游戏的管理者和信息的中转站，那么Map则是游戏的操控者，Map中存储了所有的地图信息和道具信息，地图的初始化和设置等操作由他来实现。其中我们的地图是从一个文本文件中读取生成的，而普通豆子的生成则是随机的。Map并不直接存储其上生物的信息，但是可以通过WorldScreen中传入的Creature的ArrayList来找到对应的生物，并调用相应的方法，来实现生物的操作。值得关注的是所有生物的移动都是通过Map来实现的。

同时为了实现游戏屏幕的及时刷新，我们创建了Refresher类（implements Runnable），在Main中即调用Refresher,并使得Refresher作为一个单独的线程来以一定的评率对屏幕进行刷新。从而避免了每次操作完成后要手动刷新，一方面可能带来刷新不及时和遗漏的问题，另一方面也无端增加了很多的工作量。

本游戏的设计其实概括起来主要就是基于地块的游戏实现，将所有的操作都传递给Map和WorldScreen,在由他们传递给对应的类，这样的设计可以间接地实现生物之间的相互作用，而解决了生物之间信息获取困难的问题。这种基于地图的游戏设计，其实参考了之前高级程序设计课程中，对于植物大战僵尸游戏的地块攻击法的设计，由于有了之前的编程经验，再次采用这样的设计也更加清晰明了。

2.1.3　并发控制实现

本次实验设计，要求把所有的生物都设计成单独的线程，我在初步的设计中是添加了6个幽灵，1一个玩家，也就是说，我们至少同时有7个独立的线程在进行游戏，那么解决线程的冲突问题就显得十分重要。我们在设计幽灵和玩家的时候，为了方便起见，所有的类都是基于Runnable接口（可以继承其

他类实现对Runnable实现类的增强，避免了Thread类由于继承Thread类而无法继承其他类的问题）实现的。为了简化游戏设计，我们对于Player的run设计，仅仅是对于魔法状态的一个计时器的计时设计以及检查。而对于幽灵的run的设计，也仅仅只是保持一定频率的随机移动。而他们的移动信息则统一传递给map，由map来实现对应的移动，也就是moveAtcion函数。我们知道由于生物不能同时占用同一个位置，所以我们必须要将移动判断和移动动作整合起来，统一时间只能有一个对象进行移动的判断和移动的进行。总结起来就是一次性，只能调用一次moveAction函数，由此我们能够想到synchronized（Java语言的关键字，当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码）。我们直接用synchronized来修饰moveAtcion函数（方法声明时使用,放在范围操作符(public等)之后,返回类型声明(void等)之前.这时,线程获得的是成员锁,即一次只能有一个线程进入该方法,其他线程要想在此时调用该方法,只能排队等候,当前线程(就是在synchronized方法内部的线程)执行完该方法后,别的线程才能进入）。从而基本解决了线程冲突的问题。

我们可以顺便看一下Map的moveAction的部分实现：

```
else if (map[nx][ny] == 2 && map[x][y] == 3) { // calabash to monster
    Creature m = null;
    Creature c = null;
    for (int i = 0; i < creatures.size(); i++) {
        int ax = creatures.get(i).getX();
        int ay = creatures.get(i).getY();
        if (ax == nx && ay == ny)
            m = creatures.get(i);
        else if (ax == x && ay == y)
            c = creatures.get(i);
    }
    if (c != null && m != null) {
        if (c.attack(m)) {
            map[nx][ny] = 1;
            m.beenkill();
            creatures.remove(m); // the monster been killed
            world.put(new Floor(world), nx, ny);
            world.swap(x, y, nx, ny);
            swap(x, y, nx, ny);
            if (prop_map[nx][ny] == 4 || prop_map[nx][ny] == 5 || prop_map[nx][ny] == 6) {
                if (prop_map[nx][ny] == 5)
                    has_magic++;
                c.eatBeans(prop_map[nx][ny]);
                prop_map[nx][ny] = 1;
                world.put_props(new Floor(world), nx, ny);
            }
        } else { // the calabash been killed
            c.beenkill();
            if(!c.checkifdead()){
```

```
                world.put(c, 17, 10);
                map[17][10] = 3;
                world.put(new Floor(world), x, y);
                map[x][y] = 1;
            }
            else{
                world.put(new Floor(world), x, y);
                map[x][y] = 1;
                creatures.remove(c); //the calabash been killed
            }
        }
    }
```

由这部分实现，我们可以基本了解Map移动的实现逻辑，为了方便判断操作，我们将地图中上对应位置的东西种类用一个二维数组存储了，根据位置的坐标信息，获取当前位置和要移动的位置中的东西种类，从而加快对于方法的选择判断，这部分代码展示的是玩家移动到下一个位置，而下一个位置中是幽灵的情况，首先依据位置信息，遍历生物的数组找到对应的幽灵和玩家对象，玩家对幽灵发起攻击操作，检查玩家状态，若为魔法状态，则对幽灵造成伤害，杀死了幽灵，幽灵地块变为地板，玩家移动，若为非魔法状态，则被幽灵反杀，玩家地块变为地板，检查玩家是否死亡，若未死亡则在固定地点复活。

现在我们了解了Map的基本实现，对于游戏的基础功能有了一个基本的认识。

2.2 继续旧游戏

依据JW06的要求，我们要实现一个游戏的存储功能，也就是在游戏退出以后可以继续上一次的游戏。目前该功能只支持单机游戏。为了实现这一功能，我们必定会使用一个文件来进行存储，但是具体存储的内容要进一步的设计，要想复原游戏，必须要存的信息，主要是地图信息和生物信息，我们就直接写入文件进行存储。首先将地图的两种信息（生物位置信息和道具信息）存入文件，实际是30x30的两个数组。然后将生物的信息，依据一定的顺序进行存储。实现代码如下：这一函数在玩家按下“Q”键退出时被调用。

```
public void save_game() throws IOException{
        //saving log
        log=map.get_log();
        BufferedWriter writer;
        writer = new BufferedWriter(new FileWriter("mylog.txt",true));
        writer.write(log);
        writer.flush();
        writer.close();

        //saving map
        saved_game="";
```

```
        int[][] now_map = map.get_map();
        for(int i=0; i<30;i++){
            for(int j=0; j<30;j++){
                saved_game += now_map[i][j] + " ";
            }
            saved_game += "\n";
        }
        //saving map prop
        int[][] now_prop_map = map.get_prop_map();
        for(int i=0; i<30;i++){
            for(int j=0; j<30;j++){
                saved_game += now_prop_map[i][j] + " ";
            }
            saved_game += "\n";
        }
        //saving monster
        if(!monster1.checkifdead()) saved_game += monster1.saving_state(); else saved_game +=
            "d\n";
        if(!monster2.checkifdead()) saved_game += monster2.saving_state(); else saved_game +=
            "d\n";
        if(!monster3.checkifdead()) saved_game += monster3.saving_state(); else saved_game +=
            "d\n";
        if(!monster4.checkifdead()) saved_game += monster4.saving_state(); else saved_game +=
            "d\n";
        if(!monster5.checkifdead()) saved_game += monster5.saving_state(); else saved_game +=
            "d\n";
        if(!monster6.checkifdead()) saved_game += monster6.saving_state(); else saved_game +=
            "d\n";
        //saving player
        if(!player.checkifdead()) saved_game += player.saving_state(); else saved_game += "d\n";

        BufferedWriter writer2;
        writer2 = new BufferedWriter(new FileWriter("saved_game.txt"));
        writer2.write(saved_game);
        writer2.flush();
        writer2.close();
    }
```

在保存了信息之后，继续开始新的游戏的时候，只需要从文件中读取相应的信息，进行对象的初始化即可实现状态的复原，复原后再开启生物对应的线程即可继续游戏，具体的过程不再重复。

2.3 游戏回放

JW06还要求了游戏回放的功能，我这里设计的游戏回放，并不是严格意义上的游戏回放，更多的是一个游戏的复盘过程，为了尽可能少的存储信息，我们这里的回放是基于日志进行的回放，而日志只记录了生物的基本操作，包括移动和魔法状态的变化，在游戏开始的时候，我们会清空日志，在游戏退出保存的时候，我们可以看到上面的代码中有存储log的部分，也就是将log写入到对应的txt文件中，实现存储操作。同时为了修复游戏状态，我们在每次新游戏开始的时候，都对于生成的道具地图进行存储，写入到文件中。有了道具信息文件和log文件，我们只需要依据信息文件进行游戏初始状态的还原，再由一个Replayer类对象按照一定的间隔执行日志中的操作即可。之所以说不是严格的回放，是由于这样的复盘方式，在时间上与原来的游戏过程是不一致的，他只能保证操作顺序的一致。由于操作顺序的严格一致性要求，我们将log的添加，安排在了Map的moveAction函数当中。

```
public synchronized void moveAction(int x, int y, int nx, int ny) {
  if (magic_log == 0 && has_magic == 1) {
      if (check_magic()) {
          log_map += "magic" + "\n";
          magic_log++;
          if(server != null){
              String s = "magic" + "\n";
              server.send_info(s);
          }
      }
  }
  // change dirction icon of calabashbros
  log_map += "move" + " ";
  log_map += x + " " + y + " " + nx + " " + ny + "\n";
```

2.4 网络联机

于是我们到了JW07的要求，实现一个简单的网络对战游戏。我目前只实现了一台服务器，两个客户端的内容，也就是有三个玩家。这三个玩家是一个队伍，三个玩家全部死亡，则游戏失败，三个玩家吃完了所有的豆子，则游戏成功。基本的游戏逻辑与之前保持一致，不过三个玩家组队时，只有一条生命，无法进行复活。详细的游戏界面，我们可以看图4和图5。图四为等待状态，必须有一台服务器和两个客户端才能开始游戏，默认的网络连接ip是本机，也就是127.0.0.1，后续联网可能需要进一步的更改。

2.4.1 网络通信的实现

由于传输的要求不高，仅仅只有两个客户端，所以我们没有使用reactor模式，而是直接使用了SOCKET套接字进行通信，对于建立的两个端口进行持续的监听操作，接受和发送信息。首先建立客户端和服务端的连接，再依据连接的结果，对端口进行监听，依据得到的信息进行具体的操作。为了实现这一效果，我们创建了Client, Client_send, Client_receive, Server_receive, Server_send 总共5个类对象来实现相应的操作，具体的网络信息仍然由WorldScreen进行处理。

图 4　等待连接

Figure 4　Connecting...

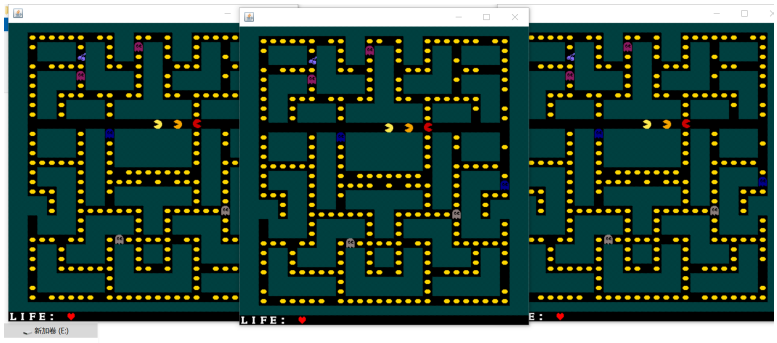

图 5　联机游戏

Figure 5　Online Game

2.4.2　实现三方信息同步

在实现三方信息同步上，我依据上面的回放功能做了调整，实际运行的只有Server，服务器负责运行所有的生物线程，进行正常的游戏，而用户端只负责执行由服务器发出的指令，同时将受到的按键操作发送给服务器进行处理。同样的为了同步，服务器一开始分发的是地图生成时的道具信息，客户端收到后就构建地图，而后等待服务器的下一步指令，指令格式与之前的log格式一致，可以直接分析执行，而当客户端监听到键盘操作后，将键盘操作信息与客户端在连接时的id一起发送给服务端，服务端按照指令进行操作，而服务端的每一步操作都会发给客户端，从而实现同步。这样做的好处是可以避免服务端与客户端不一致出现冲突，但是由于客户端要完全等待服务端分析后再进行操作，相当于等待了两个传输时间和一个处理时间，所以客户端的操作时延会比较明显。

3　工程管理

为了实现代码测试、打包等的一体化管理，我们统一采用了Maven管理工具进行代码的管理，并通过junit测试来保证代码的局部正确性。依据课程要求，进行分段设计，可以明确每一阶段的目标，便于进行代码的分解，避免不必要的麻烦。

4 课程感想

通过本学期的JAVA课程学习，我对于面向对象编程有了更加深刻的认识。通过实验的训练，了解了java的一些基础工具包，同时对于MAVEN的管理和Junit测试，JAVA的网络通信有了一定的尝试，建立了一些认识。同时本学期的实验完成上也有不足，主要是实验设计中的对象行为与现实世界没能实现很好的对应，且代码不太符合面向对象设计的原则，方法设计有很多的副作用，而少有对象的返回，这给Junit单元测试带来了巨大的困难，后续应该进一步进行代码的重构和改进。

本课程的课程安排较为合理，尤其是实验的设计，在保证一定代码训练量的同时，帮助我们更好地建立对java 的整体认识，但是实验中间的间隔有些不确定，也许可以进行更加合理的时间安排。