

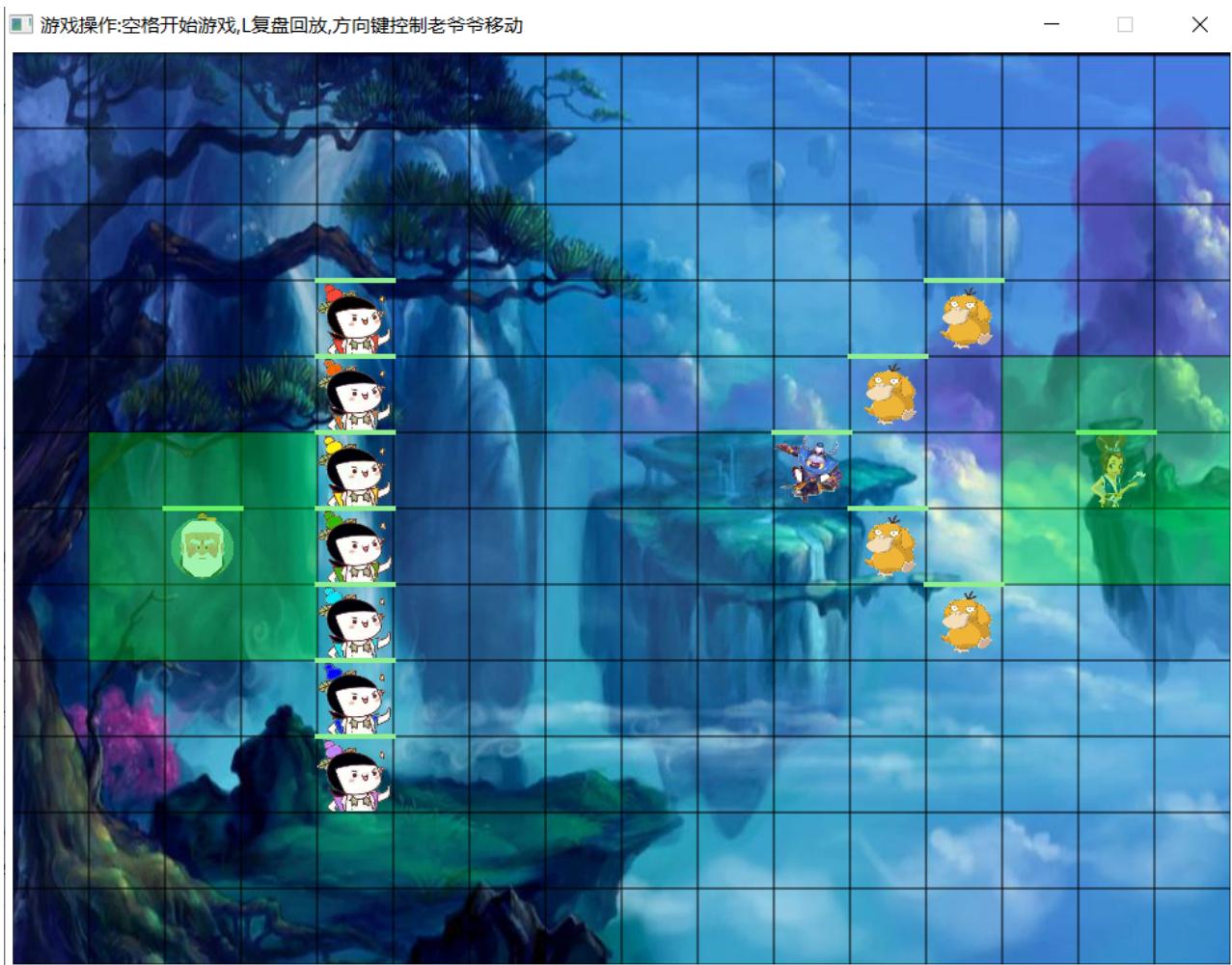
Java 葫芦娃大作业

171860525 计算机科学与技术系 陈善梁 邮箱: 171860525@smail.nju.edu.cn

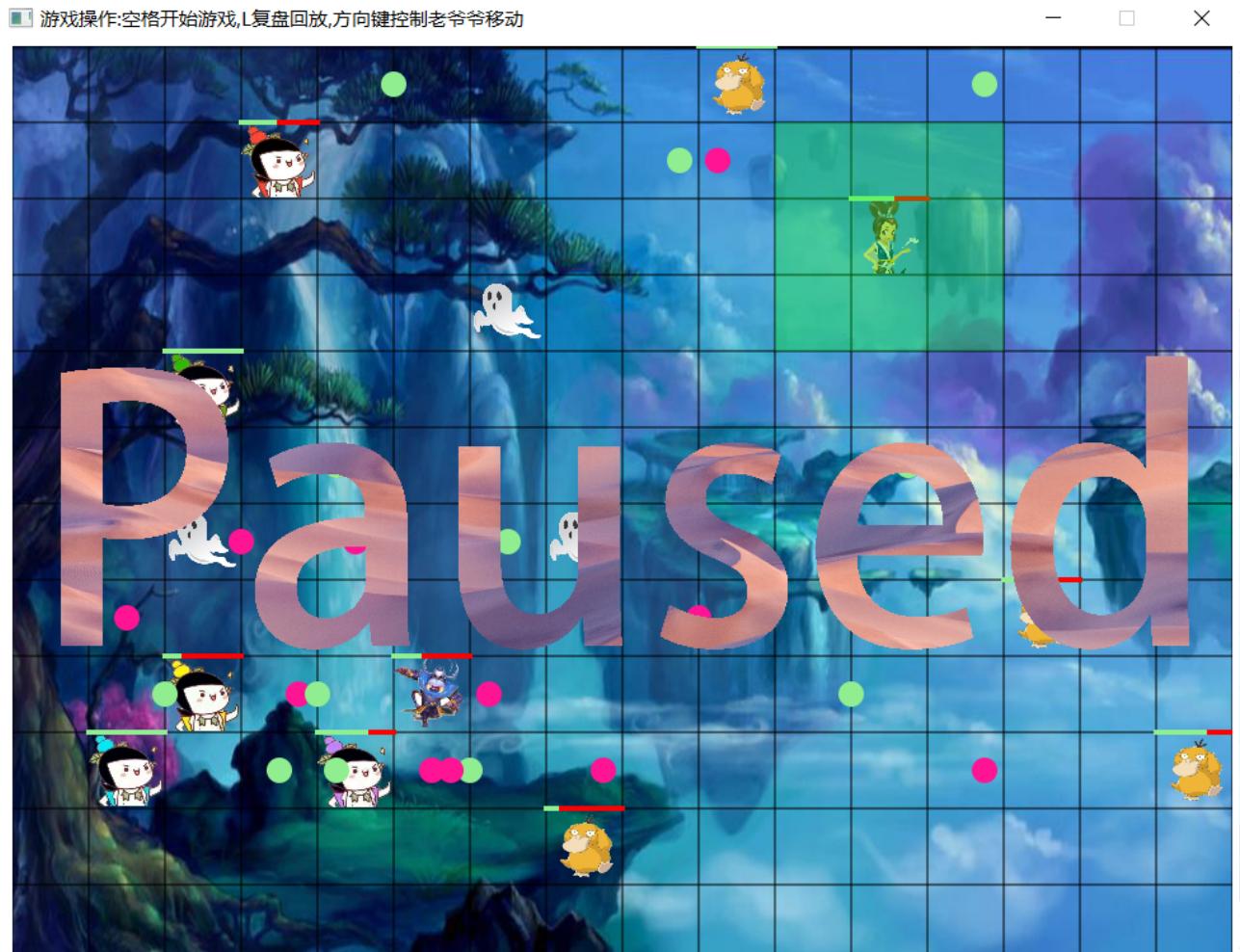
游戏操作说明

大作业的要求是“当某个生物体于敌方相遇（两者间的X轴距离和Y轴距离小于某个常量）时，选取一个概率决定双方生死”，当时我觉得这样的攻击方式太过粗暴，不能很好地体现真实的打斗伤害场景，而且可玩性较低。于是，在征得老师同意后，我改变了攻击的形式：发射不同类型的子弹。其他部分仍然按照大作业的要求完成。以下是游戏操作说明：

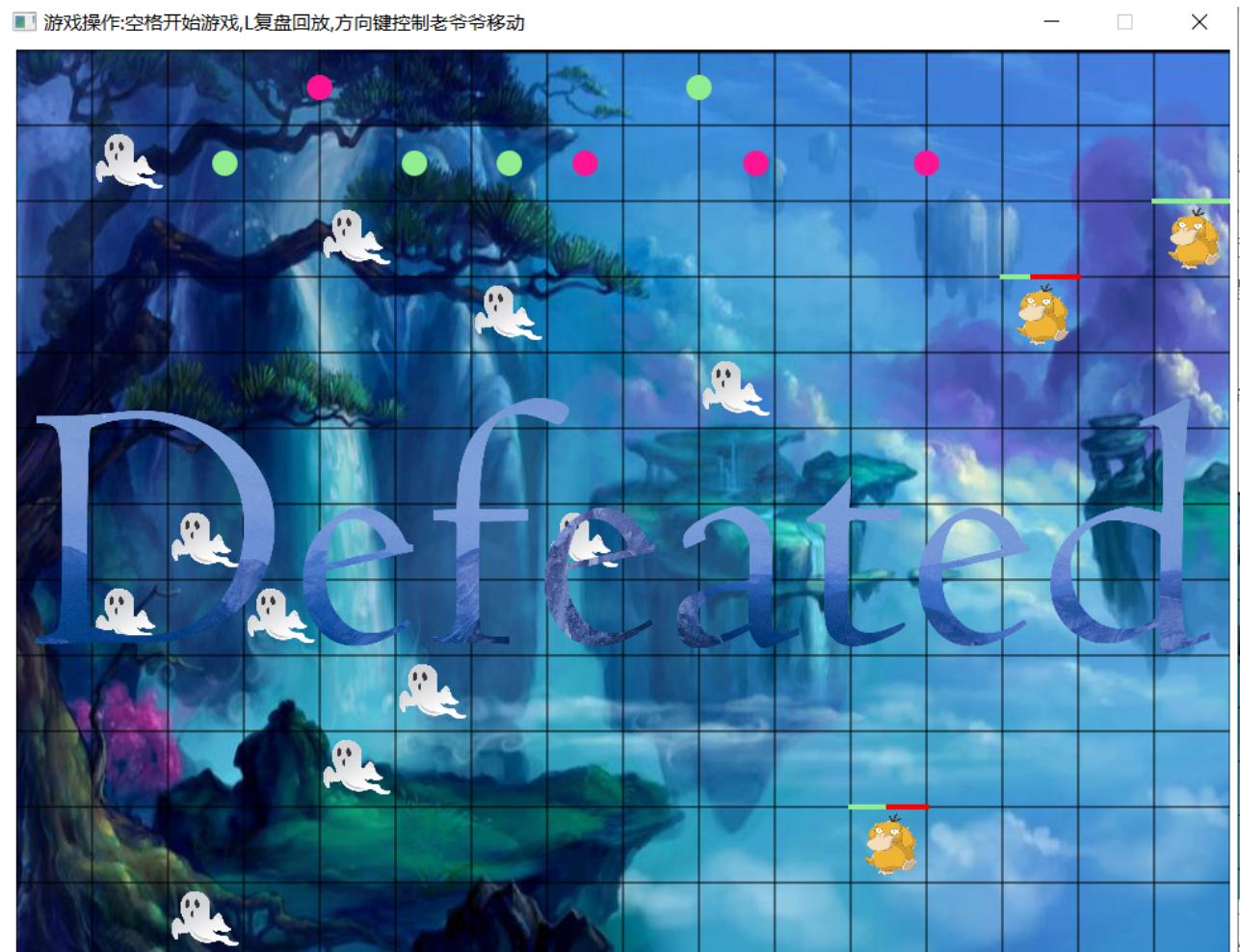
1. 打开游戏后，进入初始界面，葫芦娃阵营在左侧按照长蛇阵排好，妖精阵营在右侧按照锋矢阵法排列：



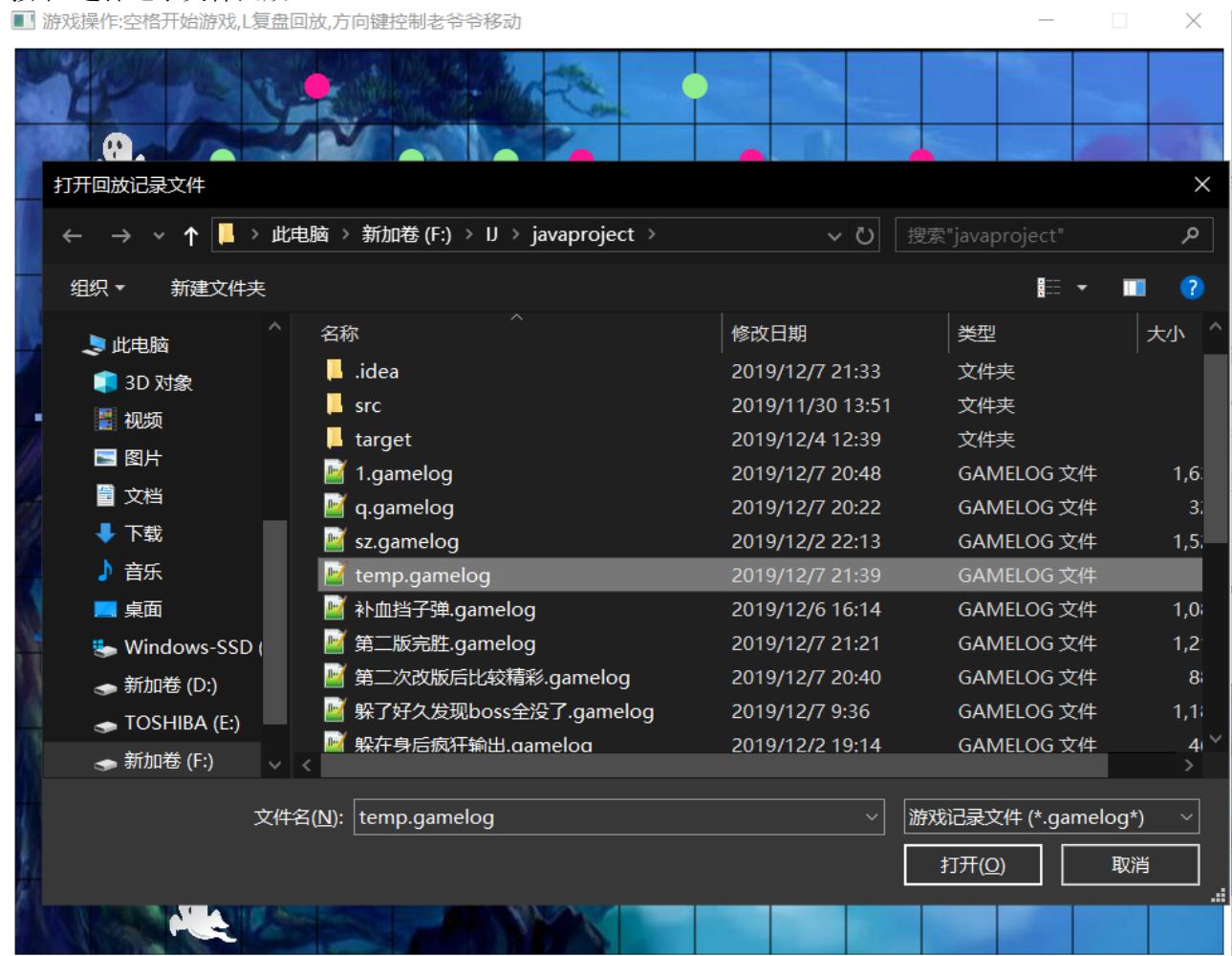
2. 按下空格键选择距离文件保存的目录及文件名,开始游戏。在游戏过程中可以用方向键控制老爷爷移动，用WSAD控制老爷爷向上下左右四个方向发射子弹:[]开始(src/main/resources/md/start.png) 开始游戏后按下空格键则暂停游戏:



游戏结束:



按下L选择记录文件回放:



项目框架

五个package:battle,bullet,creature,formation和record，它们的职责如下:

package	职责
battle	1. (BattleController.java) 负责UI组件(BorderPane、Canvas等)的初始化、战场元素(葫芦娃、妖精、子弹列表、子弹管理器、地图)的初始化、与记录有关的输入输出流(writer、reader)的声明、用于UI界面刷新的Timeline的声明、线程池的声明
	2. (BattleController.java) 监听键盘事件:空格键开始/暂停/继续游戏, F键改变阵型、L选择记录文件回放以及方向键控制移动等
	3. (BattleController.java) 实现对于键盘事件的相应函数, 如startGame()、gameOver()、pauseGame()、continueGame()等。
	4. (Map.java) 定义重要的战场地图Map类, 负责控制访问和修改战场某一位置的生物、显示战斗过程中的每一帧画面、显示回放过程中的每一帧画面。该类由所有生物对象共享。
	5. (Config.java) 配置战场规格、不同生物特性, , 如战场大小、行列数、生物攻击力等
	6. (BattleState.java) 定义全局的BattleState类, 表示当前状态:未开始、开始、暂停等。所有战场元素共享该对象。
creature	定义所有的生物, 基类Creature、葫芦娃类Huluwa、老爷爷GrandPa和妖精Evil等
bullet	定义所有的子弹类及其工厂类。
formation	定义阵法Formation类, 实现所有改变妖精阵型的方法

package 职责

record 定义游戏记录的三个类:BulletRecord(记录当前某一个子弹及其位置、伤害等信息)、CreatureRecord(记录当前某一个生物及其位置、状态信息)、Record(记录当前帧所有的子弹和生物)

面向对象的思想

1. 继承

继承最直接地体现在生物类上

所有生物都派生自基类Creature，而这个Creature定义了

1. 所有生物共有的属性：生命值HP、存活状态alive、生物形象Image、位置position、攻击力、防御力。此外，为了支持在地图中寻找敌人、发射子弹等操作，还有含有共享的Map对象以及用于生产子弹的bulletGenerator对象。
2. 所有生物共有的方法：攻击attack()、移动move()以及其他一些对属性进行修改、获取的方法。值得一提的是，Creature类实现了Runnable接口，重写了run()方法，而在run方法中描述了战场生物的普遍行为：sleep -> attack -> move。代码体现如下：

```
while (alive && Thread.interrupted() == false) { //死亡或者pool调用了shutDownNow则本线程退出
    //如果暂停的话，BattleField只要把所以生物的alive置为false，就能结束所有生物线程并且生物状态不变了
    try {
        Thread.sleep(1000 / moveRate);
        if (alive == false)
            break; //sleep后发现自己死了
        synchronized (map) { //上锁顺序 map -> creature(this)
            attack(); //attack()对map、enemy、bullet上锁
            move(); //move方法内部已经对map上锁了
        }
    } catch (InterruptedException e) {
        break; //在sleep的时候shutDownNow结束线程
    }
}
```

葫芦娃类、老爷爷类、妖精、蛇精、蝎子精都派生自Creature类，但是他们又根据特定的需要增加了一些属性，重写或者添加了一些方法。以GrandPa类为例：由于玩家需要对GrandPa进行控制(移动和子弹发射)，因此GrandPa需要记录玩家的键盘输入指令，包括方向键指令和WSAD子弹发射指令。

```
private LinkedList<Direction> moveDirections = new LinkedList<Direction>();
private LinkedList<Direction> bulletDirection = new LinkedList<>();
```

相应的，老爷爷的attack()、move()行为也是基于玩家指令的，因此必然异于生物基类的attack()和move(),因此需要重写，基于用户输入来确定如何攻击与移动。

此外，继承也体现在子弹

所有子弹都继承自子弹基类Bullet，而Bullet定义了

1. 子弹的共有属性:当前位置参数x与y、子弹颜色、子弹伤害、子弹的发射者和目标。
2. 子弹的普遍行为:move(), 即子弹移动。

而子弹具体课分为直行的子弹StraightBullet以及追踪弹TrackBullet，前者在生产之后只能按照既定的方向飞行。而后者会不断地追踪目标，直至击中敌人或者目标死亡。显然，两者的move()方法不同，前者只要把当前位置加上一个固定的偏移量。而后者需要根据当前目标的位置，调整飞行方向。

继承同样体现在子弹工厂上

所有子弹工厂都继承自BulletGenerator类，而它只含有一个抽象方法getBullet()，即制造出子弹。对应于不同的子弹，子弹工程也有StraightBulletGenerator以及TrackBulletGenerator，它们重写了getBullet()方法以产生不同的子弹。

2. 多态

多态和继承密不可分。多态的基础是：基类引用直线子类对象，该引用对基类方法的调用会动态绑定到子类的重写方法上。在本项目中，具体体现如下：所有生物都含有一个BulletGenerator引用，而因为BulletGenerator是抽象类，bulletGenerator显然要指向一个StraightBulletGenerator或者TrackBulletGenerator，而这是在生物对像构造器中决定的。对于葫芦娃和蝎子精，它们都没有重写Creature的attack()、run()函数，仅仅是因为它们的bulletGenerator引用了BulletGenerator的不同子类对象，它们就能产生出不同类型的子弹，从而实现了代码的精简和可维护性。如果不采用工厂模式的话，那么一个生物要发射子弹，就要显示地用new创建，这不够优雅，代码可读性也差一些。

3. 封装

封装主要体现在Map类对于战场的访问、修改上。战场是一个M*N的二维数组grounds。将这样一个较为底层的数据结构暴露给所有生物，让他们直接修改显然有些不够合适。而Map则实现了将战场进行封装，只提供一些接口供外界访问、修改战场，比如放置生物到指定位置的setCreatureAt()、在指定位置溢出生物的removeCreatureAt()以及判断一个位置是否有生物的removeCreatureAt()。

封装使得外部对象无法直接访问内部实现细节，体现了高内聚、低耦合的设计原则。同时，封装将一系列固定的操作用方法名易于读懂的方法表示，增加了代码的可读性。

用到的设计模式

1. 工厂模式。在工厂模式中，我们在创建对象时不会暴露创建的底层逻辑，而是提供一个统一的接口来创建新的对象。在本项目中，工厂模式的运用具体体现在生物对象的子弹工厂bulletGenerator上，不能的生物需要发射不同种类的子弹，如果在发射时才由生物对象指定创建哪一种子弹，那么后期如果想让该生物改变射击方式，发射其他种类的子弹，就需要比较复杂的代码逻辑，可拓展性低。运用工厂模式的话，只要让这个子弹工程引用另一种工厂对象的实例，而生物发射子弹的代码则不需要修改。
2. 观察者模式。本项目一个困惑我的问题是：当战斗结束时，主线程应该能够监测到战斗的结束，然后让仍然存活的生物线程、子弹管理线程终止。显然，战斗结束这个事件是可以由Map类的map对象在显示每一帧时通过计算两个阵营人数来确定的，但是map对象对主界面进行画面刷新是在另一个线程中执行的(javafx的Application Thread)，那么要如何通知主线程战斗已经结束呢？我原先打算的是在游戏开始时，主线程再创建一个线程，该线程用一个while循环侦听全局共享的battleState对象，一旦游戏状态为结束，就终止生物等线程。但是显然这种方式不太好，这种忙等待的方式浪费了cpu资源。结合线程并发的知识，我通过wait()和notifyAll()获取、释放battleState对象上的锁，实现了侦听线程。关键代码如下：

- 这是在gameStart()方法中创建战场侦听线程的代码:

```

new Thread(() -> {//用lambda表达式代替匿名内部类
    synchronized (battleState) {//观察者模式
        while (battleState.isBattleStarted()) {//等待战斗结束
            try {
                battleState.wait();//等待battleState的锁，而不是忙等待监听
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    gameOver();//现在这个侦听线程也要结束了
}).start();//这个侦听线程不经过pool控制

```

- 这是map对象的display()方法(显示每一帧图像)唤醒侦听线程的代码:

```

if (numEvilLeft == 0 || numJusticeLeft == 0) {
    battleState.setStarted(false);
    if (numEvilLeft == 0) {//设置战斗胜利者并且绘制
        battleState.setWinner(Camp.JUSTICE);
        gc.drawImage(justiceWinImage, 0, 0, CANVAS_WIDTH, CANVAS_HEIGHT);
    } else {
        battleState.setWinner(Camp.EVIL);
        gc.drawImage(evilWinImage, 0, 0, CANVAS_WIDTH, CANVAS_HEIGHT);
    }
    synchronized (battleState) {
        battleState.notifyAll();//唤醒侦听线程
    }
}

```

上述对于wait()和notifyAll()方法的使用真正让我感受到java对于并发的支持，加深了我对“锁”的认识，通过合理地使用对象和锁以及wait()和notifyAll()函数，能够编写出高效、优美的代码。

多线程的协同

本项目遇到的线程问题及解决方案:

1. 多个生物占据了同一个位置

解决方案: 在生物移动指向，用synchronized关键字对地图对象map上锁，防止其他生物线程修改战场上的生物位置信息，造成位置重叠。代码如下:

```

synchronized (map) {//上锁顺序 map -> creature(this)
    attack();//attack()对map、enemy、bullet上锁
    move();//move方法内部已经对map上锁了
}

```

2. 多个子弹同时杀死一个生物

解决方案：bulletController在发现子弹击中敌人时，锁住敌人，使它这一时刻不能移动也不能被其他子弹击中，之后对它进行打击伤害。代码如下：

```
Creature c = map.getCreatureAt(squareX, squareY);
if (c != null) {
    synchronized (c) { //不能移动,和生物run获得锁的顺序一样 map -> creature
        if ((c.getCamp() != bullet.getSender().getCamp()) && c.isAlive()) { //是敌方生物
            并且活着
                c.getHurt(bullet.getDamage()); //受到伤害
                it.remove(); //删除子弹
                continue;
            }
        }
    }
}
```

3.死锁

本项目中有战场map对象、所有生物对象、子弹列表bullets三个对象的锁被不同线程频繁地获取、释放，如果并发控制不当，极有可能发生死锁，导致游戏卡住。我采取的解决方式是：所有线程按照相同的顺序申请锁:map->creature和map->bullets(没有出现creature和bullets被synchronized嵌套的情况)，这样可以防止循环等待。

4. javafx本身不是线程安全的,不能在Application Thread之外的线程中修改ui组件

这个问题比较隐蔽，错误也不太容易发现。

我原本采取的方式是让Map实现Runnable和callable接口，分别执行display()实现绘制战场的每一帧图像和执行drawRecord()从记录文件中读取每一帧的记录再绘制到画布。这种方式在大多数情况下可以正常执行，但是少数情况下会出错，抛出异常，异常信息我也看不太懂。后来在网上查阅相关博客后发现，javafx并不是线程安全的，对UI组件的修改只能在javafx自己的application Thread线程中执行，如果在其他线程中执行，有可能会抛出异常。对于这个问题，有两种解决方式：

1. 让Map对象实现javafx.concurrent包中的task<V>接口，重写call()方法，在call()中绘制战场，修改canvas。但是这种方式不太容易同时实现map对象的正常对战显示和战斗回放两个功能。
2. 使用javafx.animation包中的TimeLine对象。TimeLine是一个时间轴，然后通过添加关键帧来形成动画，对于普通的战斗过程，UI界面刷新的时间轴是这样的：调用map.display()绘制当前场景 -> 等待一段时间 -> 调用map.display()绘制当前场景 ...
因此，TimeLine的初始化如下：

```
timeline = new Timeline(//用Timeline 来实现UI的刷新，是javafx安全的
    new KeyFrame(Duration.millis(0),
        event1 -> {
            if(!battleState.gamePaused())
                map.display(true); //每一帧都记录
            else{
                //显示暂停画面
            }
        }
    )
);
```

```

        map.displayPause();
    }
}),
new KeyFrame(Duration.millis(1000 / MAP_REFRESH_RATE))
);
timeline.setCycleCount(Timeline.INDEFINITE);
timeline.play();

```

上述代码在startGame()方法中被调用，表示战斗开始后要实时刷新画布，展现当前战局。

对于回放部分，同样也是用时间轴来实现，具体代码在map的startReview()方法中：

```

public void startReview() {
    //现在是单线程
    //独立创建一个TimeLine进行画面刷新
    reviewTimeline = new Timeline(
        new KeyFrame(Duration.millis(0),
            event1 -> {
                Record record = getNextRecord();
                if (record != null) {
                    drawRecord(record);
                }
            }),
        new KeyFrame(Duration.millis(1000 / MAP_REFRESH_RATE))
    );
    reviewTimeline.setCycleCount(Timeline.INDEFINITE);
    reviewTimeline.play();
}

```

那么时间轴什么时候停止刷新呢？对于前者，是战斗结束时，因此，当战场侦听线程被map.diaplay()唤醒后，就将timeLine终止：

```

public void gameOver() {
    //这个函数在侦听线程中被调用
    //失败一方的所有生物线段都因为alive = false 导致线程退出
    //胜利一方的所有生物线程、map刷新线程、bulletManager线程都可能还在运行
    //因此需要shutDownNow向所有线程发送interrupt()让他们退出
    pool.shutdownNow();
    timeline.stop(); //停止显示
    System.out.println("TimeLine stop");
}

```

对于回放，回放的timeLine终止是在读取记录文件的过程中，发现读到末尾了，就将回放的timeLine终止：

```

private Record getNextRecord() {
    Record record = null;
    try {
        record = (Record) reader.readObject();
    } catch (EOFException e) {

```

```

reviewTimeline.stop(); //结束timeline动画
battleState.setReviewing(false); //回放结束
return null;
} catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
return record;
}

```

序列化的使用

序列化的使用体现在游戏记录和复盘上。关于游戏的记录，我采取的方案是记录每一帧，要如何确定每一帧的图像呢？只要记录两个部分：生物和子弹。我原先的想法是把Creature和Bullet都实现Serializable接口，这样就可以把生物和子弹对象输出到文件了。但是，会存在一些多余的信息，比如生物对象中的攻击力和防御力等，这些信息在复盘时是用不到的，不需要存储，再比如生物图像image，所有普通妖精都是一个样子的，如果把image也一一存储，也是对空间的浪费。一个解决方案是把它们设置为transient的，这样就不会被存储了，但是我觉得还是把生物和生物的记录分开为好，因为它们逻辑上是两种不同的东西。子弹也是一个道理。于是我创建了CreatureRecord和BulletRecord类，分别表示一个生物、子弹的状态信息。而一帧信息的存储，需要记录当前所有的生物和子弹，因此还需要一个总的Record类，用表的形式存储所有的生物和子弹。如下：

```

public class Record implements Serializable {
    public ArrayList<CreatureRecord> creatureRecords;
    public ArrayList<BulletRecord> bulletRecords;
    public Record() {
        creatureRecords = new ArrayList<>();
        bulletRecords = new ArrayList<>();
    }
}

```

在map.display()绘制一帧图像的时候，也把当前的生物和子弹存入一个Record对象中，再把这个Record用writeObject()方法输出到文件：

```

if (needRecord) {
    try {
        writer.writeObject(record);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

对于回放，要进行反序列化，该过程是在Map的getNextRecord()方法中进行的，通过调用readObject()方法获取Record对象，再进行解析：

```
private Record getNextRecord() {  
    Record record = null;  
    try {  
        record = (Record) reader.readObject();  
    } catch (EOFException e) {  
        reviewTimeline.stop(); //结束timeline动画  
        battleState.setReviewing(false); //回放结束  
        return null;  
    } catch (IOException e) {  
        e.printStackTrace();  
    } catch (ClassNotFoundException e) {  
        e.printStackTrace();  
    }  
    return record;  
}
```

上述过程还涉及异常处理，利用EOFException来判断记录文件的结尾。

JAVA特性的使用：反射、异常处理等