中国科学: 信息科学

SCIENTIA SINICA Informationis

论文

Developing a Java Game from Scratch

谷石磊[1]

1. 南京大学, 南京 210023
E-mail: 211220079@smail.nju.edu.cn

摘要 从零开始基于 libGDX 实现的一个 Java 小游戏，使用 Gradle 进行构建，技术栈包括了多线程，IO，NIO，网络通信，测试，Gradle 等。本文介绍了实现过程中相关的问题和解决办法，以及相关的设计和思考。

关键词 Java, libGDX, 网络通信, 测试, 多线程

1 游戏介绍

本次作业实现了一个简单的对战游戏。游戏分为两方，在一张地图上进行战斗。地图分为左右两边，玩家和右方位于地图左半边，敌人位于地图右半边，双方都不可越过中间线。地图默认为 10 X 10 的格子，双方各有 5 个人位于两侧。双方可进行上下左右的移动和攻击操作。在单机模式中，玩家操纵一个角色，其余角色全由 AI 控制，那一方全部阵亡则对面获胜。在联机模式中，多个玩家位于同一方，位于地图左边，和右方的敌人进行战斗。在游戏中，AI 操纵的角色每隔一定时间会做出攻击和移动操作，玩家操纵的角色通过 wasd 或上下左右进行移动，鼠标单击进行攻击，攻击方式为发射子弹，玩家发射的方向为鼠标单机时所在的位置。为增加游戏平衡性和难度，对玩家的攻击操作和移动操作有限制，一定时间内只能进行一次操作。

总体而言，游戏趣味性和可玩性较低，属于是为完成作业带来的赶工之作。之所以选择这种地图模式，主要还是认为它比较简单，相比于其他形式，这种形式在实现上有很大的便利性，有些取巧了。这种地图只需维护一个代表地图的矩阵即可，对于互斥方面比较容易。

2 设计介绍

2.1 游戏基本实现

在最开始阶段的实现中，游戏的整体结构如下:

```
com
└── mygdx
    └── bullet
```

```
    ├── Bullet.java ..... 子弹
    └── BulletUpdater.java ..... 负责更新子弹的线程
├── character
    ├── Character.java ..... 角色
    ├── Enemy.java ..... 敌方
    └── Hero.java ..... 我方
├── constants
    └── Constants.java ..... 一些常量
├── entity
    └── Entity.java ..... 游戏实体
├── game
    ├── GameScreen.java ..... 游戏界面
    ├── MainMenu.java ..... 主菜单
    ├── MyGdxGame.java ..... 入口
    └── ResultsScreen.java ..... 结果界面
└── matrix
    └── Map.java ..... 地图
```

对于地图有一个 ‘Map‘ 类，里面维护了一个代表地图资源的数组，通过 ‘get‘ 和 ‘set‘ 方法来实现对地图资源的访问。

对于所有的游戏实体有一个基本的 ‘Entity‘ 类来实现，其中实现了一些基本的属性和方法，包括位置、血量、攻击力等。‘Entity‘ 有两个子类：‘Character‘ 和 ‘Bullet‘。

‘Character‘实现了一些角色的共同的属性和方法，它有两个子类：‘Enemy‘ 和 ‘Hero‘，他们分别代表地方和我方角色，实现了各自独有的属性和方法。为了实现多线程，‘Enemy‘ 和 ‘Hero‘ 都实现了 ‘Runnable‘ 接口，每个角色有一个单独的线程控制。

‘Bullet‘ 代表子弹类，实现了子弹的属性和方法，考虑到子弹较多，因此将所有的子弹由一个线程控制，该线程实现于 ‘BulletUpdater‘。

一共实现了三个界面，主菜单 (‘MainMenu‘)，游戏界面 (‘GameScreen‘)，结果界面 (‘ResultsScreen‘)。主菜单实现了游戏的开始和退出，游戏界面实现了游戏的主要逻辑，结果界面实现了游戏结束后的结果展示。此时游戏逻辑都在 ‘GameScreen‘ 中实现，‘GameScreen‘ 有一个 ‘Map‘ 类型的属性，用于维护地图资源。我认为这种实现方式比较简单，但是也有一些问题，比如说 ‘GameScreen‘ 类的代码量过大，不够清晰，而且 ‘GameScreen‘ 类中的逻辑过于复杂，不够清晰，不利于后期的维护和扩展。

在多线程实现的过程中，并没有遇到太多的问题，因为设计比较简单粗暴，使用 “* 一把大锁保平安 *” 的方式避免了数据竞争的问题，但是效率不高。对与 ‘Map‘ 类的访问，使用 ‘synchronized‘ 关键字来实现同步，对于 ‘Bullet‘ ‘Hero‘ ‘Enemy‘ 的访问不存在数据竞争问题，对于他们的集合，使用了 ‘CopyOnWriteArrayList‘ 来实现，这样可以避免在遍历的过程中对其进行修改导致的异常。

在线程的实现中，起初想使用 ‘while(true)‘ 循环加上 ‘sleep‘ 来实现定期的调用，但是这种实现存在隐患，采用了 ‘ScheduledThreadPoolExecutor‘ 来实现定期的调用，这样可以避免 ‘sleep‘ 的浪

费问题，而且也更灵活，同时线程池的实现也更加高效。

2.2 增加测试

在最开始的实现中，没有进行测试，后来增加了单元测试。测试的代码在 ‘test‘ 目录下。测试过程中主要的问题在于由于官方没有提供相关的测试框架，因此需要自己实现测试框架，这一点花费了较多的时间。测试框架的实现在 ‘TestRunner‘ 类中。‘TestRunner‘ 主要实现了测试时的环境问题，通过使用 ‘HeadlessApplication‘ 类和补充一些 ‘mock‘ 方法来实现。测试框架的实现参考了这个代码 (https://gist.github.com/travishaynes/57bd19616c834bc691e8ffe670173a96)。

测试的代码在 ‘test‘ 目录下，其组织方式遵循一般规则，测试的代码主要是对 ‘Map‘ 类和 ‘Character‘ 类的测试。

2.3 增加 IO 功能

在这个阶段，游戏的整体框架如下：

```
com
└── mygdx
    ├── bullet
    │   ├── Bullet.java
    │   └── BulletUpdater.java
    ├── character
    │   ├── Character.java
    │   ├── Enemy.java
    │   └── Hero.java
    ├── config
    │   └── Config.java ..................................将 Constants 类改名为 Config 类
    ├── entity
    │   └── Entity.java
    ├── game
    │   ├── AbstractGameScreen.java ..........................抽象出了一个游戏界面的类
    │   ├── BaseScreen.java ..................................................基本界面
    │   ├── ChangeMapScreen.java .............................................改变地图界面
    │   ├── GameScreen.java
    │   ├── MainMenu.java
    │   ├── MyGdxGame.java
    │   ├── ReadRecordScreen.java ............................................读取存档界面
    │   ├── ReplayScreen.java ................................................回放界面
    │   └── ResultsScreen.java
    └── matrix
```

```
└── Map.java
```

主要的变化在 ‘game‘ 包中，增加了一些界面，分别实现了不同的功能。随着界面的增多，提取出了他们的公共部分，抽象出了 ‘BaseScreen‘ 类实现了一些基本的界面功能，从而减少了代码量，提高了代码的可读性。同时，由于回放界面和游戏主界面的实现有部分相同，因此抽象出了一个 ‘AbstractGameScreen‘ 类，提取出了他们的公共方法。我认为这是一个比较差的设计，强行提取出了一个父类，实际上没有太多的共同点，只是为了减少代码量，这样的设计不够合理。

对于存档和回放的实现，使用序列化和反序列化的方式实现，将游戏的状态保存在文件中，通过读取文件来实现存档和回放。在序列化时，将一些无法序列化的属性设置为 ‘transient‘，并在反序列化时重新加载这些属性，从而避免了无法序列化的问题。在录制和回放时，将每一帧的游戏状态都序列化到文件中，通过在每一帧读取文件来实现回放，这样的实现简单粗暴，但是缺乏效率。个人认为更好的方式是存储每一帧的操作，而不是存储每一帧的游戏状态，这样可以减少存储空间，提高效率。

在实现录制时遇到了一个十分令人困扰的 Bug，就是在录制时，将每一帧的状态都保存，在回放时，且发现只保存了第一帧，之后每次读取都是第一帧的状态，经过研究，发现在序列化时，如果变量指向的对象的状态没有发生变化，那么序列化时，第一次序列化会正常序列化对象，但是在之后的序列化中，不会再次序列化，而是认为这个和之前的是同一个对象，只是做一个标记，指明和之前一样。举个例子：我要序列化一个 ‘list = List<Hero>;‘，在第一次序列化时，会将 ‘list‘ 中的所有元素都序列化，但是在第二次序列化时，如果 ‘list‘ 中的元素没有发生变化，那么就不会再次序列化，而是认为这个 ‘list‘ 和之前的是同一个，只是做一个标记，指明和之前一样。即使你的 ‘list‘ 中的元素指向的对象（如 ‘list‘ 中的第一个 ‘Hero‘ 的属性发生了变化）发生了变化，但是 ‘list‘ 指向的对象没有发生变化，那么也不会再次序列化，而是认为这个 ‘list‘ 和之前的是同一个。为解决这个问题，有两种解决办法，一种是在每一帧序列化时，都重新创建一个 ‘list‘，进行一次深拷贝，这样就不会认为是同一个对象了；另一解决办法为调用 ‘reset‘ 方法，这个方法官方解释如下：

> `Reset` will disregard the state of any objects already written to the stream. The state is reset to be the same as a new `ObjectOutputStream`. The current point in the stream is marked as reset so the corresponding `ObjectInputStream` will be reset at the same point. Objects previously written to the stream will not be referred to as already being in the stream. They will be written to the stream again.

即调用 ‘reset‘ 方法后，会将之前的状态抛弃，这样就会重新序列化，就可以解决这个问题了。

此外，从本阶段开始，引入了 ‘lombok‘ 库，使用 ‘@Getter‘ 和 ‘@Setter‘ 注解来自动生成 ‘getter‘ 和 ‘setter‘ 方法，这样可以减少代码量，十分方便。

2.4 增加网络功能

在这个阶段，游戏的整体框架如下：

```
com
└── mygdx
```

```
├── bullet
│   ├── Bullet.java
│   └── BulletUpdater.java
├── character
│   ├── Character.java
│   ├── Enemy.java
│   └── Hero.java
├── config
│   └── Config.java
├── controller
│   ├── GameController.java ..... 游戏逻辑控制
│   ├── GameState.java ..... 游戏状态
│   └── TextureManager.java ..... 纹理管理
├── entity
│   └── Entity.java
├── event
│   ├── CharacterAttack.java ..... 角色攻击事件
│   ├── CharacterMove.java ..... 角色移动事件
│   ├── GameEvent.java ..... 游戏事件基类
│   └── HeroAttack.java ..... 英雄攻击事件
├── game
│   ├── BaseScreen.java
│   ├── ChangeMapScreen.java
│   ├── GameScreen.java
│   ├── MainMenu.java
│   ├── MyGdxGame.java
│   ├── ReadRecordScreen.java
│   ├── ReplayScreen.java
│   ├── ResultsScreen.java
│   └── StartScreen.java
├── map
│   └── Map.java
├── nio
│   ├── GameClient.java ..... 客户端
│   └── GameServer.java ..... 服务端
└── observer
    └── GameObserver.java ..... 游戏观察者接口
```

在这个阶段, 游戏的逻辑全部在 ‘GameController‘ 类中实现, 将游戏逻辑从 ‘GameScreen‘ 中抽

离出来，这样的设计更加合理，也更加清晰，更加符合单一职责原则。同时也删除了 ‘AbstractGameScreen‘ 类，因为它的存在没有太多的意义，如今使用组合替代了继承，在 ‘ReplayScreen‘ 中也修改为使用 ‘GameController‘ 类来实现对应的逻辑。这个改动深化了我对组合和继承的理解。相较于继承，组合要更为灵活。

又提取出了一个 ‘TextureManager‘ 类，用于管理纹理，这样可以避免重复加载纹理，提高效率。这个类使用单例模式来实现，因为只需要加载一遍。在具体的实现过程中，采用了内部静态类的方式来实现，具体如下：

```
    public class TextureManager {

    private TextureManager() {
    }

    private static class TextureManagerHolder {
        private static final TextureManager INSTANCE = new TextureManager();
    }

    public static TextureManager getInstance() {
        return TextureManagerHolder.INSTANCE;
    }
}
```

使用这种方式既实现了懒加载，又保证了线程安全，而且使用了静态内部类的特性，不依赖外部类的实例，同时避免了性能开销。

在这个阶段，增加了网络功能，使用 ‘nio‘ 包中的 ‘GameClient‘ 和 ‘GameServer‘ 类来实现。我们首先需要开启一个服务端，然后开启三个客户端，服务端会首先发送初始游戏信息给客户端，游戏开始后，客户端会发送自己的操作给服务端，服务端会将这些操作进行处理，之后将处理后的结果发送给客户端，客户端再进行更新。这样就实现了多人联机的功能。在实现过程中，遇到了一些问题，

首先，我们需要将游戏的状态进行序列化，然后发送给客户端，客户端再进行反序列化，这样就可以实现客户端和服务端的同步。此处使用了 ‘jackson‘ 库将游戏状态等信息序列化为 ‘json‘ 格式，然后发送给客户端，客户端再进行反序列化，这样就可以实现客户端和服务端的同步。

其次，服务端需要能够观察到游戏的变化，从而将其发送给客户端，这里使用了观察者模式。我们在 ‘GameState‘ 中，维护了一个 ‘GameObserver‘ 的集合，当游戏状态发生变化时，就会通知所有的观察者：

```
    public class GameState {
    private List<GameObserver> observers = new ArrayList<>();

```

```
    public void addObserver(GameObserver observer) {
        observers.add(observer);
    }

    public void removeObserver(GameObserver observer) {
        observers.remove(observer);
    }

    public void notifyObservers(GameEvent event) {
        for (GameObserver observer : observers) {
            observer.onNotify(event);
        }
    }
}
```

我们只需要在原来游戏变化处增加少量的的代码就能实现对于观察者的通知。同时，为了便于发送与序列化，我们创造了 ‘GameEvent‘ 及其几个子类来实现对于游戏变化事件的处理。这样发送信息时只需要将这个类序列化即可。

在服务端中，实现了 ‘GameObserver‘ 接口，当服务端接收到客户端发送的操作时，就会通知所有的观察者，这样就实现了服务端对游戏状态的观察，从而实现了服务端对客户端的同步。

然后在实现网络通信的过程中，由于对 ‘NIO‘ 的不了解，导致了一些问题，最后的解决方案为：在服务端为每一个 ‘channel‘ 都维护一个 ‘buffer‘ 来存储信息，当 ‘channel‘ 可读时，就将信息读入 ‘buffer‘ 中，然后再从 ‘buffer‘ 中读取信息，这样就可以实现信息的正常发送与接受。在客户端，使用阻塞式的 ‘socket‘ 来实现，当客户端接收到信息时，就将信息反序列化，然后更新游戏状态；当用户实现操作时，就直接发送信息。

此外，为了支持 ‘jaskson‘ 库的使用，增加了 ‘lombok‘ 的 ‘@NoArgsConstructor‘ 注解，用于生成无参构造函数，同时增加了 ‘@JsonIgnoreProperties()‘ 注解，用于忽略无法或不需要序列化属性，这样就可以避免反序列化时的异常。

在本阶段，遇到了一个很困扰的 Bug，Bug 的产生原因如下：使用矩阵表示地图，在矩阵中的移动和放置使用的数据都为 0-9 的数据（以 10X10 为例），但对于实际地图中的移动，会乘一个 ‘CELL_SIZE‘ 来表示实际地图中的大小。此时，由于不同的移动和放置参数不同，造成了混乱，从而产生了 Bug。由于刚开始没有意识到，困扰了半天。

为避免这种情况，想到了三种解决方案：

- 一是统一所有的参数，均采用同一种参数；
- 二是在注释中标明传递的参数的限制；
- 三是增加传入参数的判断，例如使用 ‘assert‘ 语句来判断传入的参数。

意识到这个问题后，由于第一种方式工作量较大，我采用了第二种和第三种方式混合的方法来避免错误。

本阶段在使用多态的过程中，遇到了一个问题，加深了我对多态和泛型的理解，首先看一个示例代码:

```
class F {}

class G extends F {}

class H extends F {}

public class Main implements Serializable {
  public void f1(F f) {
    System.out.println("f");
  }

  public void f1(G g) {
    System.out.println("g");
  }

  public void f1(H h) {
    System.out.println("h");
  }

  public <T extends F>void f2(T f) {
    f1(f);
  }

  public void f3(F f) {
    f1(f);
  }

  public static void main(String[] args) {
    Main main = new Main();

    main.f2(new G());
    main.f2(new H());

    main.f3(new G());
    main.f3(new H());
```

```
  }
}
```

起初我以为这两种方式输出的结果都为 ‘g h‘，可事实上输出结果都为 ‘f f‘。

首先解释 ‘f3‘，将一个方法调用和一个方法主体关联起来称作绑定。若绑定发生在程序运行前（如果有的话，由编译器和链接器实现），叫做前期绑定。它是面向过程语言不需选择默认的绑定方式，例如在 C 语言中就只有前期绑定这一种方法调用。而 ‘Java‘ 通过后期绑定来实现多态，后期绑定意味着在运行时根据对象的类型进行绑定。后期绑定也称为动态绑定或运行时绑定。在传递给 ‘f3‘ 参数时，会将参数的类型认定为 ‘F‘，从而选择与方法 ‘void f1(F f)‘ 进行绑定，故输出为 ‘f f‘。

接着解释 ‘f2‘，‘Java‘ 泛型的实现方式是将类型参数用边界类型替换，在上面的例子中就是把 ‘T‘ 用 ‘F‘ 替换。我们编译字节码看一下：

```
  public <T extends org.example.F> void f2(T);
    Code:
      0: aload_0
      1: aload_1
      2: invokevirtual #25            // Method f1:(Lorg/example/F;)V
      5: return

  public void f3(org.example.F);
    Code:
      0: aload_0
      1: aload_1
      2: invokevirtual #25            // Method f1:(Lorg/example/F;)V
      5: return
```

我们发现他们的字节码完全一样，这就恰好证明了泛型的类型擦除。

由于没有找到好的解决办法，我选择了使用 ‘instanceof‘ 来将父类对象转变为子类对象，之后再调用相应的多态函数。

3 结语

总体而言，这次作业的任务基本完成，但实现的较为一般。由于前期没有进行设计，导致后期的实现较为困难，因此在实现过程中，多次进行重构，导致了代码的冗余。同时，由于没有安排好时间，前期没怎么写，而后期又有较多的事情，因此时间较为紧张，导致了代码的质量不高，代码的可读性和可维护性较差。

在这次作业中，我学到了很多东西：

- **第一阶段：**学会了多线程的用法，使用了 `ScheduledThreadPoolExecutor`。

- **第二阶段：**掌握了 `gradle` 的基本使用。
- **第三阶段：**学会了 `junit5` 的使用。
- **第四阶段：**学会了 Java 的序列化和反序列化，以及 IO 相关的函数，了解了 `lombok` 库。
- **第五阶段：**学会了 `NIO selector` 的使用，了解了 `jackson` 库，了解了单例模式和观察者模式。

这次作业虽然带来了一些痛苦，但是收获很大。

在完成这个项目的过程中，我深刻体会到了软件开发的挑战和乐趣。通过设计和实现这个项目，我不仅加深了对 Java 编程语言的理解，还学到了许多关于项目管理、解决问题和持续改进的宝贵经验。

这个项目不仅仅是一次技术上的探险，也是一次个人成长的旅程。在面对挑战时，我学到了灵活性和创造性的重要性。解决问题的过程中，我逐渐掌握了调试技术和系统优化的方法，这将在我未来的开发工作中大有裨益。

这个项目不仅是我技术上的一个里程碑，更是我职业发展的一个起点。通过这次经历，我深信，不断学习、挑战自己，将是我未来前进的动力。我期待着将这次项目的收获应用到未来的项目中，并继续不断提升自己。

再次感谢所有在这个项目中支持我的人，你们的帮助让这个旅程变得更加有意义。

Developing a Java Game from Scratch

Shilei Gu[1]

1. *NJU, Nanjing* 210023*, China*
E-mail: 211220079@smail.nju.edu.cn

Abstract Creating a Java mini-game from scratch using libGDX, this article employs Gradle for build management and encompasses a tech stack that includes multithreading, IO, NIO, network communication, testing, and Gradle itself. The document details challenges encountered during the implementation, provides solutions, and offers insights into the design principles and considerations that shaped the development process.

Keywords Java, libGDX, network, testing, multithreading