中国科学 : 信息科学 在审文章 onlinelatex

SCIENTIA SINICA Informationis

论文

Developing a Java Game from Scratch

宋承柏

* 通信作者. E-mail:

摘要 本文介绍了一次开发基于 libGDX 的 Java 游戏的过程。

关键词 Java, Game Developing, libGDX

1 开发目标

我制作的游戏为传统的 Roguelike 割草小游戏，玩家将负责扮演一位骑士与怪物战斗，并尽力生存。支持多人同时参与对战，支持存档、加载和录制游戏过程。游戏拥有较为丰富的人物动画，便捷的操作和美观的用户界面。制作灵感来自于用作示例的元气骑士游戏，以及我之前开发过的一个类似项目。

我在本游戏中使用的素材来自网络或由 AI 绘制。

2 游戏简介

游戏的主要功能包括：

进入游戏后首先来到主菜单界面，填写想要进入的服务器地址和用户名即可开始游戏。

使用用户名可以获取对应的录像，点击 Replay 可进入录像回放。

按键说明：

- WASD：移动
- J：攻击
- K：显示攻击判定（蓝）、受伤判定（红）矩形。再次按下时关闭该视图。
- P：保存当前游戏状态。
- L：即时记载已保存的游戏。
- R：开启录制，录制时画面左上角会显示闪烁的 Recording...字样，再次按下时结束录制。
- ESC：退回到主菜单。

功能演示：

- 保存、加载和录像：https://www.bilibili.com/video/BV15iCKYREEG

视频中演示了在角色被怪物攻击，血量将要清空时，使用加载功能恢复到之前的状态。录像使用逻辑帧记录信息而不是直接截图重放，因此再次播放时，帧率可能不同，但状态不会变化。

- 并发的生物决策：https://www.bilibili.com/video/BV1HHCKYNEFL

每一个生物体绑定两个矩形，分别表示攻击范围（蓝）和身体大小（红）。在游戏中按下 K 键，即可开启显示矩形的视图。该视图仅在开启的客户端可见，在其他客户端、服务器端、录制回放中均不可见。当攻击范围与敌方身体范围 overlap 时，我方攻击可对敌方造成伤害。友方的身体可以堆叠，不同方身体无法堆叠。我方人物无法越过地图上的障碍物。怪物方会根据我方位置进行决策，若足够接近会进行攻击，否则不断靠近我方最近的单位。如视频最后显示，当怪物数量很多时，我方在攻击下迅速死亡。然后怪物进入 Idle 状态，等待其他客户端接入游戏。

- 网络通信：https://www.bilibili.com/video/BV14gCKYkEZ8

本视频演示了 jwork05 功能要求中的网络通信。视频开始时首先在我的电脑上启动了服务器进程，然后启动 3 个客户端进程。分别输入服务器地址和用户名信息后，进入游戏房间。然后演示了不同客户端分别控制自己对应的角色，由按键录制信息可见，每个客户端的输入仅对自己的角色有效。当对服务器进程输入时，没有任何效果。服务器进程仅展示了游戏画面。由窗口画面对比可知，每个客户端所见的角色、怪物、地图信息均一致。最后，本视频还展示了当场上有多个角色时，怪物会选择距离最近的角色展开行动。而当有客户端中断连接离场时，该名角色会从所有用户的游戏画面中消失。

3　设计理念

3.1　开发框架的选择

在最初选择开发框架时，我对比了 lwjgl、libGDX、FXGL 等图形框架，最终选用了 libGDX。因为 libGDX 是跨平台的开发框架，而且有完善的文档和 wiki，并且简单易用。

因此我首先学习了 libGDX 的官方文档和示例项目，了解了其生命周期机制，以及对文件处理、输入处理、资源管理、对象序列化的支持方式。然后选用了其中的 scene2D 框架来制作游戏的主体部分。

scene2D 框架由 Game, Screen, Stage, Actor 组成。当游戏开始运行时，Lwjgl3Launcher（libGDX 在桌面端封装了 lwjgl 框架）会启动 Game 类，Game 类为对整个游戏程序的封装，它可以设置其要显示的 Screen，如 `MainMenuScreen`, `GameScreen`，从而管统一理窗口内不同的游戏界面，方便统一管理 UI、游戏画面。在 Screen 类里面，可以使用 Stage 类统一管理所有要显示的对象，而这些对象可以继承 Actor 类，从而实现对位置信息、状态、动画帧、绑定矩形的统一管理。该框架很大程度上方便了画面的渲染，减少了工作量。

但是随之而来的也是很大的缺陷：Scene2D 框架不符合标准的 MVC 设计模式，它将数据模型、控制器和视图全部封装在一起，造成渲染与逻辑的耦合，对我后面实现网络模块、进行测试带来了极大的不便。为了达成任务要求，我努力对其进行了解耦，但是未能完全抽离渲染和游戏逻辑。

3.2　我的总体设计

我的整个项目分为两个子项目：core 和 lwjgl3，分别负责实现游戏的核心逻辑和启动服务器和客户端窗口程序。

在 core 子项目中有以下主要的类：

- Main 类：继承自 Game，负责管理整个游戏流程。
- FirstScreen 类：负责呈现客户端的主菜单，管理按钮、文本框等 UI。
- GameScreen 类：负责客户端的游戏画面管理，包含有启动客户端程序、监听服务器消息、录制游戏画面、渲染画面、接受用户输入并向服务器发送等方法。GameScreen 包含有 GameWorld 对象，但不会主动更新游戏数据，而是在其渲染方法中根据服务器消息设置 GameWorld 并进行画面更新。
- ReplayScreen 类：负责读取录制文件信息，在其渲染方法中重建场景。
- ResourceManager 类：负责管理所有资源文件。
- GameWorld 类：实现游戏逻辑的核心，对游戏的地图、生物等所有数据建立模型。

Creatures：

- Knight：游戏的主角，玩家操控的对象。
- Orc：怪物，使用独立的线程池管理行为。

Tiles:

- Square：普通的地图方块。
- Obstacle：障碍物方块。

Network：

- GameClient：实现了游戏客户端逻辑，与 Server 收、发消息。
- GameServer：实现了游戏服务器逻辑。
- Message：包含 ControlMessage, GameStatMessage, RequestMessage 等不同的消息类型，便于序列化和反序列化消息内容。

在 lwjgl3 子项目中有以下主要的类：

- Lwjgl3Launcher：负责启动整个客户端程序。
- ServerLauncher：负责启动整个服务器程序。
- ServerGameScreen：负责服务器的游戏画面管理，负责监听客户端请求，并不断广播实时的游戏数据，从而驱动客户端画面更新。该类中包含有 GameWorld 对象，并真实进行游戏数据的 update。

3.3　有什么好处？

我的设计拆分了 Server 和 Client 的逻辑，并使用 GameWorld 对象统一为游戏数据建模，从而使 Server 和 Client 能共享大部分必要的代码，并且尽力减少了逻辑和渲染的耦合。

在我的框架中，Client 只负责两件事：接收用户输入并向 Server 发请求、接收 Server 消息并更新画面。这使得 Server 和 Client 的职责划分更清晰，并有效避免了不同 Client 画面不一致问题。

在 Client 端，使用不同的 Screen 来管理不同功能的画面，使得模块划分更清晰，同时便捷地实现了切换显示画面来满足游戏、回放、设置的需求。

4　技术问题

在开发过程中我遇到了大量的技术问题。

4.1　网络通信

网络模块是我遭遇技术问题最多的地方，也消耗了大量时间 debug。

耦合带来的弊端　事实上，我是以单人游戏为起点来开发本项目的。因此在我最初的项目结构中，并没有像现在的 GameWorld 类这样的一个类封装游戏的所有数据状态，而是将各种逻辑散布在 GameScreen、Creature 类以及渲染方法内。

当我开始开发网络模块时，才意识到问题的严重性。因为需要对客户端和服务端进行职责拆分，而客户端只应该负责接收用户输入并向服务器发送、接收服务器消息并更新画面，因此不应该在其渲染方法中对其持有的人物、地图进行任何主动的数据更新，它只能等待服务器的消息然后被动地更新；而服务端则不应该响应任何直接的用户输入，它只能等待客户端通过网络发送消息来改变角色状态。

于是我重构了代码，从原来的设计中抽取了数据模型：GameWorld 类，让它负责对地图、人物进行统一管理，并负责碰撞检测、生成怪物和更新状态。它的每一次 update 可以看做是游戏的“逻辑帧”。然后由服务端将状态信息打包并序列化，再通过网络发送给客户端从而驱动其更新画面。

TCP 粘包、拆包问题　在开发过程中我遇到了一个奇怪的 Bug：客户端接收到来自服务器消息时，有时会将两条消息合并，有时又会将一条消息拆分，从而导致数据无法正确解析引发崩溃。我通过打印客户端接收到的消息观察到了这一现象。

通过查询资料，我得知这是 TCP 存在的粘包和拆包问题。其原因可能是由于发送方默认使用 Nagle 算法，该算法会将多个小数据包合并成一个大数据包发送，以减少网络中的报文段数量；也可能是因为接收方在接收到数据包后，将数据包保存在接收缓存中。如果接收缓存中的数据包数量超过了应用程序处理的速度，就会导致多个数据包粘在一起。

为了解决粘包问题，我使用了在消息末尾加`\n`作为结束标志的方式，简洁地解决了粘包；而拆包问题则比较麻烦，需要接收方确认收到完整的消息时，才执行解析。因此我使用了一个 `messageBuffer` 暂存消息，对于未解析的数据，保留到下一次接收中继续处理，直到读到完整消息才进行解析。

如何高效地序列化？　为了在 Server 和 Client 之间传递消息，我使用了 libGDX 提供的 json 序列化功能来进行对象图的序列化。该库支持对所有 POJO 直接进行序列化，但是由于我使用了 scene2D 框架，其中的类有复杂的继承关系并且设计图形，无法对人物等类直接序列化，因此我手动实现了 `Json.Serializable` 接口自定义了序列化方法。

最初我将对象的几乎所有字段写入了序列化信息，并且在 Client 端直接对消息进行解析并全部反序列化，然后将原有对象全部清除，通过 set 方法设置为新对象。但经过测试发现在 Client 段发生了严重的卡顿。

后来我改进了序列化方法，只写入与状态更新有关的消息，在 Client 段反序列化后，寻找差异的字段进行更新，解决了卡顿问题。

为了进一步提高序列化效率，我还学习了使用 Kryo 进行二进制序列化，并了解了在此基础上开发的、专用于此类游戏的对象通信的 KryoNet 库。但由于在我的项目中 Json 序列化已经满足的简单的通信要求，故没有进一步尝试。

4.2　文件 IO

在解决了网络相关的问题后，我很快意识到持久化和网络通信的实现方式是类似的，区别仅在于网络模块是将序列化消息发送出去，而持久化则是将序列化消息写入磁盘。

而保存和录像的实现方式也是类似的，保存是指对某一逻辑帧的全局状态信息生成快照并写入文件，那么也可以通过序列化所有状态来解决；而录像就是在某一时段内对每一逻辑帧的数据依次保存并写入文件。

因此我设计了 GameStat 类专用于记录某一时刻的游戏状态信息，包括玩家角色和敌人的所有信息。它可以被当做 GameWorld 在某一瞬间的快照。然后在 GameWorld 类中实现了 buildFromGameStat 方法和 getGameStat 方法。通过这套接口，我将保存、加载、录制、回放、网络通信的实现统一了起来。

4.3　并发控制

libGDX 由于使用了 OpenGL，所有渲染有关的工作都必须在渲染线程执行，而 scene2D 框架中的 Actor 类直接和图形相关，因此都是线程不安全的，实现并发控制生物行为并不容易。

不过 libGDX 提供了 `Gdx.app.postRunnable` 方法，它接收一个 Runnable 对象，于是我们可以通过 lambda 表达式将其它线程的计算结果传回主线程，这样就能实现并发决策。

于是我实现了 Orc 类的 startBehavior 方法，Orc 类持有一个静态的 `ExecutorService` 线程池，startBehavior 通过向线程池 submit 新的 Runnable 来实现多线程。在线程池中的线程完成选择目标、计算目标距离、选择行动等决策，然后通过 `Gdx.app.postRunnable` 传回主线程，主线程中的对象完成状态更新。同时，由于其读取了 GameWorld 中的各种状态信息，所有相关的方法都必须加锁进行保护。

4.4　测试

失败的尝试　由于技术选型的潜在问题，测试给我带来了极大的困扰。最初我因为绘制画面简单便捷而选用了 libGDX 下的 scene2D 框架，然而这个框架造成了数据模型和渲染模型的耦合。我使用 scene2D 框架中的 Actor 类来表示所有的生物体，在渲染方面，这确实使工作简便不少，因为所有的 Actor 都和某个 Stage 绑定，在设置好 Actor 的位置信息等属性后，我只需要调用 `stage.draw()` 即可完成所有对象的绘制。

然而当我着手开始写测试方法时，却发现几乎完全无法在不启动游戏窗口的情况下初始化任何对象。经过调试发现，scene2D 中几乎所有的对象都依赖于 OpenGL 上下文，因此无法在简单的单元测试环境中完成构造。

经过漫长的调试，我尝试了几种方法来绕过：

1. 使用 Headless Backend + mockito：libGDX 提供了一种无头后端，用于在不需要图形渲染的环境下运行 libGDX 程序。然而在 scene2D 框架下这没有成功，因为 Headless Backend 无法为 SpriteBatch，shaperenderer 等特殊的图形对象提供上下文。另外，由于 Stage 类内部自行定义了 Batch 字段并有初始化方法，所以难以使用 mockito 向它传递 mock 的 Batch 对象。

2. 使用 Xvfb 虚拟显示器：由于无头后端的尝试失败，我直接启用了真实的 lwjgl3 程序来进行测试。但是为了在 GitHub Actions 中执行自动化测试，我使用了 Xvfb 来在 GitHub 虚拟机中

模拟显示环境。然而由于 libGDX 对象的特殊要求，无论如何调试始终无法正常地启动窗口程序，因此该尝试也失败了。

最终的测试方法　经过不断尝试各种方法均告失败后，我最终使用了启动真实的桌面程序，在本地执行测试的方法。而 libGDX 的桌面程序 `Lwjgl3Application` 启动后会进入一个图形渲染循环，从而阻塞测试方法，这意味着需要在程序启动之后自动地完成测试、自行退出。另外，为了单元测试的“单元”性质，需要能够单独地启动不同的测试方法，互不干扰。因此我使用了如下的解决方案：

1. 使用 lambda 表达式动态调整测试方法：在测试类中，定义字段 `testBehavior` 为 Runnable。
2. 在每个具体的测试类中，重写 `testBehavior`，但不真正执行测试。
3. 定义 @AfterEach 方法：`executeTestBehavior`，在该方法中，创建桌面应用程序，并执行 testBehavior。此外，使用 `try-finally` 块包围，从而在测试后自动执行 `Gdx.app.exit();` 退出程序。

最后我设计了两个测试框架：

- GameLogicTest 测试了游戏的主要逻辑，包括游戏的保存和加载、生物攻击和碰撞检测、怪物自动生成、游戏数据结构内的各种设置方法、游戏对象状态的序列化、界面的绘制、生物体的生命值系统和行动。
- NetworkTest 较为简单，未覆盖网络模块的所有功能，因为网络模块需要 client-server 之间的交互，因此更适合启动窗口程序直接进行测试。我在该测试类中简单测试了 client 和 server 之间的连接情况。当同时启动 server 和 client 时，不会发生异常，当仅启动 client 时，使用 assertThrows 检测了异常情况。

最后的覆盖率结果如图1所示。

5　工程问题

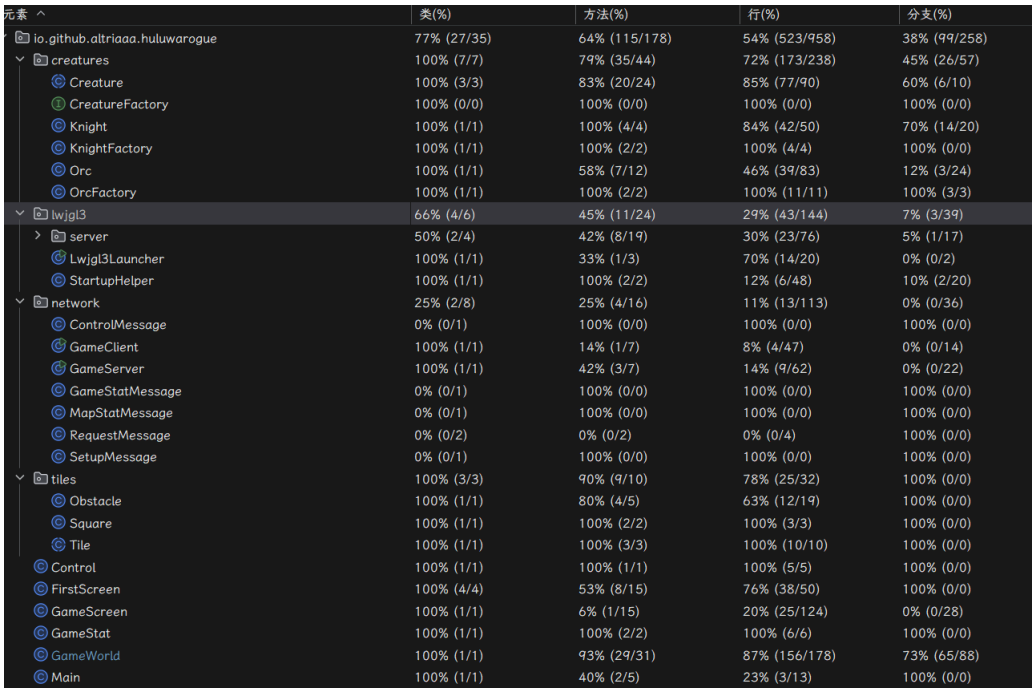
5.1　使用单例模式实现 GameWorld 以及资源的统一管理

我的 GameWorld 类使用了单例模式，在一个进程中只有一个 GameWorld 对象，从而确保了全局唯一性，并为不同的对象提供了全局访问点，简化了参数传递流程。

我的 ResourceManager 类也使用了单例模式。因为资源需要在游戏开始时统一加载到内存，在游戏结束时统一销毁（libGDX 的图形资源需要手动销毁），因此单例 ResourceManager 为资源加载、获取和销毁提供了统一的接口，防止了资源管理的混乱。

5.2　使用工厂模式创建所有 Creature

Knight 类和 Orc 类在构造器执行完之后，需要不同的初始化方式。在我的设计中，Knight 类被放置在 Stage 左下角，而 Orc 类则随机出现在画面边缘。因此使用工厂模式解决了不同的初始设置问题，并为 GameWorld 中新建对象提供了统一的接口。



元素	类(%)	方法(%)	行(%)	分支(%)
io.github.altriaaa.huluwarogue	77% (27/35)	64% (115/178)	54% (523/958)	38% (99/258)
creatures	100% (7/7)	79% (35/44)	72% (173/238)	45% (26/57)
Creature	100% (3/3)	83% (20/24)	85% (77/90)	60% (6/10)
CreatureFactory	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
Knight	100% (1/1)	100% (4/4)	84% (42/50)	70% (14/20)
KnightFactory	100% (1/1)	100% (2/2)	100% (4/4)	100% (0/0)
Orc	100% (1/1)	58% (7/12)	46% (39/83)	12% (3/24)
OrcFactory	100% (1/1)	100% (2/2)	100% (11/11)	100% (3/3)
lwjgl3	66% (4/6)	45% (11/24)	29% (43/144)	7% (3/39)
server	50% (2/4)	42% (8/19)	30% (23/76)	5% (1/17)
Lwjgl3Launcher	100% (1/1)	33% (1/3)	70% (14/20)	0% (0/2)
StartupHelper	100% (1/1)	100% (2/2)	12% (6/48)	10% (2/20)
network	25% (2/8)	25% (4/16)	11% (13/113)	0% (0/36)
ControlMessage	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
GameClient	100% (1/1)	14% (1/7)	8% (4/47)	0% (0/14)
GameServer	100% (1/1)	42% (3/7)	14% (9/62)	0% (0/22)
GameStatMessage	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
MapStatMessage	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
RequestMessage	0% (0/2)	0% (0/2)	0% (0/4)	100% (0/0)
SetupMessage	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
tiles	100% (3/3)	90% (9/10)	78% (25/32)	100% (0/0)
Obstacle	100% (1/1)	80% (4/5)	63% (12/19)	100% (0/0)
Square	100% (1/1)	100% (2/2)	100% (3/3)	100% (0/0)
Tile	100% (1/1)	100% (3/3)	100% (10/10)	100% (0/0)
Control	100% (1/1)	100% (1/1)	100% (5/5)	100% (0/0)
FirstScreen	100% (4/4)	53% (8/15)	76% (38/50)	100% (0/0)
GameScreen	100% (1/1)	6% (1/15)	20% (25/124)	0% (0/28)
GameStat	100% (1/1)	100% (2/2)	100% (6/6)	100% (0/0)
GameWorld	100% (1/1)	93% (29/31)	87% (156/178)	73% (65/88)
Main	100% (1/1)	40% (2/5)	23% (3/13)	100% (0/0)

图 1　覆盖率
Figure 1　Coverage

5.3　使用 Control 类统一输入处理

最初我在每个需要判断输入的地方调用 Gdx 的 input 方法处理输入，但很快发现造成了大量代码克隆。于是我将输入处理的逻辑封装到 Control 类中，它包含了每个按键是否按下的 boolean 字段，于是只需要将其传给涉及到输入处理的对象或方法即可。

6　课程感言

6.1　从底层做起得到的锻炼

经过多次的重构、不断地优化和漫长的调试，我终于完成了基于 libGDX 的 Java 游戏开发。Build from Scratch 给我的锻炼在于，不再对许多 API 的实现和用法感到疑虑，对常用的库函数、接口变得更熟悉而不是每次用到时都必须查手册。

例如网络模块，虽然有 Socket.IO，KryoNet 这样开箱即用的库，但是从 NIO 做起才能真正获得对底层的感知，比如我在开发过程中遇到的 TCP 粘包/拆包问题，只有从底层做起才能知道这些问题场景的处理方法，而不是成为一个只会使用现有方法的“调包侠”。

6.2　获得的经验教训

我也从本学期的课程项目中获取了很多教训。最深刻的一条就是“解耦”。因为数据模型和渲染的耦合，我在开发网络通信、进行测试时经历了巨大的痛苦。如果我重新设计项目，对于游戏场景不会再采用 scene2D 框架，而是将数据模型全部独立出来，构建标准的 MVC 设计模式。

而 scene2D 框架则应该用于 UI 设计，在主菜单界面使用该框架，能极大地简化实现。这也让我更深刻地体会到，应该在技术选型时就采取慎重态度，用正确的技术做正确的事。

无论如何，经过这一次开发游戏的经历，我不仅积累了技术经验，也形成了新的思考方式，收获了珍贵的成就感。

6.3　我对课程内容的建议

我认为课程大项目的形式很好，如果说有何改进之处，我希望老师能根据不同的技术要求，提供不同的项目供我们选择，或者可以拆分为几个小项目。因为事实上网络通信、存档录像等机制在一个游戏中不能很好地相容。真实的游戏市场上，同时包含存档、录制、实时通信对战的游戏也并不多，大多数的网络游戏或是只有短平快的对局、或是随时在服务器保存数据，客户端无法自由地操作存档。因此我在制作过程中也感到过一些困惑，不过经过思考也完成了任务要求。不管怎样，课程大项目都为我们提供了得天独厚的锻炼机会，是我喜欢的课程形式。