中国科学: 信息科学

SCIENTIA SINICA Informationis

论文

Developing a Java Game from Scratch

钟家成[1]

1. 南京大学计算机科学与技术系, 中国江苏省南京市

E-mail: zhong20000905@qq.com

摘要 我根据这学期学到的 Java 相关知识、面向对象设计思想、高级编程思想等，开发了一款简单的多人联机游戏，取名为《疯狂葫芦娃乱斗》。该游戏采用 C/S 设计，实现游戏渲染与物理计算的分离，并且支持多人联机并保持同步。服务器端使用了 Selector 技术以支持与客户端连接的低开销。客户端使用 Javafx 来实现用户图形界面的展示。

关键词 Java，面向对象，Selector，Flow

1 开发目标

我实现的这个游戏叫做《疯狂葫芦娃乱斗》，灵感取自童年经典游戏《疯狂小人乱斗》，图片如1所示。

该游戏是一款多人联机对战游戏。每位玩家可以选择一位角色与其他玩家对战，角色有一娃、二娃、三娃、六娃或者七娃。每个角色都有血条和能量条，共通的操作有左右移动、起跳、出拳，以及进入超级模式。出拳能使其他玩家扣血条。不同的角色有不同的技能，一娃有伤害加成，超级模式下伤害更高；二娃有速度加成，在超级模式下能使周围物体静止；三娃有防御加成，在超级模式下完全免疫；六娃在超级模式下可以隐身；七娃在超级模式下可以吸引前方的物体靠近。
当一个角色血条为零，判定该玩家输了。当一个玩家打败了其他玩家，他获得最终胜利。

2 设计理念

2.1 C/S 架构

这个游戏的结构如2所示。总体而言，游戏采用了 C/S 结构的设计，把游戏的逻辑计算部分和画面渲染部分完全分离开，服务器端只考虑游戏逻辑，而客户端只考虑游戏展示。这样设计的好处有三点：一是让结构更清晰，更便于开发理解和维护；而是便于实现各个客户端的同步，尤其是避免在客户端修改数据作弊的情况。

整个游戏分为 4 个部分：

图 1　疯狂葫芦娃乱斗

Figure 1

游戏世界（World） 这里负责游戏的逻辑计算，角色（一娃、二娃等）只是游戏世界里面的一个成员。游戏世界持有所有元素的位置和速度的状态信息，以固定的帧率来更新这些状态，并且处理碰撞。游戏世界会以另一帧率产生一次快照，记录下所有元素的位置及其他状态，发送给各玩家。

抽象玩家（Player） 这是对真实玩家的抽象。它从服务器得到真实玩家的操作数据，进行合法性判断，再传递给游戏世界中的角色；同时游戏世界的每一个快照会经过玩家的过滤，保留各玩家能够观察、关心的部分，再交由服务器发送。

服务器（Sever） 这里负责与客户端的交互，响应连接，并且把数据分发给各抽象玩家。

客户端（Client） 这里是实现对真实玩家的交互。这里对游戏逻辑完全无知，在这里只有各种图片以及他们的位置。客户端从服务器接收数据，确定各元素的位置，通过 GUI 展示给真实玩家。客户端会监听真实玩家的键盘事件，再解析成响应的动作（左移、右移等）发送给服务器。

其中，服务器和客户端的传输数据内容遵循我设计的一个简单协议，基本就是 **Command, arg1, arg2, ...** 形式。**Command** 是 byte 的形式，参数可能是 int、double、或者 string。

2.2　游戏世界的状态更新

游戏的另一个重要设计在于游戏世界。游戏世界里面包含各种元素，被称为实体（Entity），其中还包含一些可移动实体（Movable）。无论是角色、道具（未实现）、还是建筑都属于实体。而世界（World）负责管理这些实体的位置，而对于可移动实体，还需要管理它们的速度。位置和速度是这个世界的状态，是游戏的重要组成部分。游戏世界的主要任务就是以固定频率更新这些状态。

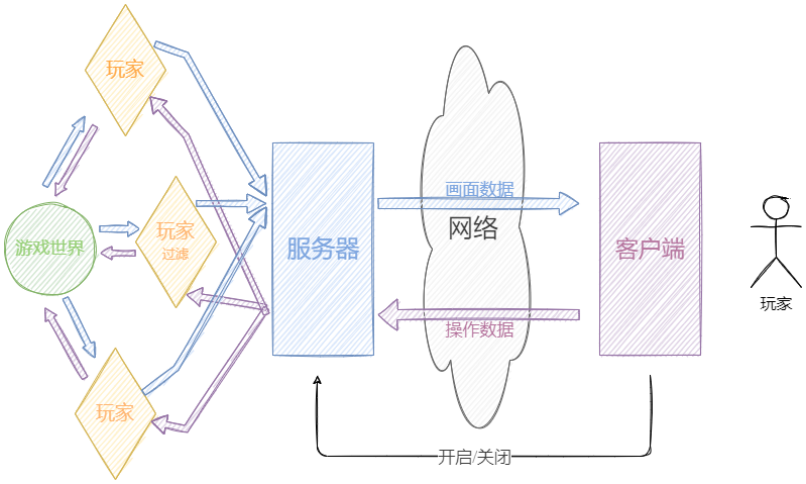

图 2　疯狂葫芦娃乱斗

Figure 2

但是游戏世界自己并不知道如何更新，比如角色是否移动等等。因此它需要开放 API 出来。然而外部直接调用 API 来更新可能导致竞争问题，引入不一致性。因此我使用事件形式来表示一次更新。所有的更新注册为一个事件，可能由世界初始化时注册，也可能由角色注册等。世界以固定频率遍历这些事件，并执行更新。

一个事件有以下属性：

类型 有 **ONESHOT**、**FINITE**、**INFINITE**，分别表示这个事件的持续时间：一次性、有限时间、永久。

优先级 不同事件可能会对同一个元素的状态进行修改，而我的解决方法是让事件分为不同优先级，低优先级会先执行，高优先级后执行。高优先级会覆盖低优先级。优先级由低到高为：**BASIC**、**CALABASH_ACTION**、**CALABASH_SUPER**、**REMOVE_ENTITY**、**VALIDATION_CHECK**、**ADD_ENTITY**。分别表示基本物理事件（重力、速度）、角色动作（左右移动、起跳）、角色大招（吸引、定格）、删除实体、碰撞检测、添加实体。

更新内容 这里表示这次更新要做什么。

在代码实现上，更新表示为 World 中的抽象内部类 Update。World 只提供 API 让外部读取状态，却不允许外部修改状态。所有修改状态的方法都封装在 Update 中，且权限为 **protected**，这使得所有的修改都发生在事件内部。外部可以实现具体的 Update，重载 update() 方法，update() 方法内可以调用修改状态的方法，最后再注册在 World 上。而在 World 内会以固定频率遍历注册的更新事件，调用其 update() 方法。这部分代码如下所示。

```
public abstract class Update{
    // 更新类型
    public final UpdateType type;
    public int remain;
```

```
    public Update(UpdateType type, int remain){
        this.type = type;
        this.remain = remain;
    }
    public Update(UpdateType type){
        this(type, 0);
    }

    // 更新内容: 需要被重载
    abstract void update();

    // Some other utils
}

// 在World的定期更新方法中
for(int i = 0;i < UpdateOrder.getTotalNum();i++){
    Queue<Update> queue = updateQueues[i];
    Iterator<Update> it = queue.iterator();
    while(it.hasNext()){
        Update update = it.next();

        if(update.type == UpdateType.FINITE){
            update.remain -= 1000 / Settings.UPDATE_RATE;
        }

        if(update.type == UpdateType.ONESHOT || update.type == UpdateType.FINIT
            it.remove();
        }
        update.update();

    }
}
```

3 技术问题

3.1 通信效率

在服务器端，假如连接用户过多，若每一个用户都分配一个线程，这样的线程的创建、切换、销毁开销会过于巨大。为了应对大量用户的情况，我采用了 Selector 技术，把所有连接的读取和写入都在一个线程内完成。

服务器在开启后会打开一个 daemon 线程，其主要内容就是轮询 SelectionKey 并且交给相应的抽象玩家处理。

另外，为了减小网络通信的开销，我对画面内容选择性地传输。服务器只向客户端传输变化的内容。比如只发送移动了的实体的位置，只发送增长了或减小了的血条或能量条。这样可以减少一些网络开销。客户端在接受数据后，只需要更改相关部分，其他内容保持不变。

3.2 并发控制

根据不同场景，并发控制采用的技术不尽相同。这个游戏共有 4 处地方可能产生线程的竞争问题。

3.2.1 游戏世界的状态

当游戏世界开启后，将会开启一个计划任务以固定的频率更新游戏状态。在这里更新涉及对状态的读取、计算、然后再写入。然而游戏世界有开放的接口，让外部获得游戏世界此刻的状态。在这里更新的写入和外部读取游戏状态就可能存在冲突，使得外部获得不一致的状态。

由于更新十分频繁，而且冲突是已知的读写关系冲突，我们不便于对整个游戏世界加锁，这会使得所有对游戏世界的访问效率非常低下（这里的访问除了包括从游戏世界读取状态，还有向游戏世界注册更新事件）。因此我的解决方案是对游戏世界中的状态进行加锁，而且是读写所。这样很大程度保证了所有对状态的读取是并发的，无论是更新还是从外部读取。而对状态的写入只会发生在短暂的一段时间内，这样很大程度保留了游戏世界的并发性。

3.2.2 角色的状态

由于游戏的逻辑，角色的血条和能量条可能不时地上下起伏，这可能由其他角色的出拳导致的，又或者随时间定时提高。总之这些原因都是异步的，可能导致对角色状态访问产生冲突。然而对角色状态的访问的频率比较低，因此我使用 **synchronzied** 关键字进行同步，所有对角色状态的访问都会被同步。

3.2.3 服务器数据的产生与发送

上面说到，我的服务器采用 Selector 技术，所有连接的读写在一个线程内完成。然而，数据的产生却是在另外一个线程。这里的是数据是指游戏世界的快照，它以一个固定频率产生。数据产生时由玩家写入 buffer，数据发送时由 socketchannel 读取，它们对 buffer 的访问会产生冲突。

我的解决方法是使用 **synchronzied** 代码块，对 buffer 对象 **synchronzied**。游戏的快照产生后，由各个玩家过滤、处理、翻译成网络上传输的数据，即刻写入 buffer，这个过程在 **synchronzied**。若遇到 Buffer 已满，则调用 buffer.wait()，等待 buffer 被 socketchannel 读取。socketchannel 读取 buffer、发送至网络后，调用 buffer.notifyAll() 唤醒玩家继续处理快照、向 buffer 写入数据。

3.2.4 客户端画面刷新

我的客户端使用 Javafx 来实现用户图形界面。客户端一方面需要从网络接收数据，改变画面中元素的位置；另一方面，Javafx 有自己的线程来维护图形界面、处理事件循环。因此不能在接收数据的同时立即改变元素的位置。

我使用 Javafx 提供的 **Platform.runLater(Runnable)** 方法来实现画面元素的调整。这个方法会把待执行的任务排入 Javafx 自己的主线程的队列中，因此所有对画面的改动都在一个线程内完成。

3.3 池化技术

一个游戏将会有很多并发任务，大部分是定时任务，还有一部分是后台常驻的任务。若每个任务都用一个线程去执行将会造成极大浪费，尤其是定时任务。因此我使用线程池技术，整个游戏，包括服务器和客户端共用同一个池子。

另外，游戏会从文件中读取许多元数据，比如实体的长宽等。假如每次使用数据都从文件读一次，这将造成巨大的文件读写开销。因此我将读取到的数据池化，以防止对同一数据的反复读取。

3.4 响应式编程

从服务器端发送的数据会经历这么一个流程：一、由游戏世界产生这个世界的快照；二、由玩家筛选出可见的、感兴趣的部分内容；三、翻译成客户端能理解的数据；四、发送至客户端。

很明显，这是一个流，因此我采用了课程上提及的响应式编程思想，采用了 Java 9 中的 Flow 的技术。让游戏世界成为 Publisher，中间的两个环节是 Processor，最后发送至客户端是 Subscriber。

4 工程问题

4.1 数据与逻辑分离

游戏里面存在很多数据。除了图片这类数据之外，还存在大量的元数据。比如每个种类的实体都需要有宽和高的数据，用来描述实体的大小，用于碰撞检测；每个图片还有偏移量，用来表示图片中心和左上角之间的偏移，用来在客户端确定贴图的位置；一张地图中记录了实体的位置等等。

这些数据不便于放在代码中，这会使得代码过于繁琐和和臃肿。因此我把这些数据剥离出来，放在文件中。一方面，这使得程序只保留逻辑部分，更加干净；另一方面，把数据记录在文件中，那么形式更加随意，方便我进行修改。然而，读取这些文件和解析的工作也过于繁琐，因此我把这部分工作集中在一起，创造出 Loader 类。Loader 类负责读取文件、解析数据。不同的数据以不同的

形式存储，因此对每种数据都有特定的方法读取、解析。Loader 还有 API 用来得到解析后的数据。游戏中加载所有数据都需要通过 Loader 类。

另外 Loader 类还维护一个池，用来存储已经读取和解析后的数据，以防止对同一个数据反复读取而反复对文件进行读写。这样能够节省很大的文件读写开销。

5 课程感想

我认为 Java 课程对我十分有益。一方面本课程不拘泥于技术细节，教了我们编程思想，比如说响应式编程、池化等；另一方面，本课程跟我们介绍了一些业界前沿的技术，而不是局限在老旧的学院知识中。