

Developing a Java Game from Scratch

徐杨鑫¹

1. 南京大学仙林校区, 中国-江苏-南京 210023

E-mail: poker-svg@foxmail.com

为爱发电

摘要 本文主要内容是描述一个网络对战的简易肉鸽风 Java 游戏的项目结构, 其研究目的是通过从零开始开发这个 Java 游戏, 来学习 Java 语言的基本语法、高级特性, 以及学习使用 Java 项目开发、测试、部署过程中所使用的工具。本文的研究结果是: 学习利用 Maven 管理项目结构和外部导入包, 以及进行测试和打包发布; 学习使用 Junit 对项目进行单元测试, 并借助 Coverage Gutters 统计测试信息并生成项目覆盖率的详细报告; 学习将 Java 的理论知识应用到实际开发过程中, 包括 OOP、设计模式、泛型、测试、自动构建等。本文得出的研究结论是: 《Java 高级程序设计》课程教的东西是真有用!

关键词 面向对象程序设计, Java, 自动构建, 单元测试, 设计模式, roguelike

1 开发目标

我想开发的游戏一个网络对战的简易肉鸽风小游戏。

具有移动、攻击、联机、迷雾、武器系统、药剂增益、传送门等地形、护甲、背景音乐、录像、保存等基本游戏元素;

具有地形隐藏、传送区域、放置炸弹、火堆回血、武器掉落、武器捡起等游戏特性;

具有提升视野、提升炸弹范围、武器充能、提升血量、提升护甲、提升移速等药剂增益;

具有武器切换、武器大招、武器攻击、人物移动、瞄准镜移动等玩家操作。

1.1 灵感来源

我的灵感来源有两个:

- 一个是我小时候玩的游戏——Q 版泡泡堂, 因为只看过葫芦娃动画片, 而葫芦娃相关的游戏没玩过, 而游戏的童年回忆则是 Q 版泡泡堂。所以想做一个相似的用炸弹炸东西的游戏。

- 另一个则是现在玩的游戏——荒野乱斗, 是一款横版自由射击游戏, 画风简洁、操作简单, 所以想做一个相似的, 有草丛、有远程进攻武器的游戏, 实际上很类似于坦克大战那种感觉。



图 1 Q 版泡泡堂
Figure 1 Q Bubble House



图 2 荒野乱斗
Figure 2 Wild fighting

1.2 游戏框架

AsciiPanel 游戏框架很戳我心，简单的画风有种复古的感觉。而且由于我没有体验过早期游戏的那种简单的画风（像 flyBird, 糖豆人等），所以这种简单复古的游戏反而让我感到新奇，所以我的游戏框架便采用了 AsciiPanel。

由于 AsciiPanel 框架自带的图片风格还是比较单调的，于是我去 roguelike 的 wiki 论坛找了个很戳我的像素风主题——Kenney · 1-Bit Pack，其所有插图如上所示。

注 1 (覆盖率) 由于 AsciiPanel 框架属于他人写好的代码，所以我将这部分代码排除在单元测试覆盖率统计之外，具体内容后面会提及。

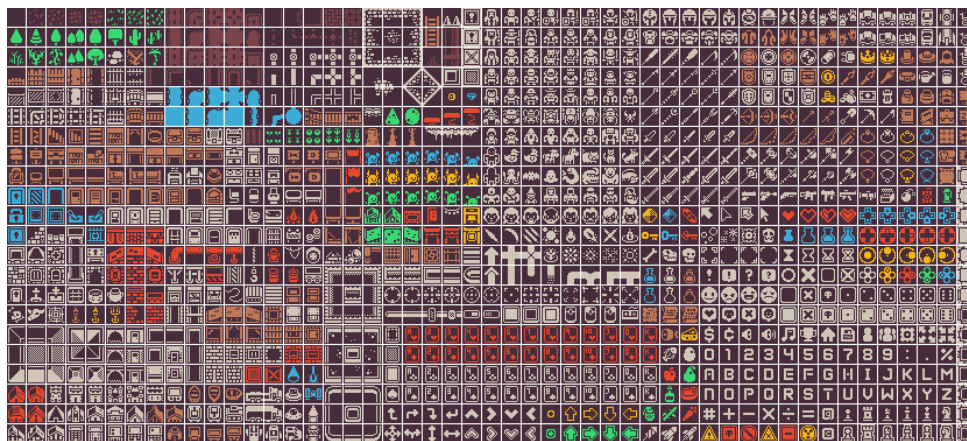


图 3 Kenney · 1-Bit Pack
Figure 3 Kenney · 1-Bit Pack





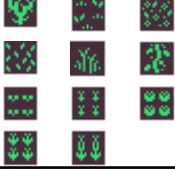


2 游戏简介

2.1 游戏项目介绍

项目代码委托在 github 上: <https://github.com/jwork-2022/jfp-poker-svg>

2.2 地形体系介绍

表 1 地形
Table 1 Terrain

外观	名称	说明
	地板	玩家可在上面自由行走
	火堆	玩家可站在上面回复血量
	传送门	玩家可站在上面被传送到另一个门
	箱子	内部装着随机的武器, 玩家可进入拿出, 也可以藏在箱子中
	墙体	会遮挡玩家视线, 玩家无法穿过, 但可以使用炸弹炸碎
	草丛	可以隐藏玩家
	树木	相当于墙体
	房屋	相当于墙体

2.3 生物体系介绍

表 2 生物

续












外观	名称	说明
	大娃	血量: 300; 手刀攻击力: 20; 护甲: 10; 视野: 9; 速度: 20
	二娃	血量: 100; 手刀攻击力: 15; 护甲: 5; 视野: 15; 速度: 20

表 2 生物

续



	三娃	血量：200；手刀攻击力：15；护甲：40；视野：9；速度：20
	四娃	血量：200；手刀攻击力：30；护甲：10；视野：9；速度：20
	五娃	血量：200；手刀攻击力：30；护甲：10；视野：9；速度：20
	六娃	血量：80；手刀攻击力：30；护甲：1；视野：3；速度：20
	七娃	血量：80；手刀攻击力：20；护甲：1；视野：9；速度：20
	蝙蝠精	
	蝎子精	血量：100；手刀攻击力：30；护甲：10；视野：9；速度：10
	蛇精	
	蜘蛛精	

2.4 武器系统介绍

表 3 生物


外观	名称	说明
	炸弹	可摧毁遮挡视物
	普通枪支	射程：5；枪支冷却时间：800ms；充能次数：3； 添加子弹间隔：1000ms；弹匣容量：3； 普攻：射出普通子弹；大招：射出双倍伤害子弹
	激光枪	射程：8；枪支冷却时间：1000ms；充能次数：5； 添加子弹间隔：2000ms；弹匣容量：5； 普攻：射出激光；大招：射出三道激光
	霰弹枪	射程：5；枪支冷却时间：2000ms；充能次数：4； 添加子弹间隔：3000ms；弹匣容量：5； 普攻：射出三枚子弹；大招：向周围发射八枚子弹
	狙击枪	射程：15；枪支冷却时间：2000ms；充能次数：3； 添加子弹间隔：3000ms；弹匣容量：3； 普攻：射出狙击子弹；大招：射出双倍伤害的狙击子弹
	棒球棍	攻击范围：2；冷却时间：2000ms；充能次数：4； 攻击持续时间：800ms；

续

普攻：对前方矩形区域造成伤害；大招：对周围方形区域造成伤害		
	火炬	非伤害武器，用于照明，可提高玩家 2 点视野
	爆竹发射器	攻击距离：15；冷却时间：2000ms； 装弹药间隔：3000ms；爆炸延迟：1000ms； 弹匣容量：3；充能次数：7； 普攻：对目标区域轰炸；大招：对目标区域造成范围更大的轰炸
	普通子弹	攻击力：10；速度：12.5；
	霰弹枪子弹	攻击力：50；速度：20；
	狙击子弹	攻击力：80；速度：50；
	激光	攻击力：30；速度：瞬间；残留时间：500ms
	爆竹	攻击力：80；范围：1；残留时间：500ms
	瞄准镜	用于投掷类武器 (爆竹) 的瞄准

2.5 玩家状态介绍

表 4 状态
Table 4 State

外观	名称	说明
	血量	
	速度	
	能量	玩家有效攻击会充能，能量充足可以放大招
	方向	用于指示玩家当前的方向
	子弹	当前剩余的子弹

2.6 药水系统介绍

表 5 药水

续





外观	名称	说明
	血药水	若血量不满则恢复 50 血量，若满血则提升 33% 血量上限
	敏捷药水	速度提高 10%
	攻击药水	手刀攻击力提高 20%
	能量药水	给武器充能一次

表 5 药水 续

	眼药水	视野 +1
	炸弹药水	炸弹爆炸范围 +1
	武器药水	武器升级一次

2.7 玩家操作介绍

表 6 操作
Table 6 Operation

按键	说明
←	玩家向左移动
→	玩家向右移动
↑	玩家向上移动
↓	玩家向下移动
X	普通攻击
Z	大招
C	切换武器
W	瞄准镜向上移动
S	瞄准镜向下移动
A	瞄准镜向左移动
D	瞄准镜向右移动
ENTER	确认选择
ESC	返回/暂停

3 设计理念

代码的总体设计遵循以下几个设计原则：

3.1 单一职责原则 Single Responsibility Principle

定义 1 (单一职责原则) 职责是指类变化的原因。如果一个类有多于一个的动机被改变，那么这个类就具有多于一个的职责。而单一职责原则就是指一个类或者模块应该有且只有一个改变的原因。

好处：

- 1、低耦合性，影响范围小。
- 2、类复杂度降低，职责分明，提高了可读性。
- 3、职责单一，利于维护。

所以我的代码的总体设计遵循万物皆对象。并且每个对象只代表现实中的一个事物，也因此每个对象必须只具有一个职责。通俗来讲就是同一个对象不能既是... 也是...

3.2 里氏替换原则 Liskov Substitution Principle

定义 2 (里氏替换原则) 如果 S 是 T 的子类型, 对于 S 类型的任意对象, 如果将他们看作是 T 类型的对象, 则对象的行为也理应与期望的行为一致。

好处:

- 1、防止继承泛滥。
- 2、增强健壮性。
- 3、提高程序的维护性, 扩展性和兼容性。

另一种关于里氏替换原则的描述为 Robert Martin 在《敏捷软件开发: 原则、模式与实践》一书中对原论文的解读: 子类型 (subtype) 必须能够替换掉他们的基类型 (base type)。

具体而言, 此原则要求在代码开发过程中应遵循:

- * 子类可以实现父类的抽象方法, 但不能覆盖父类的非抽象方法;
- * 子类可以增加自己特有的方法;
- * 当子类的方法重载父类的方法时, 方法的形参要比父类方法的输入参数更佳宽松;
- * 当子类的方法实现父类的抽象方法时, 方法的返回值要比父类更加严格;

3.3 最少知识原则 The Least Knowledge Principle

定义 3 (最少知识原则) 又称迪米特法则, 一个类对于其他类知道的越少越好, 就是说一个对象应当对其他对象有尽可能少的了解, 只和朋友通信, 不和陌生人说话。

好处:

- 1、降低类之间的耦合度, 有助于提高类之间、模块之间的相对独立性。
- 2、利于代码的可重用性以及系统的拓展性。

具体而言, 此原则要求在代码开发过程中, 应尽可能减少类与类之间的关系, 实现低耦合度, 高内聚性。这个原则在软件工程中也被反复提及。

3.4 依赖倒置原则 Dependence Inversion Principle

定义 4 (依赖倒置原则) 依赖倒置原则是指程序要依赖于抽象接口, 不要依赖于具体实现。有利于降低客户与实现模块间的耦合。

好处:

- 1、稳定、灵活、健壮。

具体而言, 此原则要求在代码开发过程中应遵循:

- * 低层模块尽量都要有抽象类或者接口, 或者两者都有;
- * 变量的声明类型尽量是抽象类或者接口;
- * 使用继承时遵循里氏替换原则;

3.5 接口隔离原则 Interface Segregation Principle

定义 5 (接口隔离原则) 接口隔离原则是指客户端不应该依赖它不需要的接口。一个类对另一个类的依赖应该建立在最小的接口上。但同时需要保证接口的专业性，使用多个专门的接口比使用单一的总接口要好。

好处：

1. 降低接口粒度，提高接口灵活性，可维护性。
2. 提高系统内聚性，减少对外互动，降低系统的耦合性。

具体而言，此原则要求在代码开发过程中应遵循：

- * 一个接口只服务于一个子模块或业务逻辑，服务定制；
- * 通过业务逻辑压缩接口中的 public 方法，让接口看起来更加精悍；
- * 已经被污染了的接口，尽量修改，如果变更风险太大，则用适配器模式进行转化；

4 技术问题

4.1 多线程

线程需要处理的任务主要分为两种，周期性的任务，和需要一直运行的任务。在我的项目中这两种任务的代表分别是周期性刷新游戏界面，和需要一直运行的服务器。多线程根据管理方法也可分为线程池管理和非线程池管理。下面我举几个例子：

例 1 (线程池 + 周期性) 我的线程池中的线程用以进行所有 AI 敌人的行为 (移动、攻击等)，这两种方法可以实现，一种是轮询式，另一种是周期性进行移动操作，显然后一种符合现实情形而且也更加节省 CPU 压力。所以我的实现如下：

```
public class CreatureFactory {
    private World world;
    private int goblinSize;
    private ScheduledExecutorService goblinsExecutorService;

    public CreatureFactory(World world, int goblinSize) {
        this.world = world;
        this.goblinSize = goblinSize;
        this.goblinsExecutorService = Executors.newScheduledThreadPool(this.goblinSize);
    }

    public Goblin newGoblin() { // ... 创建一个妖精对象 goblin ... 妖精周期性进行移动
        this.goblinsExecutorService.scheduleAtFixedRate(
            new Runnable() {
                @Override
                public void run() {
                    goblin.moveOneStep();
                    goblin.seeAndAttack();
                }
            },
            0,
            1,
            TimeUnit.SECONDS);
    }
}
```

```

        }
    },
    goblin.moveSpeedInterval, goblin.moveSpeedInterval, TimeUnit.MILLISECONDS);
    return goblin;
}
}

```

需要注意的是,我在这使用的不是课上所提及的线程池,而是 `Java.util` 库封装好的 `ScheduledThreadPool` (Java 自带的库还是很不错的)

例 2 (独立线程 + 持续运行) 我的服务器借助 `main` 函数单独独立出一个服务器线程,同时此服务器线程是个死循环,一直在运行以处理客户端的请求:

```

public static void main(String[] args) {
    System.out.println("Server Started at port : " + SERVER_PORT);
    try {
        new Server().startReactor(SERVER_PORT);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

4.2 线程同步

多线程技术的使用所带来的最为经典的一个技术问题就是线程同步,由于 Java 提供了对于某些简单的线程同步问题的解决方法,所以 Java 解决简单线程同步还是比较方便的。但是如果同步问题变得复杂或者考虑线程同步的效率问题,用 Java 反而会变得复杂。

我的项目中所要解决的大多是比较简单的线程同步问题,主要使用了 `synchronized` 关键字和同步锁。

例 3 (`synchronized` 代码块) 我的项目中,在 `World` 类中有一个 `AttackDot` 列表,有两个线程会读写这个列表:

- 刷新 (refresh) 线程会周期性刷新界面,因此会读取攻击点列表;
- 主 (main) 线程,也叫玩家线程,会产生一系列攻击点并加入攻击点列表中,因此会写该攻击点列表;

所以在 `AttackDot` 列表的对外的公共读写接口中,我使用了 `synchronized` 代码块进行同步:

```

private List<AttackDot> attackDots;
public boolean addAttackDot(AttackDot attackDot){
    ...
    synchronized(this.attackDots){
        this.attackDots.add(attackDot);
    }
}

```

```
...
    return true;
}
public boolean addAttackDot(List<AttackDot> attackDots){
    for (AttackDot attackDot : attackDots) {
        this.addAttackDot(attackDot);
    }
    return true;
}
public void removeAttackDot(AttackDot target) {
    synchronized(this.attackDots){
        this.attackDots.remove(target);
    }
}
public List<AttackDot> getAttackDots(){
    synchronized(this.attackDots){
        return this.attackDots;
    }
}
```

例 4 (同步锁序列化) 资源冲突的经典问题在上课也提及到了，即一个方格块上只能有一个生物，但由于多线程的存在可能会出现两个生物同时踩在同一个方格块上的情况。所以我在自己的代码中使用同步锁进行序列化从而化解资源冲突：

具体而言，所有的生物共享一个代表移动权限的同步锁，这使得任何时刻只能有一个生物（即线程）进行移动

```
public class Creature {
    private static Lock moveLock = new ReentrantLock(); // 用于序列化移动行为的锁

    synchronized public void moveBy(int mx, int my) {
        ...
        Creature other = world.creature(x + mx, y + my);
        ...
        moveLock.lock(); // 要移动啦！锁上锁
        try {
            ai.onEnter(x + mx, y + my, world.tile(x + mx, y + my)); // 移动
        } finally{
            moveLock.unlock(); // 必须使用finally释放锁
        }
        ...
    }
}
```

例 5 (volatile) 相比于 synchronized (synchronized 通常称为重量级锁), volatile 更轻量级, 因为它不会引起线程上下文的切换和调度。但是 volatile 变量的同步性较差。

在此引入三个并发编程中的概念:

定义 6 (原子性) 一个操作或者多个操作要么全部执行并且执行的过程不会被任何因素打断, 要么就都不执行。

定义 7 (可见性) 指当多个线程访问同一个变量时, 一个线程修改了这个变量的值, 其他线程能够立即看得到修改的值。

定义 8 (有序性) 程序执行的顺序按照代码的先后顺序执行。

下面是 JVM 的内存模型, 以及《计算机系统基础》中的缓存模型:

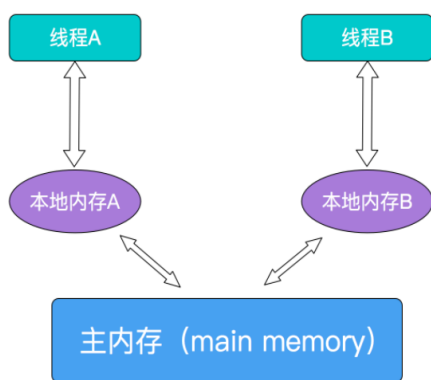


图 4 JVM 内存模型
Figure 4 JVM Memory Module

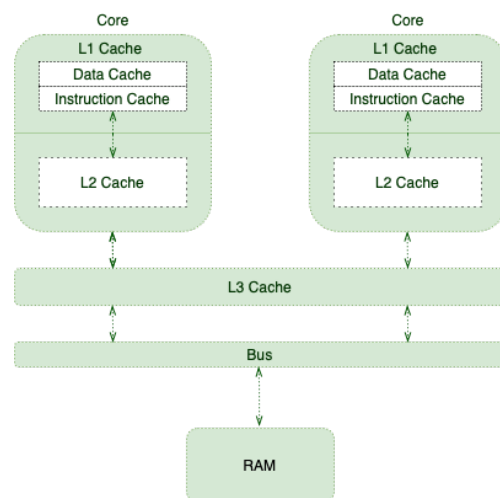


图 5 CPU 缓存内存模型
Figure 5 CPU Cache Memory Module

而 volatile 保证可见性, 不保证原子性, 同时禁止指令重排。因此 volatile 不适合符合操作。关于 volatile 的内容可查看这篇 blog: 《Guide to the Volatile Keyword in Java》

在我的代码中, 我将多线程操作的 boolean 变量用 volatile 进行修饰, 这是由于 boolean 的操作本身就是原子性的, 正好弥补 volatile 无法保证原子性的特征。

4.3 运行时类型识别

定义 9 (运行时类型识别) 运行时类型识别 (Run-Time Type Identification, RTTI) 是一些编程语言在运行时公开对象数据类型信息的特性。是更一般的概念 (称为类型自省) 的特化。

例 6 (instanceof) 在我的项目中, 武器分为三类, 近战类武器、远程类武器、投掷类武器 (灵感来源于荒野乱斗)。所以玩家的显示界面需要根据当前武器的类型来显示具体的不同的信息, 于是我使用了 Java 的 RTTI 特性, 具体而言就是 instanceof 关键字:

```
// 目前的武器
```

```
Weapon curWeapon = this.player.weaponFactory().curWeapon();
String weaponStats = "current weapon: " + curWeapon.name();
if(curWeapon instanceof Gun){ // 当前武器是枪，则显示剩余子弹
    Gun curGun = (Gun)curWeapon;
    weaponStats += " %d/%d bullets";
    weaponStats = String.format(weaponStats, curGun.remainingBullet(),
        curGun.maxBulletAmount());
}
terminal.write(weaponStats,1, 30);
```

4.4 多态与接口

接口表示方法的集合，和多态有着密切的关系，代表着现实世界中功能的集合

多态是这学期 Java 课的一个非常重要的内容。简单来说，不同于 C++，Java 是一个天然动态绑定的语言，这种多态性也十分方便了我的项目的开发。

我的项目的运行逻辑主要包含三个模块：人物模块、武器模块、物品模块。下面我就以武器模块为例，来介绍多态性在我的项目中的应用：

例 7 (Weapon) - 武器的抽象基类叫做 Weapon，它包含了武器的基本属性：名称、攻击力、攻击范围、冷却间隔、武器工厂等。实现了武器的部分基本的功能：

```
public abstract class Weapon implements Attack{
    protected String name;// 武器的名称
    protected int attackValue; // 武器的攻击力，远程武器则表示子弹的攻击力
    protected int range;// 武器的范围，近战武器表示攻击范围，远程武器代表射程
    protected int coldingInterval;// 武器的冷却时间，单位为毫秒
    protected boolean active;// 武器可用性
    protected WeaponFactory factory;
    protected Timer refreshTimer;

    public Weapon(WeaponFactory factory, String name, int attackValue, int range, int
        coldingInterval){
        ...
    }
}
```

- 而 Weapon 类实现了接口 Attack，表示武器的攻击方式，不同的子类会对该接口进行不同的实现，从而产生不同的攻击行为（射子弹、扔炸弹、抡球棍等）

```
public interface Attack {
    public boolean attack();
}
```

- 三类武器会继承抽象基类 `Weapon` 并进行不同的接口实现, 下面我以枪械类武器 `Gun` 和狙击枪 `SniperGun` 来举个例子:

```
public abstract class Gun extends Weapon{
    protected int remainingBullet; // 剩余子弹个数
    protected int addBulletInterval; // 添加子弹的间隔时间, 单位毫秒
    protected int maxBulletAmount; // 弹匣容量
    protected Timer addBulletTimer; // 添加子弹的计时器
    protected Bullet bulletSample; // 子弹样板, 用于子弹的复制
    public Gun(WeaponFactory factory, String name, int attackValue, int range, int
        coldingInterval,
            int addBulletInterval, int maxBulletAmount, Bullet bulletSample){
        // 初始化
        // 周期性添加子弹
    };

    public class SniperGun extends Gun{
        public SniperGun(WeaponFactory factory, Bullet bulletSample){
            // 初始化
        }

        @Override
        public boolean attack(){ /* 射出狙击子弹 */}
    }
}
```

4.5 泛型

在 Java 中, 泛型也是多态的另一个具体实现机制。课上也着重强调了, 和 C++ 不同的是, Java 的泛型是使用类型擦除实现的, 这在代码编写的过程中需要时刻牢记。

例 8 (ByteToObject) 在我的项目中, 服务器和客户端之间需要传送对象, 因此需要一个工具类将某个对象转换成比特数组用以发送数据, 同时需要将某个数据流比特数组转换成某个对象。由于转换方法需要目标对象的类型参数, 所以用泛型来实现:

```
public class ByteArrayUtils {
    public static<T> Optional<byte[]> objectToBytes(T obj){
        byte[] bytes = null;
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        ObjectOutputStream sOut;
        try {
            sOut = new ObjectOutputStream(out);
            sOut.writeObject(obj);
            sOut.flush();
            bytes= out.toByteArray();
        }
    }
}
```

```
    } catch (IOException e) {
        e.printStackTrace();
    }
    return Optional.ofNullable(bytes);
}

public static<T> Optional<T> bytesToObject(byte[] bytes) {
    T t = null;
    ByteArrayInputStream in = new ByteArrayInputStream(bytes);
    ObjectInputStream sIn;
    try {
        sIn = new ObjectInputStream(in);
        t = (T)sIn.readObject();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return Optional.ofNullable(t);
}
}
```

4.6 内部类

内部类也是 Java 的一个小特性，即可以在一个类内部定义一个类。而在我的项目中，我主要利用它来定义一些较为简单的，且和某些类关系密切的类，从而减少源文件的数量，例子如下：

例 9 (DoorPairs) 在我的项目中，我实现了传送门对，而一个传送门对 (DoorPair) 对象包含两个关联的传送门 (Door) 对象，两个类密切相关，所以我使用内部类来实现 Door 类：

```
public class DoorPairs{ // 传送门对
    private int width,height;
    private List<Pair<Door,Door>> doorPairs;

    public DoorPairs(int width, int height){
        this.doorPairs = new ArrayList<>();
        this.width = width;
        this.height = height;
    }
    public boolean addPairs(int x1, int y1, int x2, int y2){
        ...
    }

    public DoorPairs.Door search(int x, int y){
        ...
    }
}
```

```

    }

    public class Door{
        private int x, y;

        public Door(int x, int y){
            this.x = x;
            this.y = y;
        }

        public boolean equal(int x, int y){
            return this.x == x && this.y == y;
        }
    }
}

```

4.7 序列化

定义 10 (序列化) 序列化 (Serialization) 是将对象的状态信息转换为可以存储或传输的形式化的过程。

Java 实现序列化的方法是将某个类标记为 `Serializable`, 同时可以用 `transient` 关键字标记类中的某个域被排除在序列化范围外。最后使用 `ObjectOutputStream` 类的 `writeObject` 方法将某个对象写入到流中。

由于序列化的过程是递归的, 也就是说, 只要相关的类都标记为 `Serializable`, 那么就可以将所有相关类都保存。因此可以利用这种特性实现深拷贝功能。例子如下:

例 10 (Recorder) 在我的项目中, 我需要一个记录器来录制游戏过程。这就意味着 `Recorder` 类需要将游戏刷新的每一帧的内容都深拷贝并保存到文件中。而这就可以使用序列化来实现:

```

public class Recorder implements Serializable{
    private List<PlayScreen> playScreens;

    public void saveSnapshot(PlayScreen snapshotPlayScreen){
        try {
            this.playScreens.add(Recorder.clone(snapshotPlayScreen));
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}

public static <T extends Serializable> T clone(T obj)
throws IOException, ClassNotFoundException {

```

```
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bout);
        oos.writeObject(obj);

        ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bin);
        return (T) ois.readObject();
    }
}
```

4.8 输入输出

定义 11 (输入输出) 在计算中, 输入/输出是信息处理系统 (如计算机) 与外部世界之间的通信。

相比于其它语言, Java 进行输入输出主要是借助各种封装好的包。包括基础的输入输出、基于字符的输入输出、基于比特的输入输出、基于对象的输入输出, 以及具有功能的输入输出, 例如具有缓冲区的输入输出等。就 Java 丰富的包而言, Java 的输入输出的基础功能是很完备的。

但是, 实际上 Java 只对 I/O 做了一层很薄的封装, 这导致了对于复杂需求的输入输出系统, Java 就会产生很复杂的框架代码。

在我的项目中, 我对于 I/O 的处理是很粗糙的, 基本只采用了很基础的 I/O 框架。主要包括三个方面: 用文件流来将记录存储在本地文件中; 用 NIO+Reactor 来处理网络通信; 用对象流来传输对象;

4.9 网络通信

定义 12 (网络通信) 网络通信是通过网络将各个孤立的设备进行连接, 通过信息交换实现人与人, 人与计算机, 计算机与计算机之间的通信。

在网络通信中最重要的就是网络通信协议。网络是分层的, 链路层、网络层、传输层、应用层都有自己的通信协议。

在我的项目中, 我想要实现服务器-客户机之间的游戏同步, 简言之就是服务器需要处理客户机发来的消息并进行响应, 因此需要对消息进行协议规定。而最简单的协议规定方式就是通过定义一个消息类:

例 11 (Message)

```
public class Message implements Serializable{
    private PlayScreen playScreen;
    private String playerName;
    private int playerId;
    private int roomId;
    private String info;

    @Override
```

```

public String toString(){
    return this.info;
}

public static Message dataToObj(byte[] data){
    Message message = null;
    Object obj = ByteArrayUtils.bytesToObject(data).get();
    if(obj instanceof Message)
        message = (Message)obj;
    return message;
}
...
}

```

由上可见, 我的消息类包含了房间、玩家姓名、玩家序号、房间号和消息内容, 这样服务器和客户端就可以根据这些内容来进行解析处理:

例 12 (Server)

```

public class WriteEventHandler implements EventHandler {
    private Server server; //服务器

    public void handleEvent(SelectionKey handle) throws Exception {
        // 获取客户端发来的Message: messageFromClient
        this.parseMessage(messageFromClient, outputBuffer);
        socketChannel.write(outputBuffer);
        socketChannel.close();
    }

    private void parseMessage(Message messageFromClient, ByteBuffer outputBuffer){
        switch (messageFromClient.info()) {
            case Message.CONNECT_MESSAGE:
                this.handleConnectMessage(messageFromClient, outputBuffer);
                break;
            case Message.ADD_ROOM_MESSAGE:
                this.handleAddRoomMessage(messageFromClient, outputBuffer);
                break;
            .....
            default:
                break;
        }
    }
}

```

5 工程问题

5.1 设计模式

设计模式是众多程序员在软件开发实践中总结出来的一些代码设计结构，从而可以直接拿来解决一些特定的问题。通常这些设计模式是被实践证明十分适合于特定问题的，所以应当在项目开发中主动的多使用这些设计模式。

关于设计模式推荐 Alexander Shvets 的《深入设计模式》这本书。目前这本书最新版于 2022.1.29 发布。相比于网上找的零碎的、片面的、复杂的教程，这本书图文结合、清晰明了。包括了我上面提到的 SOLID 软件设计原则以及许多设计模式。

我读了前面一部分，感觉十分适合初学者理解，所以安利一波：



图 6 深入理解设计模式

Figure 6 Deep understanding of design patterns

令我感到十分遗憾的是，由于设计模式这个知识点的课程安排比较靠后，当我学习完设计模式后，整个项目的框架已经大致搭建完毕了，回过头来才发现很多地方都可以使用合适的设计模式进行优化（下面我会举例子来对我之前的代码进行批判）。所以在项目开发的后期我才开始主动使用设计模式。

5.1.1 工厂方法模式

定义 13 (工厂方法) 工厂方法是一种创建型设计模式，其在父类中提供一个创建对象的方法，允许子类决定实例化对象的类型。

工厂方法模式建议使用特殊的工厂方法代替对于对象构造函数的直接调用（即使用 `new` 运算符）。工厂方法返回的对象通常被称作“产品”。

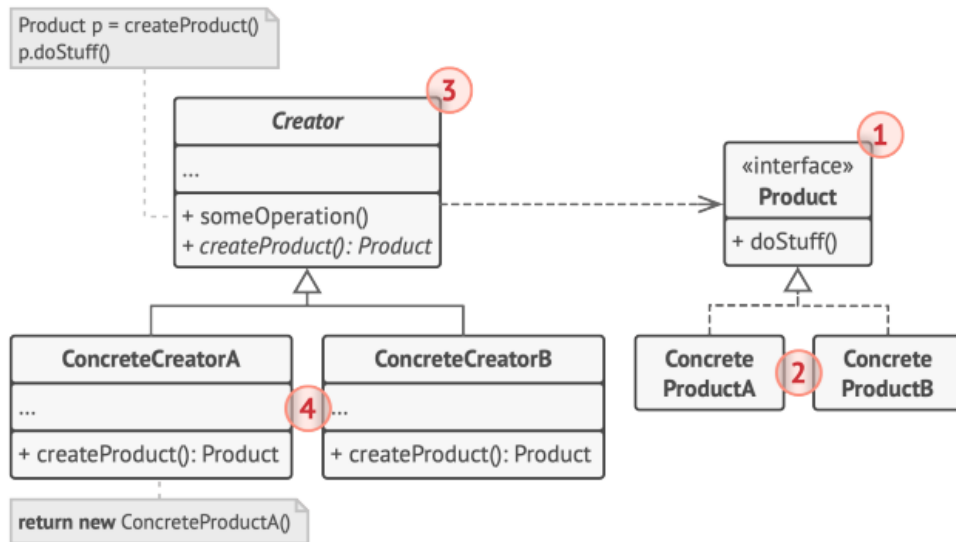


图 7 工厂设计模式

Figure 7 Factory design model

具体实现方式可参考以下方法：

- (1) 让所有产品都遵循同一接口。该接口必须声明对所有产品都有意义的方法。
- (2) 在创建类中添加一个空的工厂方法。该方法的返回类型必须遵循通用的产品接口。
- (3) 在创建者代码中找到对于产品构造函数的所有引用。将它们依次替换为对于工厂方法的调用，同时将创建产品的代码移入工厂方法。
- (4) 如果应用中的产品类型太多，那么为每个产品创建子类并无太大必要，这时你也可以在子类中复用基类中的控制参数。
- (5) 代码经过上述移动后，基础工厂方法中已经没有任何代码，可以将其转变为抽象类。

工厂设计模式的优缺点：

- √ 可以避免创建者和具体产品之间的紧密耦合。
- √ 单一职责原则。可以将产品创建代码放在程序的单一位置，从而使代码更容易维护。
- √ 开闭原则。无需更改现有客户端代码，你就可以在程序中引入新的产品类型。
- × 工厂方法模式需要引入许多新的子类，代码可能会因此变得更复杂。

例 13 (CreatureFactory) 我的项目中有三类生物：葫芦娃玩家、妖精 AI 和苹果。三类生物都继承自基类 `Creature`。所以我使用 `CreatureFactory` 来产生不同的生物：

```

public class CreatureFactory{
    private World world;
    private int goblinSize;
    private ScheduledExecutorService goblinsExecutorService;

```

```
/**
 * 产生玩家葫芦娃
 * @param messages
 * @param playerName
 * @return
 */
public Player newPlayer(List<String> messages, String playerName) {
    ...
}

/**
 * 产生苹果
 * @return
 */
public Creature newApples() {
    ...
}

/**
 * 随机产生妖精
 * @return
 */
public Goblin newRandomGoblin(){
    ...
}

public void enableGoblinsStartMove(){
    for(Goblin goblin : this.world.getGoblins()){
        goblinStartMove(goblin);
    }
}

// 妖精周期性进行移动和攻击
private void goblinStartMove(Goblin goblin){
    this.goblinsExecutorService.scheduleAtFixedRate(
        new Runnable() {
            @Override
            public void run() {
                if(goblin.hp() < 0)
                    return;
                if(world.worldIsPaused())
                    return;
                goblin.moveOneStep();
                goblin.seeAndAttack();
            }
        },
        goblin.moveSpeedInterval, goblin.moveSpeedInterval, TimeUnit.MILLISECONDS);
}
```

```

    }
}

```

需要注意的是,由于我的项目结构比较简单,所以设计工作的初期会使用工厂方法(较为简单,而且可以更方便地通过子类进行定制)。如果以后项目继续开发扩展,生物越来越多,工厂方法会演化为使用抽象工厂、原型或生成器(更灵活但更加复杂)。

5.1.2 代理模式

定义 14 (代理模式) 代理是一种结构型设计模式,让你能够提供对象的替代品 或其占位符。代理控制着对于原对象的访问,并允许在将请求提交给对象前后进行一些处理。

代理模式建议新建一个与原服务对象接口相同的代理类,然后更新应用以将代理对象传递给所有原始对象客户端。代理类接收到客户端请求后会创建实际的服务对象,并将所有工作委派给它。

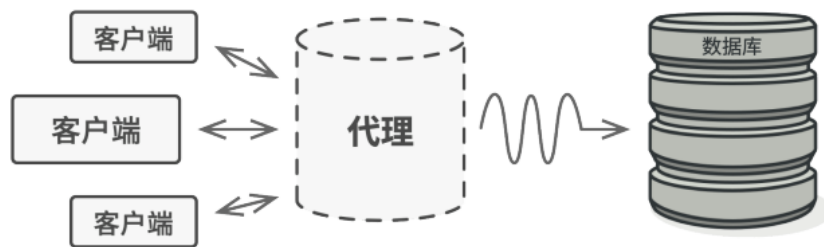


图 8 代理模式
Figure 8 Proxy design model

代理模式的优缺点:

- ✓ 你可以在客户端毫无察觉的情况下控制服务对象。
- ✓ 如果客户端对服务对象的生命周期没有特殊要求,你可以对生命周期进行管理。
- ✓ 即使服务对象还未准备好或不存在,代理也可以正常工作。
- ✓ 开闭原则。你可以在不对服务或客户端做出修改的情况下创建新代理。
- × 代码可能会变得复杂,因为需要新建许多类。
- × 服务响应可能会延迟。

例 14 (Client) 代理模式经常用于本地执行远程服务(远程代理)。适用于服务对象位于远程服务器上的情形。

我的项目中,每个玩家需要连接到服务器上的同一个房间,并在服务器的房间中进行战斗。因此玩家需要在本地的房间中发送操作信息给服务器,同时需要周期性从服务器抓取当前房间信息到本地房间中更新。这种复杂的消息传送就可以使用代理模式来实现。

```

// 玩家的本地房间
public class ClientPlayScreen implements Screen{
    ...
    private Client client; // 代理发送消息的客户端
}

```

```
private PlayScreen playScreen; // 本地房间，抓取远程服务器房间进行更新
...

// 响应玩家的操作，向服务器发送操作消息
@Override
public Screen respondToUserInput(KeyEvent key){
    ...
    switch (key.getKeyCode()) {
        case KeyEvent.VK_LEFT:
        case KeyEvent.VK_RIGHT:
        case KeyEvent.VK_UP:
        case KeyEvent.VK_DOWN:
        case KeyEvent.VK_SPACE:
        case KeyEvent.VK_X:
        case KeyEvent.VK_Z:
        case KeyEvent.VK_C:
        case KeyEvent.VK_W:
        case KeyEvent.VK_S:
        case KeyEvent.VK_A:
        case KeyEvent.VK_D:
            this.client.operation(key);
            break;
        ...
    }
    return this;
}
}
```

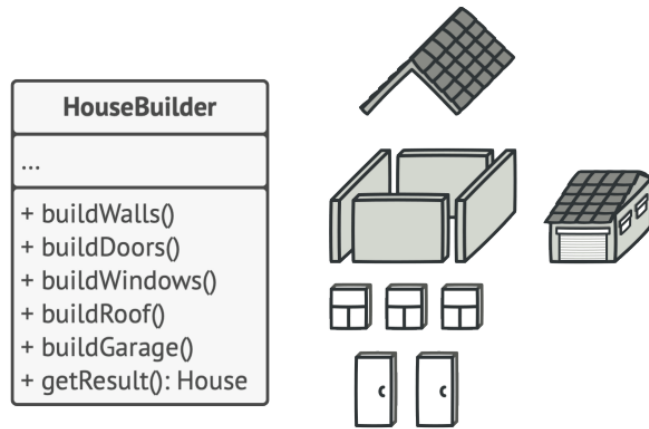
这样客户端所有的与服务器的交互都可以交给代理类 Client，包括等待服务器响应的时延、缓存等操作。而两者交互时发生的改动（例如发送消息的格式）都不会影响到本地的框架代码（只要 Client 的接口不变）

代理模式的这种优势是非常令我感同身受的，因为一开始我编写服务器-客户端的交互时，并没有封装代理类 Client，而且边写网络交互的代码，边写对应的单元测试代码。这导致每次要对网络交互框架进行改动时，都要相对应的修改单元测试代码，非常的麻烦。后来采用代理模式进行隔离后才优化了代码结构。

5.1.3 生成器模式

定义 15 (生成器模式) 生成器是一种创建型设计模式，使你能够分步骤创建复杂对象。该模式允许你使用相同的创建。代码生成不同类型和形式的对象。

生成器模式建议将对象构造代码从产品类中抽取出来，并将其放在一个名为生成器的独立对象中。



生成器模式让你能够分步骤创建复杂对象。生成器不允许其他对象访问正在创建中的产品。

图 9 生成器模式

Figure 9 Generator design model

生成器模式的优缺点：

- √ 可以分步创建对象，暂缓创建步骤或递归运行创建步骤。
- √ 生成不同形式的产品时，你可以复用相同的制造代码。
- √ 单一职责原则。你可以将复杂构造代码从产品的业务逻辑中分离出来。
- × 由于该模式需要新增多个类，因此代码整体复杂程度会有所增加。

例 15 (Creature 构造函数 (批判)) 在我的项目中，生物会具有非常多的属性（血量、护甲量、形象、颜色、所处世界等等），所以在项目开发早期没有学习过生成器模式的情况下，我采用了具有很多参数的构造函数，导致代码臃肿：

```
public Creature(World world, char glyph, Color color, int maxHP,
               int attack, int defense, int visionRadius, int moveSpeedInterval) {
    this.world = world;
    this.glyph = glyph;
    this.color = color;
    this.maxHP = maxHP;
    this.hp = maxHP;
    this.attackValue = attack;
    this.defenseValue = defense;
    this.visionRadius = visionRadius;
    this.bomb = new Bomb(world, this, this.x(), this.y(), Tile.BOMB.glyph(), color, 100);
    this.direction = Direction.Right;
    this.weaponFactory = new WeaponFactory(this);
    this.moveSpeedInterval = moveSpeedInterval;
}
```

```
this.moveTimer = null;
this.moveActive = true;
this.visible = true; // 生物一开始是可见的

this.startMove();
}
```

而早期我的解决方法也十分朴素，继承 Creature 基类，并在子类的构造函数中简化，将相关参数写死在代码中：

```
public class FirstBro extends Player{
    public static final char GLYPH = Tile.FIRST_BRO.glyph();
    public static final Color COLOR = Tile.FIRST_BRO.color();
    public static final int MAX_HP = 300;
    public static final int ATTACK_VALUE = 20;
    public static final int DEFENSE = 10;
    public static final int VISION_RADIUS = 9;
    public static final int MOVE_SPEED_INTERVAL = 500;

    public FirstBro(World world){
        super(world, FirstBro.GLYPH, FirstBro.COLOR, FirstBro.MAX_HP, FirstBro.ATTACK_VALUE,
            FirstBro.DEFENSE, FirstBro.VISION_RADIUS, FirstBro.MOVE_SPEED_INTERVAL);
        this.weaponFactory.newBaseballBat();
        this.weaponFactory.newBombGlove();
        this.weaponFactory.newTorch();
    }
}
```

这种方法无疑是非常丑陋的，而且降低了生物构造的灵活性。

所以应当修改为以下的生成器结构：

5.1.4 状态模式

定义 16 (状态模式) 状态是一种行为设计模式，让你能在一个对象的内部状态变化时改变其行为，使其看上去就像改变了自身所属的类一样。

状态模式的优缺点：

- ✓ 单一职责原则。将与特定状态相关的代码放在单独的类中。
- ✓ 开闭原则。无需修改已有状态类和上下文就能引入新状态。
- ✓ 通过消除臃肿的状态机条件语句简化上下文代码。
- × 如果状态机只有很少的几个状态，或者很少发生改变，那么应用该模式可能会显得小题大作。

在说明状态模式之前，需要提及一个至关重要的概念——有限状态机

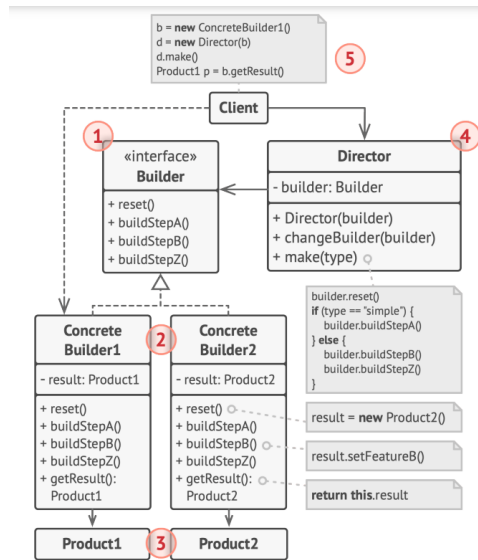


图 10 生成器模式结构

Figure 10 Generator design model structure

其主要思想是程序在任意时刻仅可处于几种有限的状态中。在任何一个特定状态中，程序的行为都不相同，且可瞬间从一个状态切换到另一个状态。不过，根据当前状态，程序可能会切换到另外一种状态，也可能会保持当前状态不变。这些数量有限且预先定义的状态切换规则被称为转移。

这种有限状态机的开发思想广泛出现在早期软件的源代码中，其经典的实现代码如下：

```
class Document is
    field state: string
    // .....
    method publish() is
        switch (state)
            "draft":
                state = "moderation"
                break
            "moderation":
                if (currentUser.role == "admin")
                    state = "published"
                    break
            "published":
                // 什么也不做。
                break
        // .....
```

而这种基于条件语句的状态机也出现在我的项目中：

例 16 (StartScreen(批判)) 在我的项目中，在正式开始游戏之前玩家需要进行一系列的选择

(游戏角色选择、地图选择、BGM 选择等)，而且这种选择是可以返回/继续的。也就是说，玩家会依次进入不同的选择状态，不同的状态对应不同的选择对象，因此我采用了基于条件语句的状态机：

```
public class StartScreen extends RestartScreen {
    ...
    @Override
    public Screen respondToUserInput(KeyEvent key) {
        ...
        switch (this.stage) {
            case 0:
                selectBranches(key);
                return this;
            case 1:// 选择房间地址/选择快照/选择并查看记录
                if(branchIndex == 0) return this.chooseRoomAddr(key);
                else if(branchIndex == 1) return this.chooseSnapShot(key);
                else if(branchIndex == 2) return this.chooseRecord(key);
            case 2:// 选择人物/确认加载游戏
                ...
            case 3:// 选择音乐
                if(branchIndex == 0) return this.chooseMusic(key);
            case 4:
                if(branchIndex == 0) return this.chooseMap(key);
            case 5:// 开始游戏
                ...
        }
    }
}
```

但是，随着项目的开发，我逐步在此类中添加更多状态和依赖于状态的行为，基于条件语句的状态机便暴露出其最大的弱点：为了能根据当前状态选择完成相应行为的方法，绝大部分方法中会包含复杂的条件语句。修改其转换逻辑可能会涉及到修改所有方法中的状态条件语句，导致代码的维护工作非常艰难。

毫无疑问的是，这个问题会随着项目进行变得越发严重。我们很难在设计阶段预测到所有可能的状态和转换。随着时间推移，最初仅包含有限条件语句的简洁状态机可能会变成臃肿的一团乱麻。

原始对象被称为上下文 (context)，它会保存一个指向表示当前状态的状态对象的引用，且将所有与状态相关的工作委派给该对象。

切换状态就是将当前活动的状态对象替换为另外一个代表新状态的对象。而这要求所有状态类都必须遵循同样的接口，而且上下文必须仅通过接口与这些对象进行交互。

因此更好的设计模式是将原来的基于条件语句的状态机重构成下面的结构：

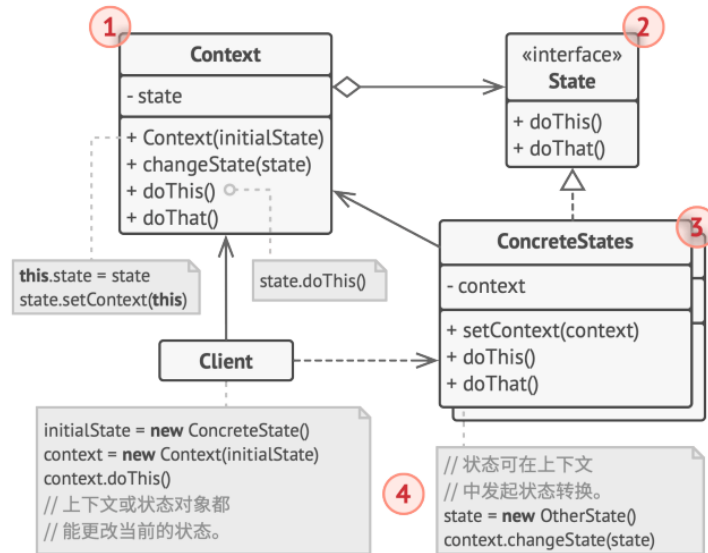


图 11 状态模式结构

Figure 11 Generator design model structure

5.2 自动构建

定义 17 (构建自动化) 构建自动化是将软件构建的创建和相关过程自动化的过程, 包括: 将计算机源代码编译成二进制代码, 打包二进制代码, 以及运行自动化测试。

对于 C/C++ 源代码来说, 我们需要在 Linux 系统上安装编译工具 (如 gcc/g++)、链接工具 (ld 等), 然后将每个源代码文件依次编译、链接最终生成可执行文件。最终将开发好的可执行文件上传到互联网上。

这种原始的开发方法无疑是非常繁琐的, 从源代码开发到最终产品的部署的每个步骤都需要人工重复地劳作。于是产生了一系列构建工具: Make、cMake、Gradle 等。

对于 Java 语言来说, 主流的自动构建工具包括 Maven、Ant、Gradle。我的项目使用 Maven 进行构建, 主要使用两个功能: 进行外部依赖包的管理, 以及对单元测试覆盖率进行限定:

例 17 (外部依赖包) 由于 Java 语言众多的工具包, Java 项目会使用到很多其它开发者封装好的包。正常使用包的步骤是将包下载到本地, 并将包所在的路径加入 ClassPath。而使用 Maven 只需要将包的 dependency 加入 pom.xml 即可, Maven 会自动下载和管理。

```

<dependencies>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
<!-- https://mvnrepository.com/artifact/org.javatuples/javatuples -->

```

```
<dependency>
  <groupId>org.javatuples</groupId>
  <artifactId>javatuples</artifactId>
  <version>1.2</version>
</dependency>
</dependencies>
```

我的项目仅使用了两个外部包，一个是用于单元测试的 junit，另一个是用于传送门对的 javatuples；

例 18 (覆盖率限定) 在我的项目中，框架代码 AsciiPanel 是其它开发者封装好发布到网上的，我将源代码下载下来并添加到我的项目中。所以这部分代码应当不包括在我的单元测试的范围内，所以我们可以借助于 Maven，将其排除在测试覆盖率的统计之外。

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.7</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>test</phase>
      <goals>
        <goal>report</goal>
      </goals>
      <configuration>
        <outputDirectory>target/site/jacoco</outputDirectory>
        <dataFile>${project.build.directory}/jacoco.exec</dataFile>
        <formats>XML</formats>
      </configuration>
    </execution>
  </executions>
  <configuration>
    <excludes>
      <exclude>cn/edu/nju/cs/game/asciiPanel/*</exclude>
    </excludes>
  </configuration>
</plugin>
```

我使用相关插件是 jacoco，在配置选项中将 cn/edu/nju/cs/game/asciiPanel/* 排除在外。

6 课程感言

6.1 我的收获

一开始选课时听说这个课要写游戏,课程有点难。我一开始选这个课是为了学习 Java 语言,让自己多一门编程语言;同时也是被开发游戏吸引。

因为大一大二我的编程经验大多局限于 OJ 和小项目(代码量低于 2000 行),所谓的大项目也是在老师搭建好的框架代码中“完形填空”,并没有真正从零开始逐渐开发出一个比较完善的项目。我想着既然选了软件工程方向,就应当将整个软件开发流程大致地走一走,多写代码。于是抱着写代码的心态选了课。

然而在学习的过程中,我逐渐发现这门课的精髓其实并不在于 Java 语法的介绍,更多的是从 Java 引申开来的额外知识以及编程思想。

一开始我学习这门课是将《On Java 8》粗略读一遍,然后就去看那种“几小时精通 Java,月入过万”的网课。然而随着课程讲解的深入,我发现有更多不属于 Java 的内容需要我去学习,老师推荐了很多资料的网址让我们去课后阅读。

第一次意识到这一点是在上“类加载和自省”这一节中,老师介绍了 JVM 语言,了解了 Java 虚拟机不仅仅是可以运行 Java 语言的,而且老师推荐阅读《JVM Internals》。读完以后我产生了点兴趣,于是去粗略地看了有关 JVM 的网课(当然是那种很浅的培训机构的网课)。但是即使是入门的课也让我收获颇丰,了解了 JVM 的内存模型、垃圾回收算法、Java 的动态绑定机制在 JVM 层面是如何实现的、双亲委派机制在源代码层面是如何实现的等等。

后来在学习“设计模式”时,我感觉除了 OOP,这是我这节课收获的最重要的编程思想,这节课让我收获了很多项目开发过程中的设计模式(但是老师推荐的“漫画书”是真的贵)。有点可惜的是很多设计模式我没用到我的项目中。

当然这节课第一个重点就是深入理解面向对象编程思想,这个词我这学期不仅在这门课,在隔壁的软件工程、高级程序设计也听老师反复强调了很多次。这当然是我的项目开发过程中基础中的基础。

还有一个意外收获是多媒体编码技术,我在完成图片隐写类的作业时,了解了图片隐写技术的最简单方法。还顺带学习了图像的编码方式,这个作业还让我探索如何将代码隐写到音频文件中,为此我还去了解声音是如何编码的,但最终还是搞不懂。听说下学期有个选修课是唐杰老师的《多媒体技术》,有空的话可以选来学学。

6.2 我的建议

第一个建议是关于课程内容,我觉得可以将“设计模式”这一节放到期末大项目刚开始发布的时候讲,这样在项目框架构建的初期很多有益于项目后续开发的设计模式就会被我们主动应用了。或者在期末大项目发布时老师将这一节的 Slide 公布,提醒我们在忙着开始写代码之前,先去预习一下这一节。

第二个建议也是关于课程内容,我觉得可以将“单元测试”这一节也往前放放(其实也可以不往前放)。主要是老师上课提到了“面向测试驱动的开发”,即在开发项目之前,先将描述功能的测试

代码写好，然后就像写 OJ 一样开发项目代码。由于我开始写单元测试时，大部分功能的代码已经大致搭起来了，所以我写的单元测试有点为了写单元测试而写单元测试的味道，并没有起到检测项目功能是否完成的作用。

6.3 我的总结

《Java 高级程序设计》牛皮