软件体系结构作业 11

姓　名：洪伟鑫

专　业：软件工程

年　级：2022 级

学　号：37220222203612

2025 年 4 月 20 日

1、请举例说明克隆模式的其他应用。

场景描述： 在游戏中，可能存在不同类型角色的基础模板（例如：战士、法师、弓箭手）。创建一个新角色可能涉及加载资源（模型、纹理）、设置基础属性、分配默认技能等，这些操作可能非常耗费资源。使用原型模式允许我们创建预先配置好的模板，并在需要生成新实例时快速克隆它们，之后可以对克隆体进行少量修改。

```java
// 代表一个可变的技能对象
class Skill implements Cloneable {
    private String name; // 技能名称
    private int level; // 技能等级

    public Skill(String name, int level) {
        this.name = name;
        this.level = level;
    }

    // 设置器和获取器
    public void setLevel(int level) {
        this.level = level;
    }

    public String getName() {
        return name;
    }

    public int getLevel() {
        return level;
    }

    @Override
    public String toString() {
        return "技能 [名称=" + name + ", 等级=" + level + "]";
    }

    @Override
    public Skill clone() throws CloneNotSupportedException {
        // Skill 类只有基本类型和不可变类型(String)成员，浅拷贝足够
        System.out.println("正在克隆技能: " + name);
        return (Skill) super.clone();
    }
}
```

```java
// 基础游戏角色类，实现 Cloneable 接口
abstract class GameCharacter implements Cloneable {
    protected String type; // 角色类型
    protected int health; // 生命值
    protected int mana; // 法力值
    protected List<Skill> skills; // 可变成员变量 - 需要深拷贝

    public GameCharacter() {
        this.skills = new ArrayList<>(); // 初始化技能列表
    }

    // 抽象方法，用于显示角色信息
    public abstract void display();

    // 克隆后用于修改状态的设置器
    public void setHealth(int health) {
        this.health = health;
    }

    public void setMana(int mana) {
        this.mana = mana;
    }

    public List<Skill> getSkills() {
        return skills;
    }

    @Override
    public GameCharacter clone() throws CloneNotSupportedException {
        System.out.println("正在克隆游戏角色，类型: " + type);
        // 1. 调用 super.clone() 执行浅拷贝
        GameCharacter clonedCharacter = (GameCharacter) super.clone();

        // 2. 对可变成员变量 'skills' 执行深拷贝
        clonedCharacter.skills = new ArrayList<>(); // 为克隆体创建一个全新的 List 实例
        for (Skill skill : this.skills) {
            // 克隆列表中的每一个 Skill 对象，并将克隆出的 Skill 添加到新列表中
            clonedCharacter.skills.add(skill.clone());
        }

        return clonedCharacter;
    }
```

```java
// 具体的战士原型类
class Warrior extends GameCharacter {
    private String weapon; // 武器 (String 是不可变的)

    public Warrior(String weapon) {
        this.type = "战士";
        this.health = 150;
        this.mana = 20;
        this.weapon = weapon;
        // 添加默认技能
        this.skills.add(new Skill(name:"冲锋", level:1));
        this.skills.add(new Skill(name:"斩击", level:1));
        System.out.println("战士原型已创建，武器: " + weapon);
    }

    // 设置武器
    public void setWeapon(String weapon) {
        this.weapon = weapon;
    }

    @Override
    public void display() {
        System.out.println("--- " + type + " ---");
        System.out.println("生命值: " + health + ", 法力值: " + mana);
        System.out.println("武器: " + weapon);
        System.out.println("技能: " + skills);
        System.out.println(x:"--------------------");
    }

    @Override
    public GameCharacter clone() throws CloneNotSupportedException {
        // 调用父类的 clone 方法处理基础字段和技能列表的深拷贝
        Warrior clonedWarrior = (Warrior) super.clone();
        // 'weapon' 是 String 类型（不可变），所以父类 clone() 中的浅拷贝就足够了。
        // 如果 weapon 是一个可变对象，则需要在这里进行深拷贝。
        System.out.println(x:"战士特定的克隆逻辑完成。");
        return clonedWarrior;
    }
}
```

```java
public class GameDemo {
    Run | Debug
    public static void main(String[] args) {
        CharacterRegistry registry = new CharacterRegistry();

        // 1. 创建并注册原型（这步可能是昂贵的初始化过程）
        System.out.println(x:"=== 正在创建原型 ===");
        Warrior baseWarrior = new Warrior(weapon:"基础剑");
        registry.addPrototype(key:"WARRIOR_SWORD", baseWarrior);
        System.out.println(x:"==========================\n");

        try {
            System.out.println(x:"=== 从原型生成角色 ===");

            // 2. 生成一个标准的战士
            GameCharacter warrior1 = registry.getClone(key:"WARRIOR_SWORD");
            System.out.println(x:"已生成战士 1:");
            warrior1.display();

            // 3. 生成另一个战士，并对其进行定制化修改
            GameCharacter warrior2 = registry.getClone(key:"WARRIOR_SWORD");
            warrior2.setHealth(health:180); // 修改克隆体的生命值
            // 修改克隆体的可变内部状态（技能等级）
            warrior2.getSkills().stream()
                    .filter(s -> s.getName().equals(anObject:"斩击"))
                    .findFirst()
                    .ifPresent(s -> s.setLevel(level:2)); // 将斩击技能升到2级

            System.out.println(x:"已生成战士 2（已定制):");
            warrior2.display();

            // 4. 验证原始原型对象没有受到影响
            System.out.println(x:"\n原始原型状态（未受影响):");
            registry.getPrototype(key:"WARRIOR_SWORD").display(); // 直接访问原型以供演示

            // 5. 验证深拷贝：修改 warrior2 的技能等级没有影响 warrior1 的技能
            System.out.println(x:"\n战士 1 状态（未受战士 2 修改的影响):");
            warrior1.display();

            System.out.println(x:"======================================");

        } catch (CloneNotSupportedException e) {
            System.err.println("克隆失败: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

运行截图:

```
PS E:\大三资料\大三下课程资料\体系结构\PPT\Code\Prototype> java GameDemo
=== 正在创建原型 ===
战士原型已创建，武器: 基础剑
============================

=== 从原型生成角色 ===
正在克隆游戏角色，类型: 战士
正在克隆技能: 冲锋
正在克隆技能: 斩击
战士特定的克隆逻辑完成。
已生成战士 1:
--- 战士 ---
生命值: 150, 法力值: 20
武器: 基础剑
技能: [技能 [名称=冲锋, 等级=1], 技能 [名称=斩击, 等级=1]]
--------------------
正在克隆游戏角色，类型: 战士
正在克隆技能: 冲锋
正在克隆技能: 斩击
战士特定的克隆逻辑完成。
已生成战士 2 (已定制):
--- 战士 ---
生命值: 180, 法力值: 20
武器: 基础剑
技能: [技能 [名称=冲锋, 等级=1], 技能 [名称=斩击, 等级=2]]
--------------------

原始原型状态 (未受影响):
--- 战士 ---
生命值: 150, 法力值: 20
武器: 基础剑
技能: [技能 [名称=冲锋, 等级=1], 技能 [名称=斩击, 等级=1]]
--------------------

战士 1 状态 (未受战士 2 修改的影响):
--- 战士 ---
生命值: 150, 法力值: 20
武器: 基础剑
技能: [技能 [名称=冲锋, 等级=1], 技能 [名称=斩击, 等级=1]]
--------------------
========================================
```

2、试描述浅克隆和深克隆。

特性	浅克隆 (Shallow Clone)	深克隆 (Deep Clone)
基本定义	创建一个新对象，然后将原始对象中的**非静态字段的值**复制到新对象。	创建一个新对象，并将原始对象中的**非静态字段的值**复制到新对象。对于引用类型的字段，它会**递归地**复制这些字段所引用的对象。
处理基本数据类型 (如 int, float, boolean)	直接复制值。克隆对象和原始对象中的基本类型字段是**完全独立**的。	直接复制值。克隆对象和原始对象中的基本类型字段也是**完全独立**的。
处理引用数据类型 (如 String, Object, List, 数组等)	仅复制**引用地址**（内存地址）。这意味着克隆对象和原始对象中的引用类型字段将**指向堆内存中同一个对象实例**。	创建并复制引用所指向的**对象本身**。这意味着克隆对象和原始对象中的引用类型字段将指向**不同的、但内容相同**的对象实例。
对象独立性	**不完全独立**。如果修改原始对象或克隆对象中的**引用类型成员变量所指向的对象内容**，这种修改会同时反映在另一个对象上（因为它们共享同一个引用对象）。	**完全独立**。修改克隆对象的任何部分（包括其内部引用的对象）**不会**影响原始对象，反之亦然。
实现方式 (Java 示例)	通常只需调用 super.clone() 方法即可（假设父类实现了正确的浅克隆逻辑）。	在调用 super.clone() 获得浅克隆副本后，需要**额外编写代码**，为所有**可变的引用类型**成员变量**也调用它们的** clone() **方法**（或其他复制方法），以实现递归复制。
性能开销	较低，速度较快，因为只复制引用地址，不复制引用对象本身。	较高，速度较慢，因为需要递归地创建和复制所有引用的对象。
适用场景	当对象只包含基本类型和不可变类型（如 String）时，或者明确希望共享引用对象状态时。	当需要一个完全独立的对象副本，不希望原始对象和克隆对象之间存在任何共享状态（尤其是可变状态）时。
潜在问题	容易因共享可变引用对象而引发意外的副作用和 bug。	实现相对复杂，需要仔细处理对象图中的所有引用关系，可能需要处理循环引用问题。

总结来说：

- **浅克隆**：只复制一层，对象内的引用变量还是指向原来的内存地址。像“复制了房子的地址，没复制房子本身”。
- **深克隆**：层层复制，直到所有引用的对象都被复制了一份新的。像“不仅复制了房子的地址，连房子本身和里面的家具都复制了一套全新的”。