软件体系结构

《软件体系结构作业十一》

学　号 22920212204396

姓　名 黄子安

2024 年 4 月 27 日

1、请举例说明克隆模式的其他应用。

克隆模式，也称为原型模式，是一种创建型设计模式，它通过复制现有的对象来创建新对象，而不是通过初始化过程。这种模式特别适用于**创建对象成本较高**的情况，例如对象的创建涉及复杂的计算、数据库操作或其他资源密集型任务

场景：假设在开发一个多人在线游戏，游戏中有各种类型的 NPC。每种类型的 NPC 都有自己独特的属性和技能，但是在游戏的不同区域需要部署许多相似的 NPC。NPC 的基本属性相同，但它们可能在不同的位置或有微小的状态变化。为了优化资源和提高性能，可以使用克隆模式来复制 NPC 对象，而不是每次都从头开始创建每个 NPC，这样可以减少初始化的开销和内存的使用。

```
public class NPC implements Cloneable {
    private String name;
    private int health;
    private String role;
    private int[] position;
    private double agility;
    private double strength;

    public NPC(String name, int health, String role, int x, int y) {
        this.name = name;
        this.health = health;
        this.role = role;
        this.position = new int[] { x, y };
        this.agility = calculateAgility(health);
        this.strength = calculateStrength(health);
    }

    // 模拟的复杂计算，为敏捷性赋值
    private double calculateAgility(int health) {
        double base = 100.0;
        for (int i = 0; i < 10 * health; i++) {
            base *= 1.01;
        }
        return base - health;
    }

    // 模拟的复杂计算，为力量赋值
    private double calculateStrength(int health) {
        double base = 100.0;
        for (int i = 0; i < 10 * health; i++) {
            base *= 0.99;
        }
        return base + health;
    }
```

拷贝克隆方法如下所示，进行一次深拷贝，因为这些 NPC 的位置应该是独立的，不可以相互影响

```
    // 克隆方法，实现深拷贝
    public NPC clone() {
        try {
            NPC cloned = (NPC) super.clone();
            cloned.position = this.position.clone(); // 深拷贝位置数组
            return cloned;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
```

在 main 方法中创建多个对象，分别使用构造和克隆，之后输出分别所花费的时间

```
    public static void main(String[] args) {
        final int NUMBER_OF_OBJECTS = 1000000;
        // 测试构造函数创建对象的时间
        long startConstructorTime = System.nanoTime();
        for (int i = 0; i < NUMBER_OF_OBJECTS; i++) {
            NPC originalNPC = new NPC("Goblin", 100, "Warrior", 100, 200);
        }
        long endConstructorTime = System.nanoTime();
        long durationConstructor = endConstructorTime - startConstructorTime;

        NPC originalNPC = new NPC("Goblin", 100, "Warrior", 100, 200);

        // 测试克隆方法创建对象的时间
        long startCloneTime = System.nanoTime();
        for (int i = 0; i < NUMBER_OF_OBJECTS; i++) {
            NPC clonedNPC = originalNPC.clone();
        }
        long endCloneTime = System.nanoTime();
        long durationClone = endCloneTime - startCloneTime;

        System.out.println("Time taken by Constructor: "
                           + durationConstructor / 1_000_000 + " ms");
        System.out.println("Time taken by Clone: " + durationClone / 1_000_000 + " ms");
    }
```

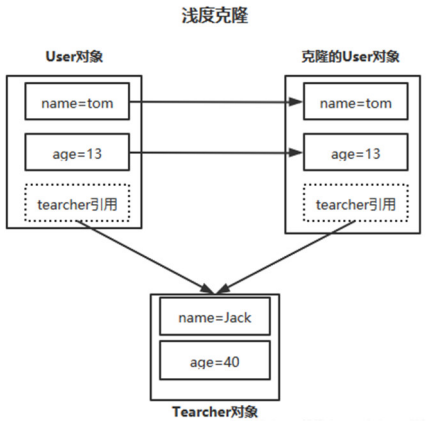
运行结果如下所示，可以发现当构造过程比较繁琐的时候明显拷贝更快

```
D:\Java17\bin\java.exe "-javaagent:D:\IdeaU\IntelliJ IDEA 2023.1.3\lib\idea_rt.j
Time taken by Constructor: 319 ms
Time taken by Clone: 37 ms

Process finished with exit code 0
```

2、试描述浅克隆和深克隆。

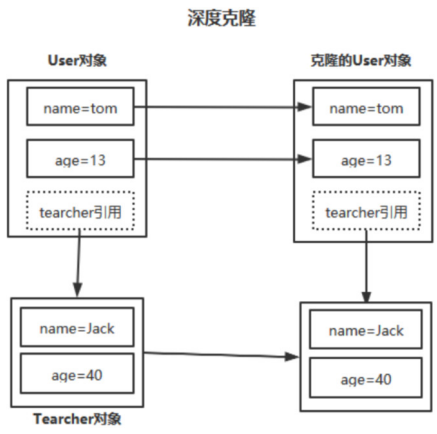
浅克隆会创建一个新对象，然后将原对象的所有字段值复制到新对象中。如果字段是值类型（如 int、float 等基本类型），则直接复制值。如果字段是引用类型（如对象、数组等），则**复制引用而不是复制对象本身**。这意味着新对象和原对象的引用类型字段实际上指向同一个对象。



优点：快速且资源消耗较少，因为它不复制引用对象。

缺点：因为多个对象可能共享同一引用对象，所以对任一对象中的引用类型的修改会影响到所有克隆的对象。这在很多情况下可能导致不希望发生的副作用。

深克隆不仅复制原对象的所有值类型字段，还递归地复制引用类型字段指向的所有对象，这意味着完全创建了一份数据结构的副本。深克隆后，新对象和原对象不会共享任何对象实例。



优点：可以避免浅克隆中引用对象共享的问题，确保新对象的独立性和数据的一致性。修改一个对象不会影响到另一个对象。

缺点：实现复杂，性能开销较大，尤其是当原对象包含大量深层次的引用或大型对象时。

下面通过代码来举例：

```
class Person implements Cloneable {
    String name;
    int age;
    Address address;

    public Person(String name, int age, Address address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }

    public Person shallowClone() {
        try {
            return (Person) super.clone();
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }

    public Person deepClone() {
        try {
            Person cloned = (Person) super.clone();
            cloned.address = new Address(address.street); // 创建新的Address实例
            return cloned;
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }
}

class Address {
    String street;
    public Address(String street) {
        this.street = street;
    }
}
```

测试主函数：

```
public class Main {
    public static void main(String[] args) {
        Address address = new Address("1234 Park Ave");
        Person original = new Person("John", 30, address);

        System.out.println("Original Address: " + original.address.street);
        Person shallowCloned = original.shallowClone();

        Person deepCloned = original.deepClone();
        original.address.street = "5678 Elm St";

        System.out.println("Changed Address: " + original.address.street);
        System.out.println("Shallow Cloned Address: " + shallowCloned.address.street);
        System.out.println("Deep Cloned Address: " + deepCloned.address.street);
    }
}
```

输出结果：

```
D:\Java17\bin\java.exe "-javaagent:D:\IdeaU\IntelliJ IDEA 2023.1.3\lib\idea_rt.jar
Original Address: 1234 Park Ave
Changed Address: 5678 Elm St
Shallow Cloned Address: 5678 Elm St
Deep Cloned Address: 1234 Park Ave

Process finished with exit code 0
```

在这个例子中，修改原对象的地址会影响浅克隆对象的地址，因为它们共享同一个 Address 实例。而深克隆对象不会被影响，因为它有自己的独立 Address 实例。