优秀借鉴

什么是原型模式（Prototype）？应用场景是什么？

【设计模式】原型模式（ 概念简介 | 使用场景 | 优缺点 | 基本用法 ）

设计模式 - Prototype 原型模式

1、简介

用一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型对象相同的新对象。这种类型的设计模式属于创建型模式，用于创建重复对象的同时又能保证性能，是创建对象的最佳方式之一。

优点为性能高,简单:

- 性能高：使用原型模式复用的方式创建实例对象,比使用构造函数重新创建对象性能要高 ；（针对类实例对象开销大的情况）
- 流程简单：原型模式可以简化创建的过程,可以直接修改现有的对象实例的值 ， 达到复用的目的 ；（针对构造函数繁琐的情况）

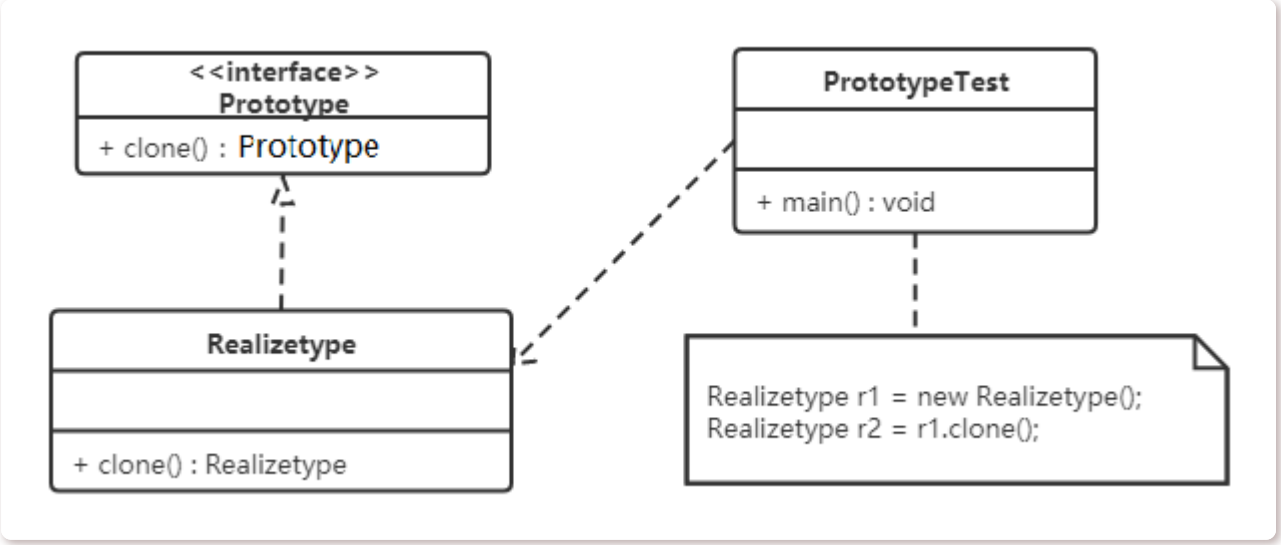
缺点为实现复杂,坑多:

- 覆盖 clone 方法(必须)：必须重写对象的 clone 方法,Java 中提供了 cloneable 标识该对象可以被拷贝,但是必须覆盖 Object 的 clone 方法才能被拷贝;
- 深拷贝与浅拷贝风险：克隆对象时进行的一些修改,容易出错；需要灵活运用深拷贝与浅拷贝操作;

2、结构

原型模式包含如下角色:

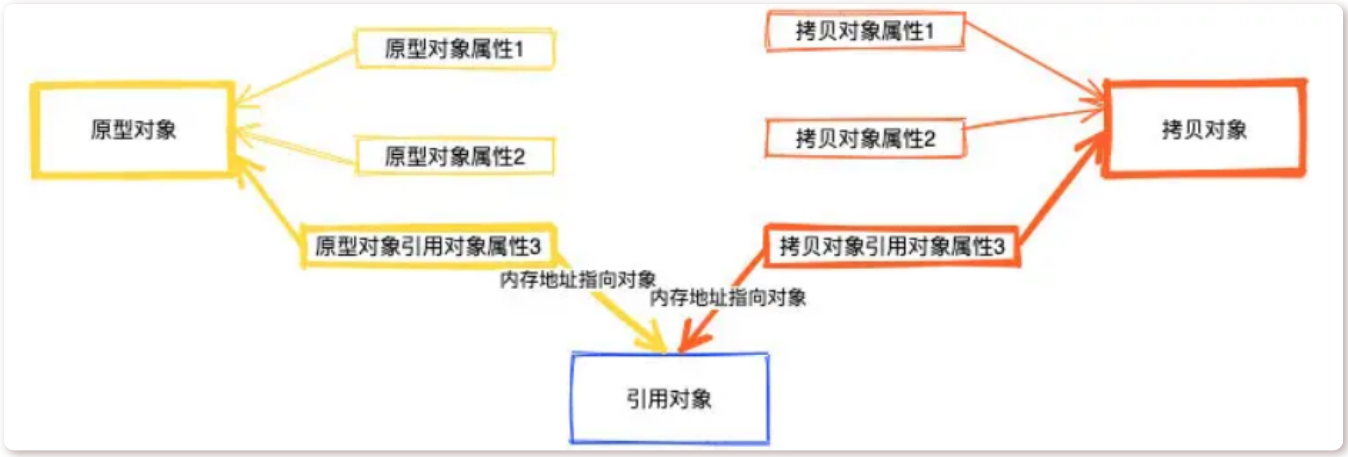
- 抽象原型类：规定了具体原型对象必须实现的的 `clone()` 方法。
- 具体原型类：实现抽象原型类的 `clone()` 方法，它是可被复制的对象。
- 访问类：使用具体原型类中的 `clone()` 方法来复制新的对象



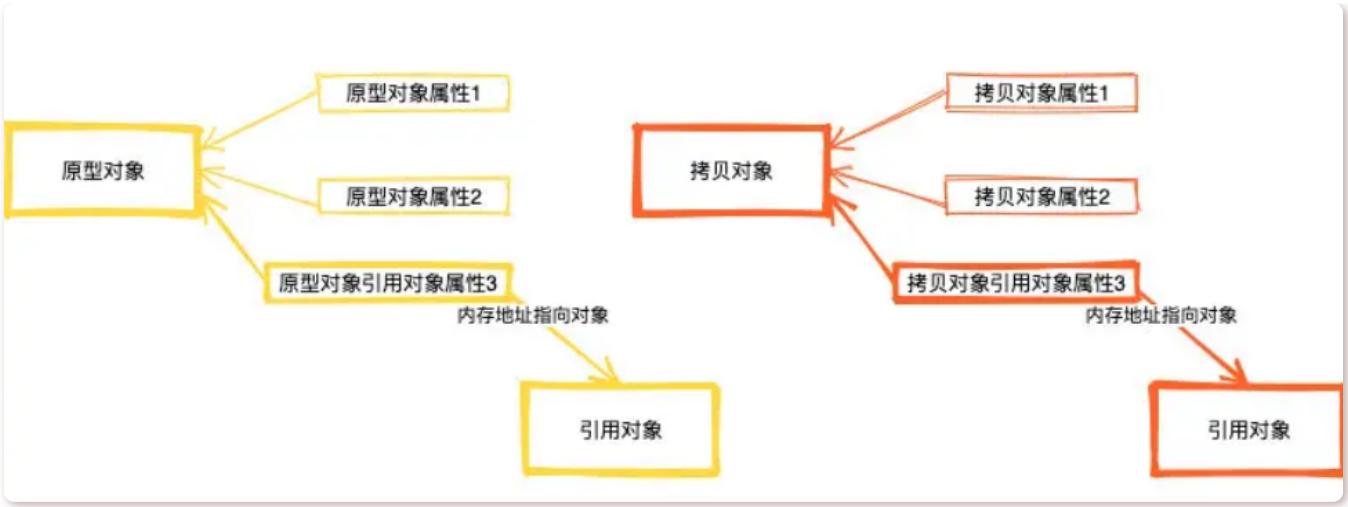
3、浅拷贝和深拷贝

在实现原型模式前，需要了解浅拷贝和深拷贝的概念：

- **浅拷贝**：当拷贝对象只包含简单的数据类型比如`int`、`float` 或者不可变的对象（字符串）时，就直接将这些字段复制到新的对象中。而引用类型的成员变量并没有复制，而是将**引用对象的地址**复制一份给克隆对象；



- **深拷贝**：不管拷贝对象里面简单数据类型还是引用对象类型都是会完全的复制一份到新的对象中。



举个不太恰当的栗子，有一个大箱子(原型对象)，里面放着多个小箱子(成员变量)，小箱子里面放着不同的货物(成员变量值)。

- **浅拷贝**就是不管三七二十一，就是拿一个新的大箱子(克隆对象)，将里面的小箱子(成员变量)直接复制过来得到一模一样的小箱子(地址相同)；
- **深拷贝**就是大箱子(克隆对象)是新的，连里面的小箱子(成员变量)都是新的，最终得到的新大箱子(克隆对象)的小箱子数和小箱子里面的货物都是一样的，但已经是不同的箱子了(地址不同)。

4、浅拷贝实现

最简单与常用的便是浅拷贝，以给三好学生发奖状为例

4.1、实现步骤

1. 原型类实现Cloneable接口并重写clone方法：

```
@Data
public class Citation implements Cloneable {

    /** 学期 */
    private int semester;

    /** 学校 */
    private String school;

    /** 时间戳 */
    private Long timestamp;

    private Student stu;

    public Citation() {
        System.out.println("正在创建奖状……");
    }

```

```
    void show() {
        System.out.println(stu.getName() + "同学：在2020学年第一学期中表现优秀，被评为三好学生。特发此状！");
        System.out.println(school + "于" + timestamp + "颁发");
    }

    @Override
    public Citation clone() {
        try {
            return (Citation) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
}

@Data
class Student {
    private String name;
    private String address;

    public Student(String name, String address) {
        this.name = name;
        this.address = address;
    }
}
```

2. 测试类

```
public class CitationTest {
    public static void main(String[] args) {
        // 生成原型奖状
        Citation c1 = new Citation();
        c1.setStu(new Student("张三", "西安"));
        c1.setSchool("牛马大学");
        c1.setSemester(1);
        c1.setTimestamp(System.currentTimeMillis());

        // 复制奖状
        Citation c2 = c1.clone();

        // 分别调用show方法
        System.out.println("===== c1的show =====");
        c1.show();
        System.out.println("===== c2的show =====");
        c2.show();
        System.out.println("===== ======= =====");

        // 比较原型与克隆对象地址
```

```
21          System.out.println("比较原型与克隆对象地址：" + (c1 == c2));
22
23          // 比较成员属性地址
24          System.out.println("比较学生成员属性的地址(Student)：\t" +
    (c1.getStu() == c2.getStu()));
25          System.out.println("比较学校成员属性的地址(String)：\t" +
    (c1.getSchool() == c2.getSchool()));
26          System.out.println("比较学期成员属性的地址(int)：\t" +
    (c1.getSemester() == c2.getSemester()));
27          System.out.println("比较时间成员属性的地址(Long)：\t" +
    (c1.getTimestamp() == c2.getTimestamp()));
28      }
29  }
```

3．运行结果

```
正在创建奖状……
===== c1的show =====
张三同学：在2020学年第一学期中表现优秀，被评为三好学生。特发此状！
牛马大学于1676199716225颁发
===== c2的show =====
张三同学：在2020学年第一学期中表现优秀，被评为三好学生。特发此状！
牛马大学于1676199716225颁发
===== ======= =====
比较原型与克隆对象地址：false
比较学生成员属性的地址(Student)：    true
比较学校成员属性的地址(String)：     true
比较学期成员属性的地址(int)：    true
比较时间成员属性的地址(Long)：   true
```

4.2、结果分析

1．构造器

在克隆生成c2时并没有调用构造器，只有在手动创建c1时才会调用构造器，这证明这是使用了原型模式复用的方式创建实例对象。

2．原型对象与克隆对象比较

通过比较，这两个对象的地址并不相同，这证明虽然复制过程没有调用构造器，但是生成的对象是一个全新的实例，在栈中有属于自己的地址。

3．引用类型比较

在奖状类中定义了三种实际应用中常遇到的类型，分别是自定义引用类型、字符串类型和包装类，但是通过比较，原型对象和拷贝对象的引用类型为同一个对象，共用一个地址。

4．普通数据类型比较

这里还设置了一个int类型的普通数据类型，由于普通类型是将值直接赋值到新的拷贝对象中，因此这里的 `==` 比较的是两者的值。

> 大家可以尝试一下修改c2或c1中的某个引用类型再调用show方法进行查看，对比两者的数据变化，这可能对这一部分的理解会更深刻。

5、深拷贝思路

5.1、clone方法

经过上述浅拷贝的介绍，我们知道在实现原型模式最重要的便是实现Cloneable接口，重写clone方法，因此我们可以在重写clone方法时，再将成员变量里的引用类型变量进行克隆(前提是这一引用类型支持拷贝)。

5.2、序列化与反序列化

正是可以破坏单例因素之一的序列化与反序列化，通过对象流将对象输出后再写入，这种方式生成的拷贝对象便是深拷贝后的实例对象。

> 针对上面的两种写法其实都是可以实现原型模式的，但是不管用哪种方式，深拷贝都比浅拷贝花时间和空间，所以还是酌情考虑。其实在现在已经有很多针对浅拷贝和深拷贝的工具类
>
> - 深拷贝(deep copy):SerializationUtils
> - 浅拷贝(shallow copy):BeanUtils

6、应用场景

- 当一个系统应该独立于它的产品创建，构成和表示时。
- 当要实例化的类是在运行时刻指定时，例如，通过动态装载。
- 为了避免创建一个与产品类层次平行的工厂类层次时。
- 当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

代码举例：需要创建 10 个 StudentVO 对象，依次调用一个创建好的 StudentVO 对象的 clone 方法 10 次，即可创建 10 个不同的对象。(这是频繁创建大量的对象 ，该场景下适合使用原型模式)

```
1  public class Main {
2      public static void main(String[] args) {
3          try {
```

```
            // 测试使用 clone 方法实现的原型模式 , 使用原型模式创建 10 个对象
            StudentVO prototypeStuVO = new StudentVO();
            for (int i = 0; i < 10; i++) {
                // 1 . 使用 clone 方法创建对象
                StudentVO student = (StudentVO) prototypeStuVO.clone();
                // 2 . 设置克隆出的对象参数
                student.setName("Tom" + i);
                student.setAge(10 + i);
                System.out.println(student);
            }
        } catch (CloneNotSupportedException e) {
            //捕获 clone 方法可能产生的异常
            e.printStackTrace();
        }
    }
}
```