原型模式

在程序中，有些对象的创建过程较为复杂，初始化时需要很多资源，而且有时候需要频繁创建，若使用 new 来创建每一个对象的话会造成很大的消耗，此时就可以使用**原型模式：用原型实例指定创建对象的种类，并且通过复制这些原型创建新的对象**。
原型模式的核心就是 Object 的 clone() 方法，借助此方法，我们可以实现在内存中直接拷贝一个对象，无需调用构造函数，这要比直接 new 一个对象的性能好很多。
相似对象的创建，很多情况下，复制所得到的对象与原型对象并不是完全相同的，它们的某些属性值存在不同。通过原型模式获得相同对象后可以再对其属性进行修改，从而获取所需对象。

自定义类实现拷贝的过程：

1）首先，派生类实现 Cloneable 接口，这是一个标志性接口。
2）其次，在派生类中覆盖基类的 clone() 方法（因为从其他包中通过继承得到的 protected 方法只能在本类中使用，无法再本包的其他类中调用），如果派生类的 clone() 方法需要在其他包中调用则声明为 public，在派生类的 clone()方法中，调用 super.clone()。
3）Object 类的 clone() 方法是浅拷贝，只是将对象成员变量的值拷贝一份，包括引用成员变量，也只是拷贝一份引用。这样的话拷贝出来的对象和原始对象拥有同一个对象的引用，一般情况下并不符合我们的需求，所以为了实现深拷贝，应该在派生类的 clone() 方法中作出相应修改。注：String 类对象也是引用类型，但是它是不可变的，所以在拷贝对象中修改的话只会重新创建一个字符串，并不会影响原始对象，所以无需对该成员变量的拷贝作出相应修改。
示例代码：

```
//实现深拷贝
class Test
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Teacher t = new Teacher("Tom");
        Student s1 = new Student("zhansan", 20, t);
        Student s2 = (Student) s1.clone();
        //修改拷贝对象的引用对象
        s2.teacher.name = "Jack";
        //输出原始对象的引用对象
        System.out.println(s1.teacher.name); //Tom  并未发生修改

    }
}

class Student implements Cloneable
```

```
{
    String name;
    int age;
    Teacher teacher;
    public Student(String name, int age, Teacher teacher)
    {
        super();
        this.name = name;
        this.age = age;
        this.teacher = teacher;
    }
    @Override
    protected Object clone() throws CloneNotSupportedException
    {
        Student s =    (Student) super.clone();
        s.teacher = (Teacher) s.teacher.clone();
        return s;
    }
}

class Teacher implements Cloneable
{
    String name;

    public Teacher(String name)
    {
        super();
        this.name = name;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}
```

模式的优缺点：

原型模式最大的优点在于可以快速创建很多相同或相似的对象，简化对象的创建过程，还可以保存对象的一些中间状态；其缺点在于需要为每一个类配备一个克隆方法，因此对已有类进行改造比较麻烦，需要修改其源代码，并且在实现深克隆时需要编写较为复杂的代码。