原型模式(Prototype)

1. 需求背景

根据接口克隆出一个与原来的实现类一模一样(或者略有不同可以根据需求改变)的实现类

如有一个订单接口,个人订单和单位订单都实现该接口,而当该订单的数量超过1000,就需要分成两个订单(与原来的订单是同类型的),如果还是多,继续拆,直到都不超过1000.

很明显,我们new出来的第一个订单肯定是

```
Prototype p=new PrototypeImp;
```

进一步做拆分时,我们只知道接口,不知道具体实现类,这时拆分成的订单需要new什么类型的类就不知道了,需要原型模式去解决(接口不能依赖于具体实现类)

2. 实现:

原型接口:

```
public interface Prototype {
    Prototype clone();
}
```

两个实现类:

```
public class PrototypeImp implements Prototype {

    private String name;
    @Override
    public Prototype clone() {
        Prototype prototype=new PrototypeImp();
        setName(name);
        return prototype;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

public class PrototypeImp2 implements Prototype {

    private String age;
    @Override
    public Prototype clone() {
        Prototype prototype=new PrototypeImp2();
        setAge(age);
        return prototype;
    }
    public String getAge() {
        return age;
    }
    public void setAge(String age) {
        this.age = age;
    }


}
```

客户端:

```
public class Client {

    private Prototype prototype;

    public Client(Prototype prototype) {
        super();
        this.prototype = prototype;
    }

    public Prototype operate(Prototype prototype){
        Prototype prototypeNew=prototype.clone();
        return prototypeNew;
    }
    public Prototype getPrototype() {
        return prototype;
    }

    public void setPrototype(Prototype prototype) {
        this.prototype = prototype;
    }

}
```

在客户端处理clone操作,可以看到,传入客户端的原型接口属性的实例我们并不清楚,但是我们通过clone方法就可以完全的复制该实例(除了内存地址),这里我们需要遵循依赖倒置原则所以不能去通过instanceof方法去通过判断其实例化类型来做出操作

注意这里clone跟new有什么相同与区别

相同点:两者都是生成了一个新的实例化对象,clone本身也是new出来的

不同点:new出来的对象属性都是null 或者是default的 而clone出的一般来说都是有值得,它依赖于被clone的实例,所以clone必先有需要被clone的实例,但是一旦clone完成则原实例与clone出来的实例就没有关系了

由此:

1. 原型模式的功能

原型模式的功能实际上包含两个方面:

- **一个是通过克隆来创建新的对象实例;**
- **另一个是为克隆出来的新的对象实例复制原型实例属性的值。**

2. 关于书中的Java提供的clone方法,我觉得是有问题的,首先,并没有解决传入接口类型时(无法判断具体的实现类)时,怎样去创建实例,违反了依赖倒置原则,而且原型接口没有必要存在了,根本没用到.

但是其方法也是可以借鉴的,可以说也许用处更广,因为该方法粒度较小,只要是实例就可以用,这里需要注意深度克隆的时候,需要注意

```
public Object clone(){
    PersonalOrder obj=null;
    try {
        obj =(PersonalOrder)super.clone();
        //下面这一句话不可少
        obj.setProduct(
            (Product)this.product.clone());
    } catch (CloneNotSupportedException e)
        e.printStackTrace();
    }
    return obj;
}
```

以及

不可缺少“obj.setProduct((Product)this.product.clone());”这句话。为什么呢？

原因在于调用 super.clone()方法的时候，Java 是先开辟一块内存的空间，然后把实例对象的值原样拷贝过去，对于基本数据类型这样做是没有问题的，而属性 product 是一个引用类型，把值拷贝过去的意思就是把对应的内存地址拷贝过去了，也就是说克隆后的对象实例的 product 和原型对象实例的 product 指向的是同一块内存空间，是同一个产品实例。

因此要想正确地执行深度拷贝，必须手工地对每一个引用类型的属性进行克隆，并重新设置，覆盖掉 super.clone()所拷贝的值。

- 浅度克隆：只负责克隆按值传递的数据（比如基本数据类型、String 类型）。
- 深度克隆：除了浅度克隆要克隆的值外，还负责克隆引用类型的数据，基本上就是被克隆实例所有的属性数据都会被克隆出来。

如果clone的属性值改变影响到了原型实例的属性值,说明深度clone失败.

3.原型管理器.

原型管理器是由于系统可能存在多个原型,这些原型有可能可以增添销毁,所以需要维护一个原型的注册表,是一个Map,注意这里的方法都是static并且是加了锁的(多线程环境时明显是要保持数据的一致性的).

原型模式的缺点

原型模式最大的缺点就在于每个原型的子类都必须实现 clone 的操作，尤其在包含引用类型的对象时，clone 方法会比较麻烦，必须要能够递归地让所有的相关对象都要正确地实现克隆。