

原型模式

整理自：《java与模式》之原型模式

在阎宏博士的《JAVA与模式》一书中开头是这样描述原型（Prototype）模式的：

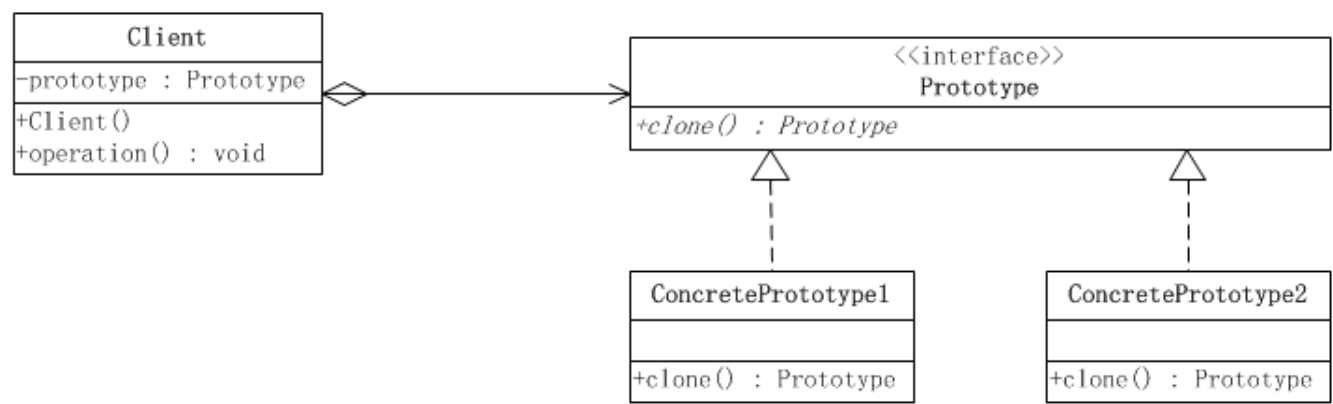
原型模式属于对象的创建模式。通过给出一个原型对象来指明所有创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象。这就是选型模式的用意。

原型模式的结构

原型模式要求对象实现一个可以“克隆”自身的接口，这样就可以通过复制一个实例对象本身来创建一个新的实例。这样一来，通过原型实例创建新的对象，就不再需要关心这个实例本身的类型，只要实现了克隆自身的方法，就可以通过这个方法来获取新的对象，而无须再去通过new来创建。

原型模式有两种表现形式：（1）简单形式、（2）登记形式，这两种表现形式仅仅是原型模式的不同实现。

简单形式的原型模式



这种形式涉及到三个角色：

- （1）客户(Client)角色：客户类提出创建对象的请求。
- （2）抽象原型(Prototype)角色：这是一个抽象角色，通常由一个Java接口或Java抽象类实现。此角色给出所有的具体原型类所需的接口。
- （3）具体原型（Concrete Prototype）角色：被复制的对象。此角色需要实现抽象的原

型角色所要求的接口。

源代码

抽象原型角色

```
public interface Prototype{  
    /**  
     * 克隆自身的方法  
     * @return 一个从自身克隆出来的对象  
     */  
    public Object clone();  
}
```

具体原型角色

```
public class ConcretePrototype1 implements Prototype {  
    public Prototype clone(){  
        //最简单的克隆，新建一个自身对象，由于没有属性就不再复制值了  
        Prototype prototype = new ConcretePrototype1();  
        return prototype;  
    }  
}
```

```
public class ConcretePrototype2 implements Prototype {  
    public Prototype clone(){  
        //最简单的克隆，新建一个自身对象，由于没有属性就不再复制值了  
        Prototype prototype = new ConcretePrototype2();  
        return prototype;  
    }  
}
```

客户端角色

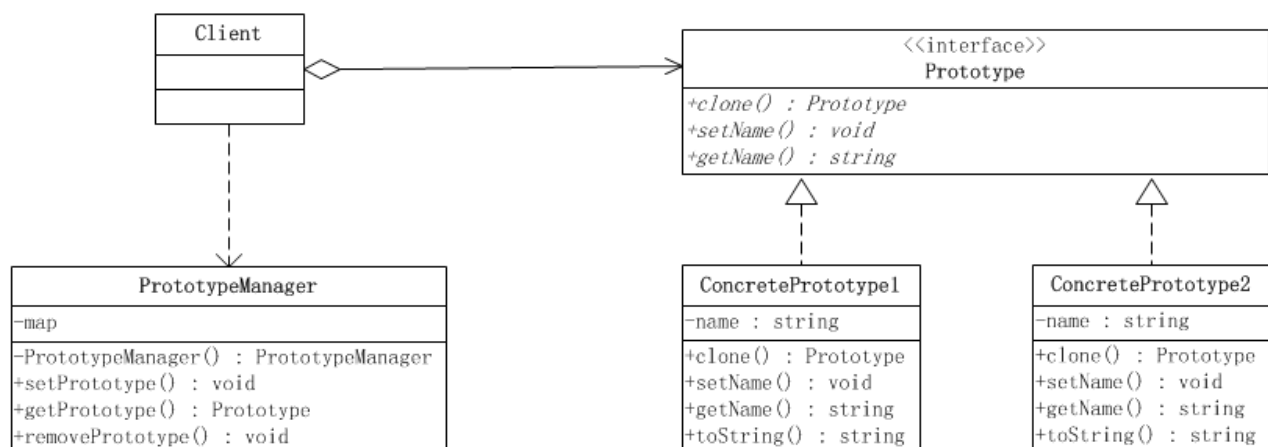
```
public class Client {  
    /**  
     * 持有需要使用的原型接口对象  
     */  
    private Prototype prototype;
```

```

/**
 * 构造方法，传入需要使用的原型接口对象
 */
public Client(Prototype prototype){
    this.prototype = prototype;
}
public void operation(Prototype example){
    //需要创建原型接口的对象
    Prototype copyPrototype = prototype.clone();
}
}

```

登记形式的原型模式



作为原型模式的第二种形式，它多了一个原型管理器(PrototypeManager)角色，该角色的作用是：创建具体原型类的对象，并记录每一个被创建的对象。

源代码

抽象原型角色

```

public interface Prototype{
    public Prototype clone();
    public String getName();
    public void setName(String name);
}

```

具体原型角色

```
public class ConcretePrototype1 implements Prototype {
    private String name;
    public Prototype clone(){
        ConcretePrototype1 prototype = new ConcretePrototype1();
        prototype.setName(this.name);
        return prototype;
    }
    public String toString(){
        return "Now in Prototype1 , name = " + this.name;
    }
    @Override
    public String getName() {
        return name;
    }

    @Override
    public void setName(String name) {
        this.name = name;
    }
}
```

```
public class ConcretePrototype2 implements Prototype {
    private String name;
    public Prototype clone(){
        ConcretePrototype2 prototype = new ConcretePrototype2();
        prototype.setName(this.name);
        return prototype;
    }
    public String toString(){
        return "Now in Prototype2 , name = " + this.name;
    }
    @Override
    public String getName() {
        return name;
    }

    @Override
    public void setName(String name) {
```

```

        this.name = name;
    }
}

```

原型管理器角色保持一个聚集，作为对所有原型对象的登记，这个角色提供必要的方法，供外界增加新的原型对象和取得已经登记过的原型对象。

```

public class PrototypeManager {
    /**
     * 用来记录原型的编号和原型实例的对应关系
     */
    private static Map<String,Prototype> map = new HashMap<String,Prototype>();
    /**
     * 私有化构造方法，避免外部创建实例
     */
    private PrototypeManager(){}
    /**
     * 向原型管理器里面添加或是修改某个原型注册
     * @param prototypeId 原型编号
     * @param prototype 原型实例
     */
    public synchronized static void setPrototype(String prototypeId , Prototype prototype){
        map.put(prototypeId, prototype);
    }
    /**
     * 从原型管理器里面删除某个原型注册
     * @param prototypeId 原型编号
     */
    public synchronized static void removePrototype(String prototypeId){
        map.remove(prototypeId);
    }
    /**
     * 获取某个原型编号对应的原型实例
     * @param prototypeId 原型编号
     * @return 原型编号对应的原型实例
     * @throws Exception 如果原型编号对应的实例不存在，则抛出异常
     */
    public synchronized static Prototype getPrototype(String prototypeId

```

```

    ) throws Exception{
        Prototype prototype = map.get(prototypeId);
        if(prototype == null){
            throw new Exception("您希望获取的原型还没有注册或已被销毁");
        }
        return prototype;
    }
}

```

客户端角色

```

public class Client {
    public static void main(String[]args){
        try{
            Prototype p1 = new ConcretePrototype1();
            PrototypeManager.setPrototype("p1", p1);
            //获取原型来创建对象
            Prototype p3 = PrototypeManager.getPrototype("p1").clone();
            p3.setName("张三");
            System.out.println("第一个实例: " + p3);
            //有人动态的切换了实现
            Prototype p2 = new ConcretePrototype2();
            PrototypeManager.setPrototype("p1", p2);
            //重新获取原型来创建对象
            Prototype p4 = PrototypeManager.getPrototype("p1").clone();
            p4.setName("李四");
            System.out.println("第二个实例: " + p4);
            //有人注销了这个原型
            PrototypeManager.removePrototype("p1");
            //再次获取原型来创建对象
            Prototype p5 = PrototypeManager.getPrototype("p1").clone();
            p5.setName("王五");
            System.out.println("第三个实例: " + p5);
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}

```

两种形式的比较

简单形式和登记形式的原型模式各有其长处和短处。

如果需要创建的原型对象数目较少而且比较固定的话，可以采取第一种形式。在这种情况下，原型对象的引用可以由客户端自己保存。

如果要创建的原型对象数目不固定的话，可以采取第二种形式。在这种情况下，客户端不保存对原型对象的引用，这个任务被交给管理员对象。在复制一个原型对象之前，客户端可以查看管理员对象是否已经有一个满足要求的原型对象。如果有，可以直接从管理员类取得这个对象引用；如果没有，客户端就需要自行复制此原型对象。

Java中的克隆方法

Java的所有类都是从`java.lang.Object`类继承而来的，而`Object`类提供`protected Object clone()`方法对对象进行复制，子类当然也可以把这个方法置换掉，提供满足自己需要的复制方法。对象的复制有一个基本问题，就是对象通常都有对其他对象的引用。当使用`Object`类的`clone()`方法来复制一个对象时，此对象对其他对象的引用也同时会被复制一份

Java语言提供的`Cloneable`接口只起一个作用，就是在运行时期通知Java虚拟机可以安全地在这个类上使用`clone()`方法。通过调用这个`clone()`方法可以得到一个对象的复制。由于`Object`类本身并不实现`Cloneable`接口，因此如果所考虑的类没有实现`Cloneable`接口时，调用`clone()`方法会抛出`CloneNotSupportedException`异常。

克隆满足的条件

`clone()`方法将对象复制了一份并返还给调用者。所谓“复制”的含义与`clone()`方法是怎么实现的。一般而言，`clone()`方法满足以下的描述：

(1) 对任何的对象`x`，都有：`x.clone() != x`。换言之，克隆对象与原对象不是同一个对象。

(2) 对任何的对象`x`，都有：`x.clone().getClass() == x.getClass()`，换言之，克隆对象与原对象的类型一样。

(3) 如果对象`x`的`equals()`方法定义其恰当的话，那么`x.clone().equals(x)`应当成立的。

在JAVA语言的API中，凡是提供了`clone()`方法的类，都满足上面的这些条件。JAVA语言

的设计师在设计自己的clone()方法时，也应当遵守着三个条件。一般来说，上面的三个条件中的前两个是必需的，而第三个是可选的。

浅克隆和深克隆

无论你是自己实现克隆方法，还是采用Java提供的克隆方法，都存在一个浅度克隆和深度克隆的问题。

- 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），而不复制它所引用的对象，换言之，所有的对其他对象的引用都仍然指向原来的对象。

- 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。那些引用其他对象的变量将指向被复制过的新对象，而不再是原有的那些被引用的对象。换言之，深度克隆把要复制的对象所引用的对象都复制了一遍，而这种对被引用到的对象的复制叫做间接复制。

深度克隆要深入到多少层，是一个不易确定的问题。在决定以深度克隆的方式复制一个对象的时候，必须决定对间接复制的对象时采取浅度克隆还是继续采用深度克隆。因此，在采取深度克隆时，需要决定多深才算深。此外，在深度克隆的过程中，很可能会出现循环引用的问题，必须小心处理。

利用序列化实现深度克隆

把对象写到流里的过程是序列化(Serialization)过程；而把对象从流中读出来的过程则叫反序列化(Deserialization)过程。应当指出的是，写到流里的是对象的一个拷贝，而原对象仍然存在于JVM里面。

在Java语言里深度克隆一个对象，常常可以先使对象实现Serializable接口，然后把对象（实际上只是对象的拷贝）写到一个流里（序列化），再从流里读回来（反序列化），便可以重建对象。

```
public Object deepClone() throws IOException, ClassNotFoundException{
    //将对象写到流里
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(bos);
    oos.writeObject(this);
}
```



```
//从流里读回来
ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
ObjectInputStream ois = new ObjectInputStream(bis);
return ois.readObject();
}
```

这样做的前提就是对象以及对象内部所有引用到的对象都是可序列化的，否则，就需要仔细考察那些不可序列化的对象可否设成transient，从而将之排除在复制过程之外。

浅度克隆显然比深度克隆更容易实现，因为Java语言的所有类都会继承一个clone()方法，而这个clone()方法所做的正式浅度克隆。

有一些对象，比如线程(Thread)对象或Socket对象，是不能简单复制或共享的。不管是使用浅度克隆还是深度克隆，只要涉及这样的间接对象，就必须把间接对象设成transient而不予复制；或者由程序自行创建出相当的同种对象，权且当做复制件使用。

孙大圣的身外身法术

孙大圣的身外身本领如果在Java语言里使用原型模式来实现的话，会怎么样呢？首先，齐天大圣(The Greatest Sage)即TheGreatestSage类扮演客户角色。齐天大圣持有一个狻猊(Monkey)的实例，而狻猊就是大圣本尊。Monkey类具有继承自java.lang.Object的clone()方法，因此，可以通过调用这个克隆方法来复制一个Monkey实例。

孙大圣本人用TheGreatestSage类代表

```
public class TheGreatestSage {
    private Monkey monkey = new Monkey();

    public void change(){
        //克隆大圣本尊
        Monkey copyMonkey = (Monkey)monkey.clone();
        System.out.println("大圣本尊的生日是：" + monkey.getBirthDate());
        System.out.println("克隆的大圣的生日是：" + monkey.getBirthDate());
        System.out.println("大圣本尊跟克隆的大圣是否为同一个对象 " + (monkey == copyMonkey));
        System.out.println("大圣本尊持有的金箍棒 跟 克隆的大圣持有的金箍棒是否为同一个对象？ " + (monkey.getStaff() == copyMonkey.getStaff()));
    }
}
```

```

public static void main(String[]args){
    TheGreatestSage sage = new TheGreatestSage();
    sage.change();
}
}

```

大圣本尊由Monkey类代表，这个类扮演具体原型角色：

```

public class Monkey implements Cloneable {
    //身高
    private int height;
    //体重
    private int weight;
    //生日
    private Date birthDate;
    //金箍棒
    private GoldRingedStaff staff;
    /**
     * 构造函数
     */
    public Monkey(){
        this.birthDate = new Date();
        this.staff = new GoldRingedStaff();
    }
    /**
     * 克隆方法
     */
    public Object clone(){
        Monkey temp = null;
        try {
            temp = (Monkey) super.clone();
        } catch (CloneNotSupportedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } finally {
            return temp;
        }
    }
    public int getHeight() {

```

```

        return height;
    }
    public void setHeight(int height) {
        this.height = height;
    }
    public int getWeight() {
        return weight;
    }
    public void setWeight(int weight) {
        this.weight = weight;
    }
    public Date getBirthDate() {
        return birthDate;
    }
    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }
    public GoldRingedStaff getStaff() {
        return staff;
    }
    public void setStaff(GoldRingedStaff staff) {
        this.staff = staff;
    }
}

```

大圣还持有一个金箍棒的实例，金箍棒类GoldRingedStaff:

```

public class GoldRingedStaff {
    private float height = 100.0f;
    private float diameter = 10.0f;
    /**
     * 增长行为，每次调用长度和半径增加一倍
     */
    public void grow(){
        this.diameter *= 2;
        this.height *= 2;
    }
    /**
     * 缩小行为，每次调用长度和半径减少一半
     */
}

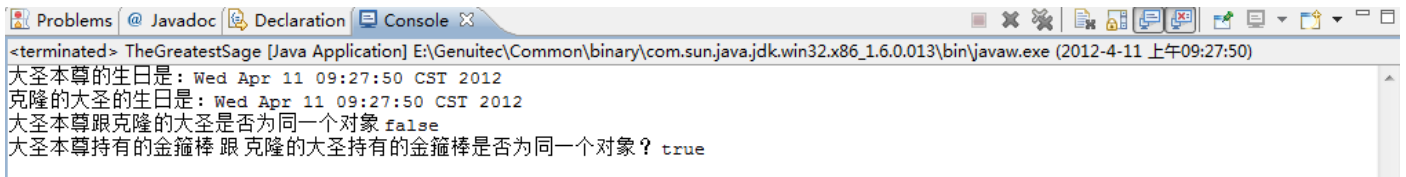
```

```

    */
    public void shrink(){
        this.diameter /= 2;
        this.height /= 2;
    }
}

```

当运行TheGreatestSage类时，首先创建大圣本尊对象，而后 **浅度克隆** 大圣本尊对象。程序在运行时打印出的信息如下：



```

<terminated> TheGreatestSage [Java Application] E:\Genuitec\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013\bin\javaw.exe (2012-4-11 上午09:27:50)
大圣本尊的生日是：Wed Apr 11 09:27:50 CST 2012
克隆的大圣的生日是：Wed Apr 11 09:27:50 CST 2012
大圣本尊跟克隆的大圣是否为同一个对象 false
大圣本尊持有的金箍棒 跟 克隆的大圣持有的金箍棒是否为同一个对象？ true

```

可以看出，首先，复制的大圣本尊具有和原始的大圣本尊对象一样的birthDate，而本尊对象不相等，这表明他们二者是克隆关系；其次，复制的大圣本尊所持有的金箍棒和原始的大圣本尊所持有的金箍棒为同一个对象。这表明二者所持有的金箍棒根本是一根，而不是两根。

正如前面所述，继承自java.lang.Object类的clone()方法是浅克隆。换言之，齐天大圣的所有化身所持有的金箍棒引用全都是指向一个对象的，这与《西游记》中的描写并不一致。要纠正这一点，就需要考虑使用 **深克隆**。

为做到 **深度克隆**，所有需要复制的对象都需要实现java.io.Serializable接口。

孙大圣的源代码：

```

public class TheGreatestSage {
    private Monkey monkey = new Monkey();

    public void change() throws IOException, ClassNotFoundException{
        Monkey copyMonkey = (Monkey)monkey.deepClone();
        System.out.println("大圣本尊的生日是：" + monkey.getBirthDate());
        System.out.println("克隆的大圣的生日是：" + monkey.getBirthDate());
        System.out.println("大圣本尊跟克隆的大圣是否为同一个对象 " + (monkey == copyMonkey));
        System.out.println("大圣本尊持有的金箍棒 跟 克隆的大圣持有的金箍棒是否为同一个对象？ " + (monkey.getStaff() == copyMonkey.getStaff()));
    }
}

```

```

    public static void main(String[]args) throws IOException, ClassNotFoundException,
undException{
        TheGreatestSage sage = new TheGreatestSage();
        sage.change();
    }
}

```

在大圣本尊Monkey类里面，有两个克隆方法，一个是clone()，也即浅克隆；另一个是deepClone()，也即深克隆。在深克隆方法里，大圣本尊对象（一个拷贝）被序列化，然后又被反序列化。反序列化的对象就成了一个深克隆的结果。

```

public class Monkey implements Cloneable,Serializable {
    //身高
    private int height;
    //体重
    private int weight;
    //生日
    private Date birthDate;
    //金箍棒
    private GoldRingedStaff staff;
    /**
     * 构造函数
     */
    public Monkey(){
        this.birthDate = new Date();
        staff = new GoldRingedStaff();
    }
    /**
     * 克隆方法
     */
    public Object clone(){
        Monkey temp = null;
        try {
            temp = (Monkey) super.clone();
        } catch (CloneNotSupportedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } finally {
            return temp;
        }
    }
}

```

```

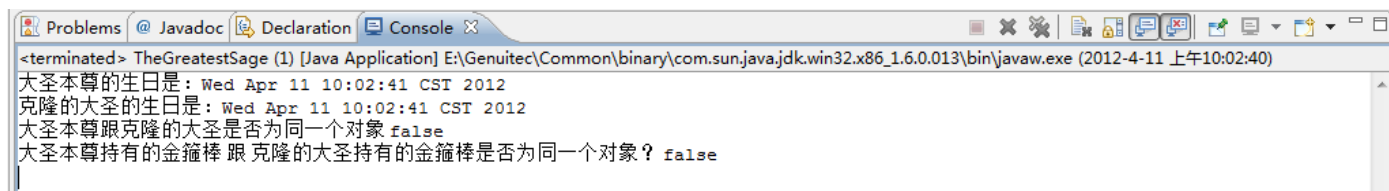
        }
    }
    public Object deepClone() throws IOException, ClassNotFoundException{
        //将对象写到流里
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        oos.writeObject(this);
        //从流里读回来
        ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bis);
        return ois.readObject();
    }
    public int getHeight() {
        return height;
    }
    public void setHeight(int height) {
        this.height = height;
    }
    public int getWeight() {
        return weight;
    }
    public void setWeight(int weight) {
        this.weight = weight;
    }
    public Date getBirthDate() {
        return birthDate;
    }
    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }
    public GoldRingedStaff getStaff() {
        return staff;
    }
    public void setStaff(GoldRingedStaff staff) {
        this.staff = staff;
    }
}

```

可以看到，大圣本尊持有一个金箍棒（GoldRingedStaff）的实例。在大圣复制件里面，此金箍棒实例是原大圣本尊对象所持有的金箍棒对象的一个拷贝。在大圣本尊对象被序列化和反序列化时，它所持有的金箍棒对象也同时被序列化和反序列化，这使得复制的大圣的金箍棒和原大圣本尊对象所持有的金箍棒对象是两个独立的对象。

```
public class GoldRingedStaff implements Serializable{
    private float height = 100.0f;
    private float diameter = 10.0f;
    /**
     * 增长行为，每次调用长度和半径增加一倍
     */
    public void grow(){
        this.diameter *= 2;
        this.height *= 2;
    }
    /**
     * 缩小行为，每次调用长度和半径减少一半
     */
    public void shrink(){
        this.diameter /= 2;
        this.height /= 2;
    }
}
```

运行结果：



```
<terminated> TheGreatestSage (1) [Java Application] E:\Genuitec\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013\bin\javaw.exe (2012-4-11 上午10:02:40)
大圣本尊的生日是：Wed Apr 11 10:02:41 CST 2012
克隆的大圣的生日是：Wed Apr 11 10:02:41 CST 2012
大圣本尊跟克隆的大圣是否为同一个对象 false
大圣本尊持有的金箍棒 跟 克隆的大圣持有的金箍棒是否为同一个对象？ false
```

从运行的结果可以看出，大圣的金箍棒和他的身外之身的金箍棒是不同的对象。这是因为使用了深克隆，从而把大圣本尊所引用的对象也都复制了一遍，其中也包括金箍棒。

原型模式的优点

原型模式允许在运行时动态改变具体的实现类型。原型模式可以在运行期间，由客户来注册符合原型接口的实现类型，也可以动态地改变具体的实现类型，看起来接口没有任何变化，但其实运行的已经是另外一个类实例了。因为克隆一个原型就类似于实例化一个类。

原型模式的缺点

原型模式最主要的缺点是每一个类都必须配备一个克隆方法。配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类来说不是很难，而对于已经有的类不一定很容易，特别是当一个类引用不支持序列化的间接对象，或者引用含有循环结构的时候。