Java 提升篇：对象克隆（复制）

引论

假如说你想复制一个简单变量。很简单：

```
int apples = 5;
int pears = apples;
```

不仅仅是 int 类型 ,其它七种原始数据类型(boolean,char,byte,short,float,double.long)同样适用于该类情况。

但是如果你复制的是一个对象，情况就有些复杂了。

假设说我是一个 beginner，我会这样写：

```
class Student {
    private int number;
    public int getNumber() {
        return number;
    }
    public void setNumber(int number) {
        this.number = number;
    }
}
public class Test {
    public static void main(String args[]) {
        Student stu1 = new Student();
        stu1.setNumber(12345);
        Student stu2 = stu1;
        System.out.println("学生 1:" + stu1.getNumber());
        System.out.println("学生 2:" + stu2.getNumber());
    }
}
```

结果：

```
学生 1:12345
学生 2:12345
```

这里我们自定义了一个学生类，该类只有一个 number 字段。

我们新建了一个学生实例，然后将该值赋值给 stu2 实例。(Student stu2 = stu1;)

再看看打印结果，作为一个新手，拍了拍胸腹，对象复制不过如此，

难道真的是这样吗？

我们试着改变 stu2 实例的 number 字段，再打印结果看看：

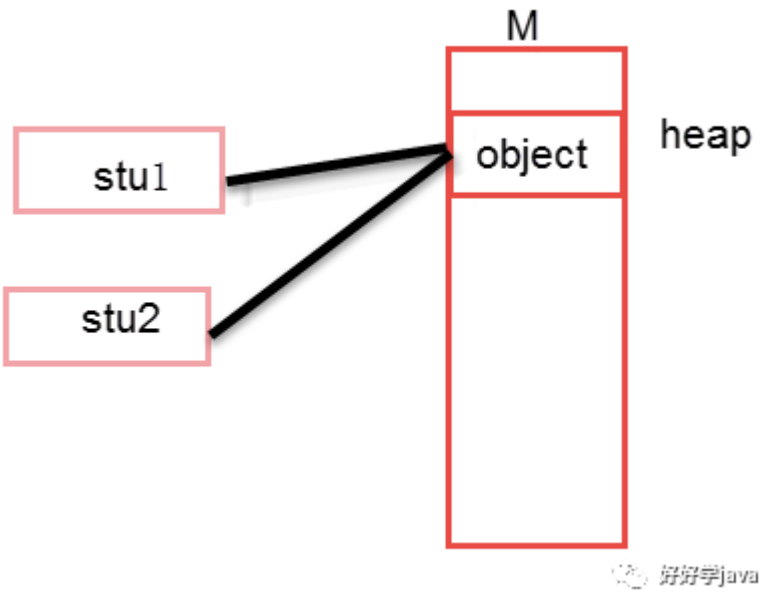
```
stu2.setNumber(54321);
System.out.println("学生 1:" + stu1.getNumber());
System.out.println("学生 2:" + stu2.getNumber());
```

结果：

```
学生 1:54321
学生 2:54321
```

这就怪了，为什么改变学生 2 的学号，学生 1 的学号也发生了变化呢？

原因出在(stu2 = stu1) 这一句。该语句的作用是将 stu1 的引用赋值给 stu2，这样，stu1 和 stu2 指向内存堆中同一个对象。如图：



那么，怎样才能达到复制一个对象呢？

是否记得万类之王 Object。它有 11 个方法，有两个 protected 的方法，其中一个为 clone 方法。

在 Java 中所有的类都是缺省的继承自 Java 语言包中的 Object 类的，查看它的源码，你可以把你的 JDK 目录下的 src.zip 复制到其他地方然后解压，里面就是所有的源码。发现里面有一个访问限定符为 protected 的方法 clone()：

```
/*
Creates and returns a copy of this object. The precise meaning of "copy" may
depend on the class of the object.
The general intent is that, for any object x, the expression:
1) x.clone() != x will be true
2) x.clone().getClass() == x.getClass() will be true, but these are not absolute
requirements.
3) x.clone().equals(x) will be true, this is not an absolute requirement.
*/
```

```
protected native Object clone() throws CloneNotSupportedException;
```

仔细一看，它还是一个 native 方法，大家都知道 native 方法是非 Java 语言实现的代码，供 Java 程序调用的，因为 Java 程序是运行在 JVM 虚拟机上面的，要想访问到比较底层的与操作系统相关的就没办法了，只能由靠近操作系统的语言来实现。

- 第一次声明保证克隆对象将有单独的内存地址分配。
- 第二次声明表明，原始和克隆的对象应该具有相同的类类型，但它不是强制性的。
- 第三声明表明，原始和克隆的对象应该是平等的 equals()方法使用，但它不是强制性的。

因为每个类直接或间接的父类都是 Object，因此它们都含有 clone()方法，但是因为该方法是 protected，所以都不能在类外进行访问。

要想对一个对象进行复制，就需要对 clone 方法覆盖。

为什么要克隆？

大家先思考一个问题，为什么需要克隆对象？直接 new 一个对象不行吗？

答案是：克隆的对象可能包含一些已经修改过的属性，而 new 出来的对象的属性都还是初始化时候的值，所以当需要一个新的对象来保存当前对象的“状态”就靠 clone 方法了。那么我把这个对象的临时属性一个一个的赋值给我新 new 的对象不也行嘛？可以是可以，但是一来麻烦不说，二来，大家通过上面的源码都发现了 clone 是一个 native 方法，就是快啊，在底层实现的。

提个醒，我们常见的 Object a=new Object();Object b;b=a;这种形式的代码复制的是引用，即对象在内存中的地址，a 和 b 对象仍然指向了同一个对象。

而通过 clone 方法赋值的对象跟原来的对象时同时独立存在的。

如何实现克隆

先介绍一下两种不同的克隆方法，**浅克隆(ShallowClone)**和**深克隆(DeepClone)**。

在 Java 语言中，数据类型分为值类型（基本数据类型）和引用类型，值类型包括 int、double、byte、boolean、char 等简单数据类型，引用类型包括类、接口、数组等复杂类型。浅克隆和深克隆的主要区别在于是否支持引用类型的成员变量的复制，下面将对两者进行详细介绍。

一般步骤是（浅克隆）：

1. 被复制的类需要实现 Clonenable 接口（不实现的话在调用 clone 方法会抛出 CloneNotSupportedException 异常），该接口为标记接口(不含任何方法)

2. 覆盖 clone()方法，访问修饰符设为 public。方法中调用 super.clone()方法得到需要的复制对象。（native 为本地方法)

下面对上面那个方法进行改造：

```
class Student implements Cloneable{
```

```
    private int number;
    public int getNumber() {
        return number;
    }
    public void setNumber(int number) {
        this.number = number;
    }
    @Override
    public Object clone() {
        Student stu = null;
        try{
            stu = (Student)super.clone();
        }catch(CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return stu;
    }
}
public class Test {
    public static void main(String args[]) {
        Student stu1 = new Student();
        stu1.setNumber(12345);
        Student stu2 = (Student)stu1.clone();
        System.out.println("学生 1:" + stu1.getNumber());
        System.out.println("学生 2:" + stu2.getNumber());
        stu2.setNumber(54321);
        System.out.println("学生 1:" + stu1.getNumber());
        System.out.println("学生 2:" + stu2.getNumber());
    }
}
```

结果：

```
学生 1:12345
学生 2:12345
学生 1:12345
学生 2:54321
```

如果你还不相信这两个对象不是同一个对象，那么你可以看看这一句：

```
System.out.println(stu1 == stu2); // false
```

上面的复制被称为浅克隆。

还有一种稍微复杂的深度复制：

我们在学生类里再加一个 Address 类。

```
class Address  {
    private String add;
    public String getAdd() {
        return add;
    }
    public void setAdd(String add) {
        this.add = add;
    }
}
class Student implements Cloneable{
    private int number;
    private Address addr;
    public Address getAddr() {
        return addr;
    }
    public void setAddr(Address addr) {
        this.addr = addr;
    }
    public int getNumber() {
        return number;
    }
    public void setNumber(int number) {
        this.number = number;
    }
    @Override
    public Object clone() {
        Student stu = null;
        try{
            stu = (Student)super.clone();
        }catch(CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return stu;
    }
}
public class Test {
    public static void main(String args[]) {
```

```
        Address addr = new Address();
        addr.setAdd("杭州市");
        Student stu1 = new Student();
        stu1.setNumber(123);
        stu1.setAddr(addr);
        Student stu2 = (Student)stu1.clone();
        System.out.println("学生 1:" + stu1.getNumber() + ",地址:" +
stu1.getAddr().getAdd());
        System.out.println("学生 2:" + stu2.getNumber() + ",地址:" +
stu2.getAddr().getAdd());
    }
}
```

结果：

```
学生 1:123,地址:杭州市
学生 2:123,地址:杭州市
```

乍一看没什么问题，真的是这样吗？

我们在 main 方法中试着改变 addr 实例的地址。

```
addr.setAdd("西湖区");
System.out.println("学生 1:" + stu1.getNumber() + ",地址:" +
stu1.getAddr().getAdd());
System.out.println("学生 2:" + stu2.getNumber() + ",地址:" +
stu2.getAddr().getAdd());
```

结果：

```
学生 1:123,地址:杭州市
学生 2:123,地址:杭州市
学生 1:123,地址:西湖区
学生 2:123,地址:西湖区
```

这就奇怪了，怎么两个学生的地址都改变了？

原因是浅复制只是复制了 addr 变量的引用，并没有真正的开辟另一块空间，将值复制后再将引用返回给新对象。

所以，为了达到真正的复制对象，而不是纯粹引用复制。我们需要将 Address 类可复制化，并且修改 clone 方法，完整代码如下：

```
class Address implements Cloneable {
    private String add;
    public String getAdd() {
        return add;
    }
    public void setAdd(String add) {
        this.add = add;
```

```
    }
    @Override
    public Object clone() {
        Address addr = null;
        try{
            addr = (Address)super.clone();
        }catch(CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return addr;
    }
}
class Student implements Cloneable{
    private int number;
    private Address addr;
    public Address getAddr() {
        return addr;
    }
    public void setAddr(Address addr) {
        this.addr = addr;
    }
    public int getNumber() {
        return number;
    }
    public void setNumber(int number) {
        this.number = number;
    }
    @Override
    public Object clone() {
        Student stu = null;
        try{
            stu = (Student)super.clone();   //浅复制
        }catch(CloneNotSupportedException e) {
            e.printStackTrace();
        }
        stu.addr = (Address)addr.clone();   //深度复制
        return stu;
    }
}
public class Test {
```

```
    public static void main(String args[]) {
        Address addr = new Address();
        addr.setAdd("杭州市");
        Student stu1 = new Student();
        stu1.setNumber(123);
        stu1.setAddr(addr);
        Student stu2 = (Student)stu1.clone();
        System.out.println("学生 1:" + stu1.getNumber() + ",地址:" +
stu1.getAddr().getAdd());
        System.out.println("学生 2:" + stu2.getNumber() + ",地址:" +
stu2.getAddr().getAdd());
        addr.setAdd("西湖区");
        System.out.println("学生 1:" + stu1.getNumber() + ",地址:" +
stu1.getAddr().getAdd());
        System.out.println("学生 2:" + stu2.getNumber() + ",地址:" +
stu2.getAddr().getAdd());
    }
}
```

结果：

```
学生 1:123,地址:杭州市
学生 2:123,地址:杭州市
学生 1:123,地址:西湖区
学生 2:123,地址:杭州市
```

这样结果就符合我们的想法了。

最后我们可以看看 API 里其中一个实现了 clone 方法的类：

```
java.util.Date:
/**
 * Return a copy of this object.
 */
public Object clone() {
    Date d = null;
    try {
        d = (Date)super.clone();
        if (cdate != null) {
            d.cdate = (BaseCalendar.Date) cdate.clone();
        }
    } catch (CloneNotSupportedException e) {} // Won't happen
    return d;
}
```

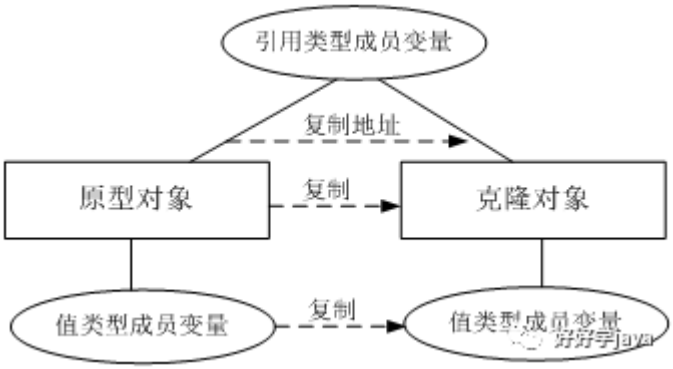
该类其实也属于深度复制。

浅克隆和深克隆

1、浅克隆

在浅克隆中，如果原型对象的成员变量是值类型，将复制一份给克隆对象；如果原型对象的成员变量是引用类型，则将引用对象的地址复制一份给克隆对象，也就是说原型对象和克隆对象的成员变量指向相同的内存地址。

简单来说，在浅克隆中，当对象被复制时只复制它本身和其中包含的值类型的成员变量，而引用类型的成员对象并没有复制。

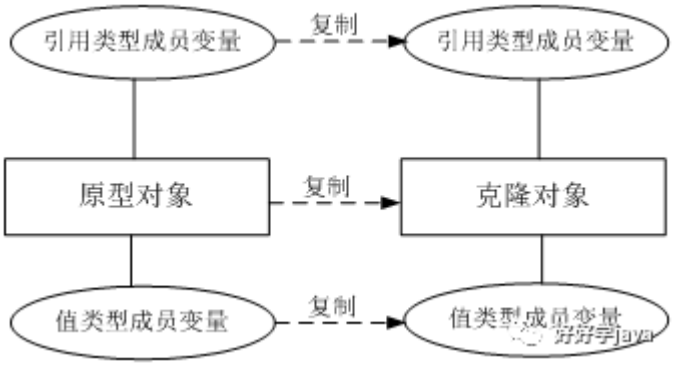


在 Java 语言中，通过覆盖 Object 类的 clone()方法可以实现浅克隆。

2、深克隆

在深克隆中，无论原型对象的成员变量是值类型还是引用类型，都将复制一份给克隆对象，深克隆将原型对象的所有引用对象也复制一份给克隆对象。

简单来说，在深克隆中，除了对象本身被复制外，对象所包含的所有成员变量也将复制。



在 Java 语言中，如果需要**实现深克隆，可以通过覆盖 Object 类的 clone()方法实现，也可以通过序列化(Serialization)等方式**来实现。

（如果引用类型里面还包含很多引用类型，或者内层引用类型的类里面又包含引用类型，使用 clone 方法就会很麻烦。这时我们可以用序列化的方式来实现对象的深克隆。）

序列化就是将对象写到流的过程，写到流中的对象是原有对象的一个拷贝，而原对象仍然存在于内存中。通过序列化实现的拷贝不仅可以复制对象本身，而且可以复制其引用的成员对象，因此通过序列化将对象写到一个流中，再从流里将其读出来，可以实现深克隆。需要注意的是能够实现序列化的对象其类必须实现 Serializable 接口，否则无法实现序列化操

作。

扩展: Java 语言提供的 Cloneable 接口和 Serializable 接口的代码非常简单，它们都是空接口，这种空接口也称为标识接口，标识接口中没有任何方法的定义，其作用是告诉 JRE 这些接口的实现类是否具有某个功能，如是否支持克隆、是否支持序列化等。

解决多层克隆问题

如果引用类型里面还包含很多引用类型，或者内层引用类型的类里面又包含引用类型，使用 clone 方法就会很麻烦。这时我们可以用序列化的方式来实现对象的深克隆。

```
public class Outer implements Serializable{

  private static final long serialVersionUID = 369285298572941L;  //最好是显式声明ID

  public Inner inner;

  //Discription:[深度复制方法,需要对象及对象所有的对象属性都实现序列化]

  public Outer myclone() {

      Outer outer = null;

      try { // 将该对象序列化成流,因为写在流里的是对象的一个拷贝，而原对象仍然存在于 JVM 里面。所以利用这个特性可以实现对象的深拷贝

          ByteArrayOutputStream baos = new ByteArrayOutputStream();

          ObjectOutputStream oos = new ObjectOutputStream(baos);

          oos.writeObject(this);

            // 将流序列化成对象

          ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());

          ObjectInputStream ois = new ObjectInputStream(bais);

          outer = (Outer) ois.readObject();

      } catch (IOException e) {

          e.printStackTrace();

      } catch (ClassNotFoundException e) {

          e.printStackTrace();

      }

      return outer;

  }

}
```

Inner 也必须实现 Serializable，否则无法序列化：

```
public class Inner implements Serializable{

  private static final long serialVersionUID = 872390113109L; //最好是显式声明 ID

  public String name = "";

  public Inner(String name) {
```

```
        this.name = name;
    }
    @Override
    public String toString() {
        return "Inner 的 name 值为：" + name;
    }
}
```

这样也能使两个对象在内存空间内完全独立存在，互不影响对方的值。

总结

实现对象克隆有两种方式：

1). 实现 Cloneable 接口并重写 Object 类中的 clone()方法；

2). 实现 Serializable 接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆。

注意：基于序列化和反序列化实现的克隆不仅仅是深度克隆，更重要的是通过泛型限定，可以检查出要克隆的对象是否支持序列化，这项检查是编译器完成的，不是在运行时抛出异常，这种是方案明显优于使用 Object 类的 clone 方法克隆对象。让问题在编译的时候暴露出来总是优于把问题留到运行时。