创建型模式（三）

2020年3月30日　　10:37

创建型模式（三）

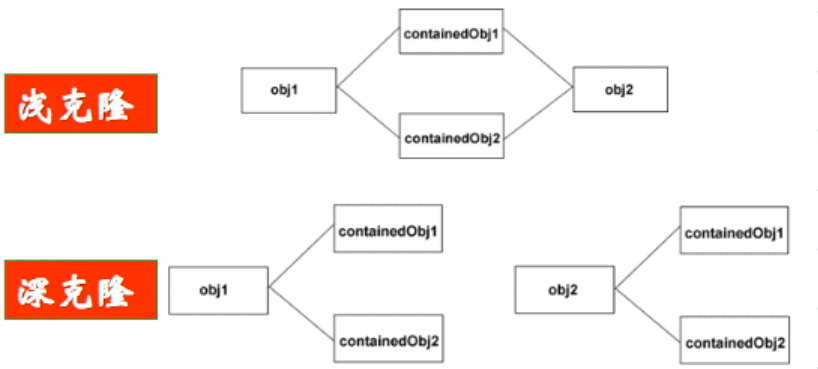
作业要求：

1、设计一个客户类 Customer，其中客户地址存储在地址类 Address中，用浅克隆和深克隆分别实现 Customer对象的复制并比较这两种克隆方式的异同。绘制类图并编程实现。

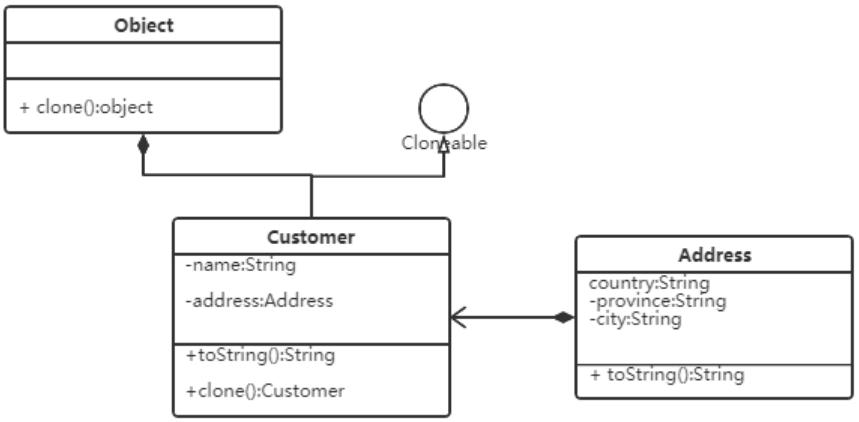
2、使用单例模式的思想实现多例模式，确保系统中某个类的对象只能存在有限个，如两个或三个，设计并编写代码实现一个多例类。

- 作业与笔记github地址：
  https://github.com/baobaotql/CCNU_DesignPattern

- **Q1：设计一个客户类 Customer，其中客户地址存储在地址类 Address中，用浅克隆和深克隆分别实现 Customer对象的复制并比较这两种克隆方式的异同。绘制类图并编程实现。**
- **A1：**
  - **原型模式分析**
    通常情况下，一个类包含一些成员对象，在使用原型模式克隆对象时，根据其成员对象是否也克隆，原型模式可以分为两种形式：深克隆与浅克隆
    【注】复制时只需要复制非静态成员



  - **浅克隆**
    在浅克隆中，被复制对象的所有普通成员变量都具有与原来对象相同的值，而所有的对其他对象的引用仍然指向原来的对象。也就是说，浅克隆仅仅复制所考虑的对象，不会复制它所引用的成员对象。模式图与代码解析如下：



- **地址类**

```
public class Address {

    private String country;

    private String province;

    private String city;

    public String getCountry() {

        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }

    public String getProvince() {

        return province;
    }

    public void setProvince(String province) {
        this.province = province;
    }

    public String getCity() {

        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }


    public Address(String country, String province, String city) {
        super();
        this.country = country;
        this.province = province;
        this.city = city;
    }
    @Override

    public String toString() {

        return "Address [city=" + city + ", country=" + country + ", p
rovince="
                + province + "]";
    }


}
```

- **客户类**

```
public class Customer implements Cloneable{
    private String name;
    private Address address;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        this.address = address;
    }
    public Customer(String name, Address address) {
        super();
        this.name = name;
        this.address = address;
    }

    @Override
    public String toString() {
        return "Customer [address=" + address + ", name=" + name + "]"
;
    }
    public Customer clone(){
        Customer clone=null;
        try {
            clone=(Customer) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return clone;
    }


}
```

- **测试类**

```
package pagebaobaotql;


public class Test {
    public static void main(String[] args) {
        Address address=new Address("SC","CD","WHQ");
        Customer c1=new Customer("sqq", address);
        Customer c2=c1.clone();
        c2.getAddress().setCity("MY");
        System.out.println("c1"+c1.toString());
```

```
        System.out.println("c2"+c2.toString());
    }
}
```

- **深克隆**

  在深克隆中被复制的对象的所有普通成员变量也都含有与原来的对象相同的值，出去那些引用其他对象的变量。换言之，在深克隆中，除了对象本身被复制外，对象包含的引用也被复制，也就是其中的成员对象也被复制。

  序列化（Serialization）就是将对象写到流的过程，写到流中的对象是原有的对象的一个拷贝，而原对象仍存在内存中。通过序列化实现的拷贝不仅可以复制对象本身，而且也可以复制其引用的成员对象，因此通过序列化将对象写入一个流中，再从流中将其读出来，从而实现深克隆。

  - 修改Customer类

```
public class Customer implements Serializable{
    private Address address;
    private String name;

    public String getName() {
        return name;
    }

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

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    public Customer(String name, Address address) {
        super();
        this.name = name;
        this.address = address;
    }

    @Override
    public String toString() {
        return "Customer [address=" + address + ", name=" + name + "]"
;
    }
    public Object deepClone() throws Exception {
```

```
            //将对象写入流中
            ByteArrayOutputStream bao=new ByteArrayOutputStream();
            ObjectOutputStream oos=new ObjectOutputStream(bao);
            oos.writeObject(this);
            //将对象从流中取出
            ByteArrayInputStream bis=new ByteArrayInputStream(bao.toByteA
rray());
            ObjectInputStream ois=new ObjectInputStream(bis);
            return(ois.readObject());
        }
}
```

- **测试类**

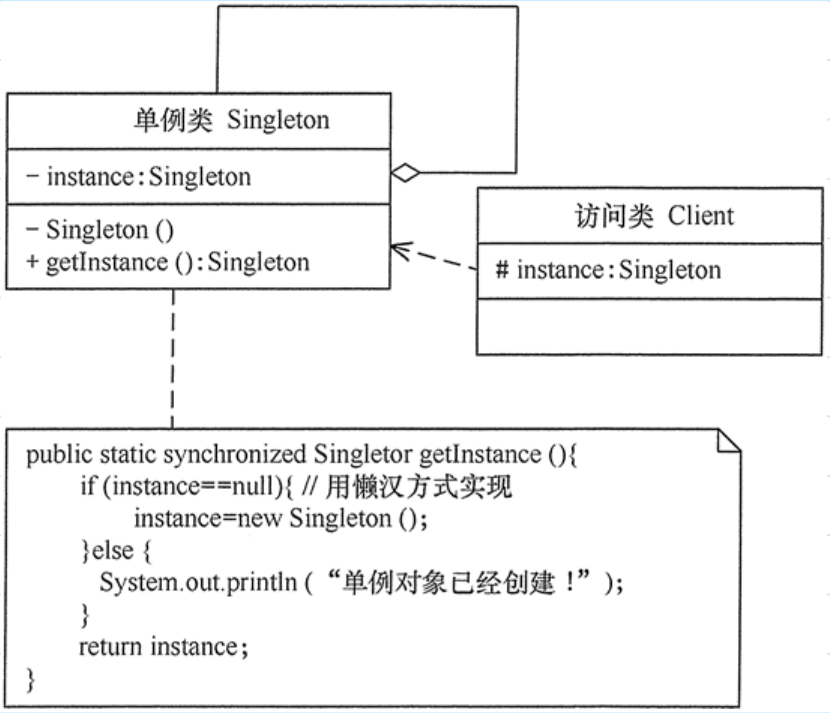
```
package pagebaobaotql;


public class Test {
    public static void main(String[] args) {
        Address address=new Address("SC","CD","WHQ");
        Customer c1=new Customer("sqq", address);
        Customer c2=c1.clone();
        c2.getAddress().setCity("MY");
        System.out.println("c1"+c1.toString());
        System.out.println("c2"+c2.toString());
    }
}
```

- **Q2：使用单例模式的思想实现多例模式，确保系统中某个类的对象只能存在有限个，如两个或三个，设计并编写代码实现一个多例类。**
- **A2：**
  - **单例模式**

    指一个类只有一个实例，且该类能自行创建这个实例的一种模式。例如，Windows 中只能打开一个任务管理器，这样可以避免因打开多个任务管理器窗口而造成内存资源的浪费，或出现各个窗口显示内容的不一致等错误。

    单例模式是**设计模式**中最简单的模式之一。通常，普通类的构造函数是公有的，外部类可以通过“new 构造函数( )”来生成多个实例。但是，如果将类的构造函数设为私有的，外部类就无法调用该构造函数，也就无法生成多个实例。这时该类自身必须定义一个静态私有实例，并向外提供一个静态的公有函数用于创建或获取该静态私有实例。

    模式结构图



- **代码解析**

```
package Singleton;

import java.util.*;

public class Multiton {

    //两个或者三个

    private static ArrayList<Multiton> multitonList = new ArrayList<>
();

    private final static int NUMBER=3;


    private Multiton(){

    }

    public static Multiton getInstance() {

        // TODO: implement

        if (multitonList.size()<NUMBER){

            System.out.println("创建新实例成功");

            Multiton current =  new Multiton();
```

```
                multitonList.add(current);

                return current;

            }else{

                System.out.println("不能创建更多的实例");

                return multitonList.get(new Random().nextInt(multitonList
.size()));

            }

        }

}
```

- **客户端类Client**

```
package Singleton;

public class Client {

    public static void main(String[] args) {

        Multiton m1 = Multiton.getInstance();

        Multiton m2 = Multiton.getInstance();

        Multiton m3 = Multiton.getInstance();

        Multiton m4 = Multiton.getInstance();


        System.out.println("m1==m2:"+(m1==m2));

        System.out.println("m2==m3:"+(m2==m4));

        System.out.println("m3==m4:"+(m3==m4));

    }

}
```