## 克隆和序列化应用 + 面试题

### 克隆

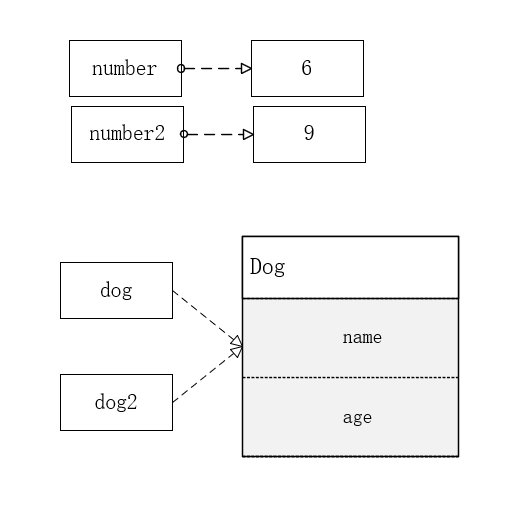
在开始学习克隆之前，我们先来看看下面的代码，普通的对象复制，存在什么问题？

class CloneTest {  
 public static void main(String[] args) throws CloneNotSupportedException {  
 // 等号赋值（ 基本类型）  
 int number = 6;  
 int number2 = number;  
 // 修改 number2 的值  
 number2 = 9;  
 System.out.println("number：" + number);  
 System.out.println("number2：" + number2);  
 // 等号赋值（对象）  
 Dog dog = new Dog();  
 dog.name = "旺财";  
 dog.age = 5;  
 Dog dog2 = dog;  
 // 修改 dog2 的值  
 dog2.name = "大黄";  
 dog2.age = 3;  
 System.out.println(dog.name + "，" + dog.age + "岁");  
 System.out.println(dog2.name + "，" + dog2.age + "岁");  
 }  
}

程序执行结果：

number：6  
number2：9  
大黄，3岁  
大黄，3岁

可以看出，如果使用等号复制时，对于值类型来说，彼此之间的修改操作是相对独立的，而对于引用类型来说，因为复制的是引用对象的内存地址，所以修改其中一个值，另一个值也会跟着变化，原理如下图所示：



因此为了防止这种问题的发生，就要使用对象克隆来解决引用类型复制的问题。

#### 1）浅克隆

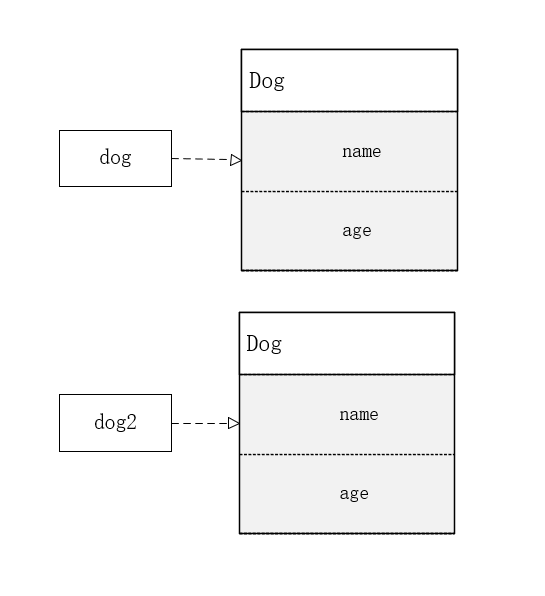
默认的 clone() 方法，为浅克隆，代码如下：

class CloneTest {  
 public static void main(String[] args) throws CloneNotSupportedException {  
 Dog dog = new Dog();  
 dog.name = "旺财";  
 dog.age = 5;  
 // 克隆  
 Dog dog3 = (Dog) dog.clone();  
 dog3.name = "小白";  
 dog3.age = 2;  
 System.out.println(dog.name + "，" + dog.age + "岁");  
 System.out.println(dog3.name + "，" + dog3.age + "岁");  
 }  
}  
class Dog implements Cloneable {  
 public String name;  
 public int age;  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
}

程序执行结果：

旺财，5岁  
小白，2岁

可以看出使用克隆就可以解决引用类型复制的问题了，原理如下图所示：



以上这种复制方式叫做 **浅克隆。**

**浅克隆的实现条件** ：需要克隆的对象必须实现 Cloneable 接口，并重写 clone() 方法，即可实现对此对象的克隆。

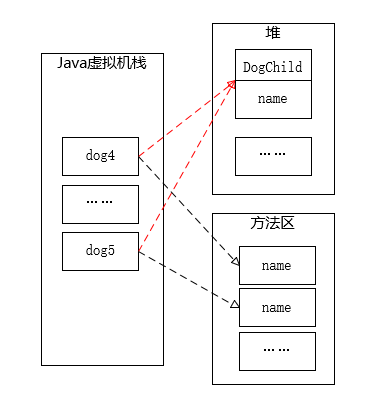
然而 **使用浅克隆也会存在一个问题** ，请参考以下代码。

class CloneTest {  
 public static void main(String[] args) throws CloneNotSupportedException {  
 DogChild dogChild = new DogChild();  
 dogChild.name = "二狗";  
 Dog dog4 = new Dog();  
 dog4.name = "大黄";  
 dog4.dogChild = dogChild;  
 Dog dog5 = (Dog) dog4.clone();  
 dog5.name = "旺财";  
 dog5.dogChild.name = "狗二";  
 System.out.println("dog name 4："+dog4.name);  
 System.out.println("dog name 5："+dog5.name);  
 System.out.println("dog child name 4："+dog4.dogChild.name);  
 System.out.println("dog child name 5："+dog5.dogChild.name);  
 }  
}  
class Dog implements Cloneable {  
 public String name;  
 public DogChild dogChild;  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
}  
class DogChild {  
 public String name;  
}

程序执行结果：

dog name 4：大黄  
dog name 5：旺财  
dog child name 4：狗二  
dog child name 5：狗二

也就是说浅克隆，只会复制对象的值类型，而不会复制对象的引用类型。原因如下图所示：



enter image description here

要处理引用类型不被复制的问题，就要使用到 **深克隆** 。

#### 2）深克隆

**定义** ：深克隆就是复制整个对象信息，包含值类型和引用类型。

**深克隆的实现方式** 通常包含以下两种。

* 序列化实现深克隆：先将原对象序列化到内存的字节流中，再从字节流中反序列化出刚刚存储的对象，这个新对象和原对象就不存在任何地址上的共享，这样就实现了深克隆。
* 所有引用类型都实现克隆：要复制对象的所有引用类型都要实现克隆，所有对象都是复制的新对象，从而实现了深克隆。

**深克隆实现方式一：序列化**

实现思路：先将要拷贝对象写入到内存中的字节流中，然后再从这个字节流中读出刚刚存储的信息，作为一个新对象返回，那么这个新对象和原对象就不存在任何地址上的共享，自然实现了深拷贝。请参考以下代码：

class CloneTest {  
 public static void main(String[] args) throws CloneNotSupportedException {  
 BirdChild birdChild = new BirdChild();  
 birdChild.name = "小小鸟";  
 Bird bird = new Bird();  
 bird.name = "小鸟";  
 bird.birdChild = birdChild;  
 // 使用序列化克隆对象  
 Bird bird2 = CloneUtils.clone(bird);  
 bird2.name = "黄雀";  
 bird2.birdChild.name = "小黄雀";  
 System.out.println("bird name:" + bird.name);  
 System.out.println("bird child name:" + bird.birdChild.name);  
 System.out.println("bird name 2:" + bird2.name);  
 System.out.println("bird child name 2:" + bird2.birdChild.name);  
 }  
}  
class CloneUtils {  
 public static <T extends Serializable> T clone(T obj) {  
 T cloneObj = null;  
 try {  
 //写入字节流  
 ByteArrayOutputStream bo = new ByteArrayOutputStream();  
 ObjectOutputStream oos = new ObjectOutputStream(bo);  
 oos.writeObject(obj);  
 oos.close();  
 //分配内存,写入原始对象,生成新对象  
 ByteArrayInputStream bi = new ByteArrayInputStream(bo.toByteArray());//获取上面的输出字节流  
 ObjectInputStream oi = new ObjectInputStream(bi);  
 //返回生成的新对象  
 cloneObj = (T) oi.readObject();  
 oi.close();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 return cloneObj;  
 }  
}

程序执行结果：

bird name:小鸟  
bird child name:小小鸟  
bird name 2:黄雀  
bird child name 2:小黄雀

**深克隆实现方式二：所有引用类型都实现克隆**

class SerializableTest {  
 public static void main(String[] args) throws IOException, ClassNotFoundException {  
 ParrotChild parrotChild = new ParrotChild();  
 parrotChild.name = "小鹦鹉";  
 Parrot parrot = new Parrot();  
 parrot.name = "大鹦鹉";  
 parrot.parrotChild = parrotChild;  
 // 克隆  
 Parrot parrot2 = (Parrot) parrot.clone();  
 parrot2.name = "老鹦鹉";  
 parrot2.parrotChild.name = "少鹦鹉";  
 System.out.println("parrot name:" + parrot.name);  
 System.out.println("parrot child name:" + parrot.parrotChild.name);  
 System.out.println("parrot name 2:" + parrot2.name);  
 System.out.println("parrot child name 2:" + parrot2.parrotChild.name);  
 }  
 }  
class Parrot implements Cloneable {  
 public String name;  
 public ParrotChild parrotChild;  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 Parrot bird = (Parrot) super.clone();  
 bird.parrotChild = (ParrotChild) parrotChild.clone();  
 return bird;  
 }  
}  
class ParrotChild implements Cloneable {  
 public String name;  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
}

程序执行结果：

parrot name:大鹦鹉  
parrot child name:小鹦鹉  
parrot name 2:老鹦鹉  
parrot child name 2:少鹦鹉

### 序列化和反序列化

#### 1）介绍

内存中的数据对象只有转换成二进制流才能进行数据持久化或者网络传输，将对象转换成二进制流的过程叫做序列化（Serialization）；相反，把二进制流恢复为数据对象的过程就称之为反序列化（Deserialization）。

#### 2）序列化和反序列代码实现

先把对象序列化到磁盘，再从磁盘中反序列化出对象，请参考以下代码：

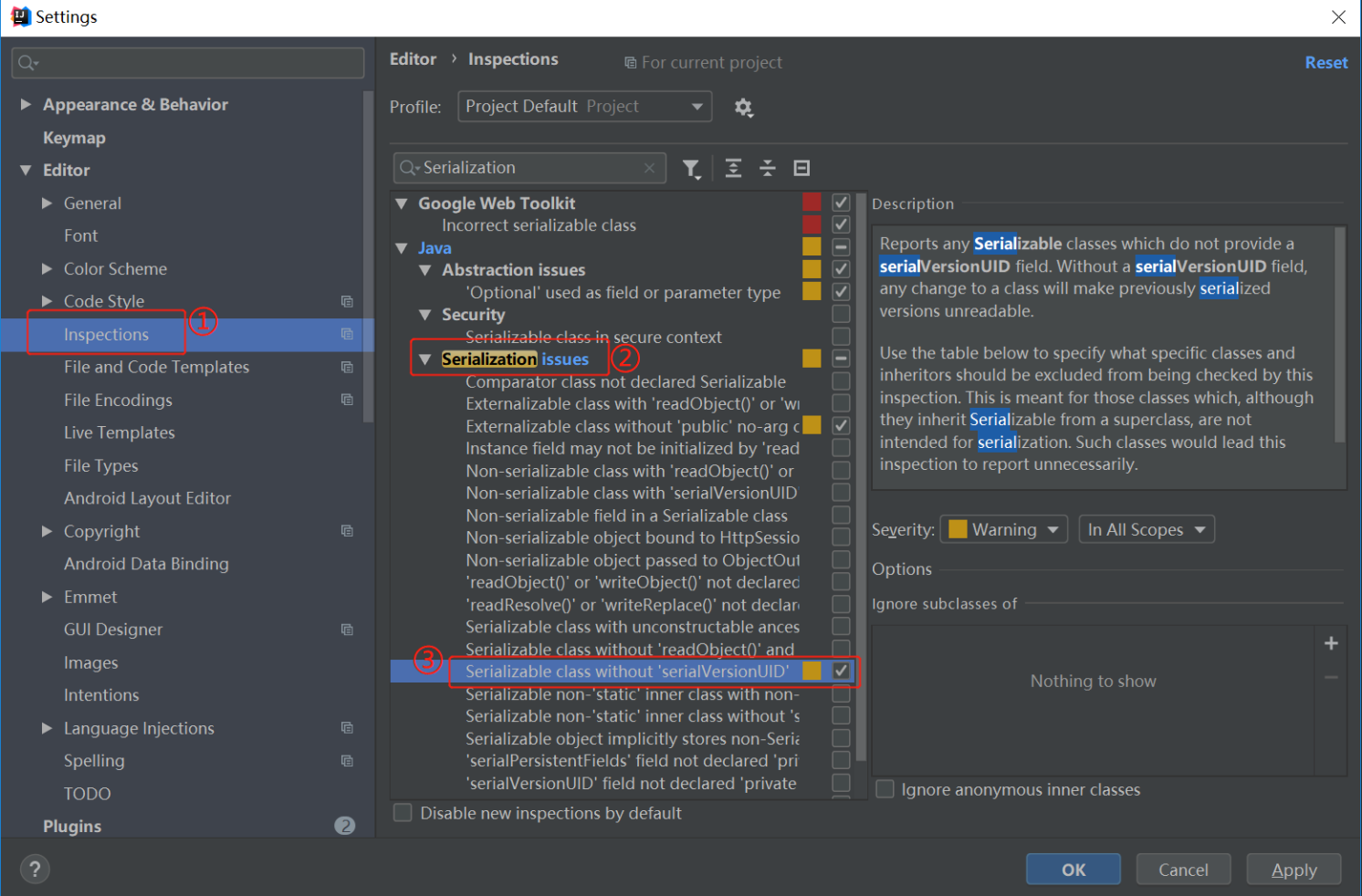
class SerializableTest {  
 public static void main(String[] args) throws IOException, ClassNotFoundException {  
 // 对象赋值  
 User user = new User();  
 user.setName("老王");  
 user.setAge(30);  
 System.out.println(user);  
 // 创建输出流（序列化内容到磁盘）  
 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("test.out"));  
 // 序列化对象  
 oos.writeObject(user);  
 oos.flush();  
 oos.close();  
 // 创建输入流（从磁盘反序列化）  
 ObjectInputStream ois = new ObjectInputStream(new FileInputStream("test.out"));  
 // 反序列化  
 User user2 = (User) ois.readObject();  
 ois.close();  
 System.out.println(user2);  
 }  
}  
class User implements Serializable {  
 private static final long serialVersionUID = 3831264392873197003L;  
 private String name;  
 private int age;  
 @Override  
 public String toString() {  
 return "{name:" + name + ",age:" + age + "}";  
 }  
 // setter/getter...  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public int getAge() {  
 return age;  
 }  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

程序执行结果：

{name:老王,age:30}  
{name:老王,age:30}

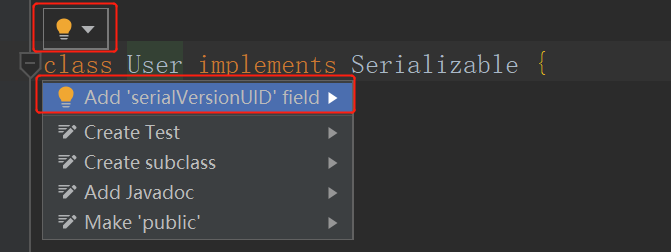
更多序列化和反序列化的实现方式以及代码示例，请看下文面试部分的内容。

**开发工具设置** ：IDEA 开启自动生成 serialVersionUID  
点击 Settings → Inspections → 搜索 Serialization issues → 勾选 Serializable class without ‘SerialVersionUID’ 保存设置，如下图所示：



enter image description here

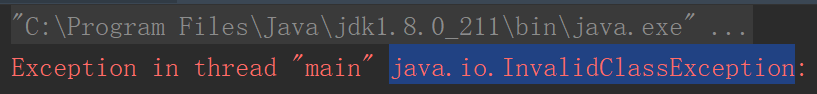
设置完之后，光标放到类名上，点击提示，生成 serialVersionUID，如下图所示：



### 相关面试题

#### 1.serialVersionUID 的作用是什么？

答：如果显示定义了 serialVersionUID 值之后，可以使序列化和反序列化向后兼容。也就是说如果 serialVersionUID 的值相同，修改对象的字段（删除或增加），程序不会报错，之后给没有的字段赋值为 null，而如果没有指定 serialVersionUID 的值，如果修改对象的字段，程序就会报错。如下图所示：



#### 2.可序列化接口（Serializalbe）的用途是什么？

答：可序列化 Serializalbe 接口存在于 java.io 包中，构成了 Java 序列化机制的核心，它没有任何方法，它的用途是标记某对象为可序列化对象，指示编译器使用 Java 序列化机制序列化此对象。

#### 3.常用的序列化方式都有哪些？

答：常用的序列化有以下三种方式：

**1）Java 原生序列化方式**

请参考以下代码：

// 序列化和反序列化  
class SerializableTest {  
 public static void main(String[] args) throws IOException, ClassNotFoundException {  
 // 对象赋值  
 User user = new User();  
 user.setName("老王");  
 user.setAge(30);  
 System.out.println(user);  
 // 创建输出流（序列化内容到磁盘）  
 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("test.out"));  
 // 序列化对象  
 oos.writeObject(user);  
 oos.flush();  
 oos.close();  
 // 创建输入流（从磁盘反序列化）  
 ObjectInputStream ois = new ObjectInputStream(new FileInputStream("test.out"));  
 // 反序列化  
 User user2 = (User) ois.readObject();  
 ois.close();  
 System.out.println(user2);  
 }  
}  
class User implements Serializable {  
 private static final long serialVersionUID = 5132320539584511249L;  
 private String name;  
 private int age;  
 @Override  
 public String toString() {  
 return "{name:" + name + ",age:" + age + "}";  
 }  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public int getAge() {  
 return age;  
 }  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

**2）JSON 格式，可使用 fastjson 或 GSON**

JSON 是一种轻量级的数据格式，JSON 序列化的优点是可读性比较高，方便调试。我们本篇以 fastjson 的序列化为例，请参考以下代码：

// 序列化和反序列化  
class SerializableTest {  
 public static void main(String[] args) throws IOException, ClassNotFoundException {  
 // 对象赋值  
 User user = new User();  
 user.setName("老王");  
 user.setAge(30);  
 System.out.println(user);  
  
 String jsonSerialize = JSON.toJSONString(user);  
 User user3 = (User) JSON.parseObject(jsonSerialize, User.class);  
 System.out.println(user3);  
 }  
}  
class User implements Serializable {  
 private static final long serialVersionUID = 5132320539584511249L;  
 private String name;  
 private int age;  
 @Override  
 public String toString() {  
 return "{name:" + name + ",age:" + age + "}";  
 }  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public int getAge() {  
 return age;  
 }  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

**3）Hessian 方式序列化**

Hessian 序列化的优点是可以跨编程语言，比 Java 原生的序列化和反序列化效率高。  
请参考以下示例代码：

// 序列化和反序列化  
class SerializableTest {  
 public static void main(String[] args) throws IOException, ClassNotFoundException {  
 // 序列化  
 ByteArrayOutputStream bo = new ByteArrayOutputStream();  
 HessianOutput hessianOutput = new HessianOutput(bo);  
 hessianOutput.writeObject(user);  
 byte[] hessianBytes = bo.toByteArray();  
 // 反序列化  
 ByteArrayInputStream bi = new ByteArrayInputStream(hessianBytes);  
 HessianInput hessianInput = new HessianInput(bi);  
 User user4 = (User) hessianInput.readObject();  
 System.out.println(user4);  
 }  
}  
class User implements Serializable {  
 private static final long serialVersionUID = 5132320539584511249L;  
 private String name;  
 private int age;  
 @Override  
 public String toString() {  
 return "{name:" + name + ",age:" + age + "}";  
 }  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public int getAge() {  
 return age;  
 }  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

#### 4.使用克隆有什么好处？

答：好处包含以下几点。

* 使用方便：假如要复制一个对象，但这个对象中的部分属性已经被修改过了，如果不使用克隆的话，需要给属性手动赋值，相比克隆而已麻烦很多；
* 性能高：查看 clone 方法可以知道，它是 native 方法，native 方法是原生函数，使用操作系统底层的语言实现的，因此执行效率更高；
* 隔离性：克隆可以确保对象操作时相互隔离。

clone() 源代码，如下图：

enter image description here

enter image description here

#### 5.浅克隆和深克隆有什么区别？

答：区别主要在对引用类型的复制上，具体信息如下。

* 浅克隆：只会复制对象的值类型，而不会复制对象的引用类型；
* 深克隆：复制整个对象，包含值类型和引用类型。

#### 6.如何实现浅克隆？

答：克隆的对象实现 Cloneable 接口，并重写 clone() 方法就可以实现浅克隆了。

#### 7.以下代码执行的结果是？

import java.util.Arrays;  
class CloneTest {  
 public static void main(String[] args) throws CloneNotSupportedException {  
 CloneObj cloneObj = new CloneObj();  
 cloneObj.name = "老王";  
 cloneObj.age = 30;  
 cloneObj.sistersAge = new int[]{18, 19};  
 CloneObj cloneObj2 = (CloneObj) cloneObj.clone();  
 cloneObj2.name = "磊哥";  
 cloneObj2.age = 33;  
 cloneObj2.sistersAge[0] = 20;  
 System.out.println(cloneObj.name + "|" + cloneObj2.name);  
 System.out.println(cloneObj.age + "|" + cloneObj2.age);  
 System.out.println(Arrays.toString(cloneObj.sistersAge) + "|" + Arrays.toString(cloneObj2.sistersAge));  
 }  
}  
class CloneObj implements Cloneable {  
 public String name;  
 public int age;  
 public int[] sistersAge;  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
}

答：执行结果如下。

老王|磊哥  
30|33  
[20, 19]|[20, 19]

#### 8.深克隆如何实现？有几种实现方式？

答：一般实现方式有两种。

* 通过序列化实现深克隆（序列化实现方式：Java 原生序列化、JSON 序列化、Hessian 序列化）；
* 所有引用类型都实现克隆，从而实现深克隆。

#### 9.为什么不能直接使用 Object 的 Clone 方法，还要重写 clone() 方法之后才能实现克隆？

答：虽然所有类都是 Object 的子类，但因为 Object 中的 clone() 方法被声明为 protected 访问级别，所以非 java.lang 包下的其他类是不能直接使用的。因此要想实现克隆功能，就必须实现 Cloneable，并重写 clone() 方法才行。

#### 10.序列化可不可以实现深克隆？实现的原理是什么？

答：先将原对象序列化到内存的字节流中，再从字节流中反序列化出刚刚存储的对象，这个新对象和原对象就不存在任何地址上的共享，这样就实现了深克隆。

#### 11.序列化时某些成员不需要序列化，如何实现？

答：可以把不需要序列化的成员设置为瞬态（trasient）和静态变量，这样就不会被序列化了，瞬态的使用如下：

public transient int num;

#### 12.是否可以自定义序列化过程，覆盖 Java 中的默认序列化过程？

答：可以，在 Java 中默认序列化一个对象需要调用 ObjectOutputStream.writeObject(saveThisObject) 和 ObjectInputStream.readObject() 读取对象，你可以自定义这两个方法，从而实现自定义序列化的过程。需要注意的重要一点是，记得声明这些方法为私有方法，以避免被继承、重写或重载。

#### 13.在 Java 中的序列化和反序列化过程中使用了哪些方法？

答：在 Java 中序列化由 java.io.ObjectOutputStream 类完成，该类是一个筛选器流，它封装在较低级别的字节流中，以处理序列化机制。要通过序列化机制存储任何对象，我们需要调用 ObjectOutputStream.writeObject(savethisobject) 方法，如果要反序列化该对象，我们需要调用 ObjectInputStream.readObject() 方法，readObject() 方法会读取字节，并把这些字节转换为对象再返回。

### 总结

序列化常见的使用场景是远程服务调用（RPC）和网络对象传输等，可通过 implements Serializable 来实现对象序列化，在序列化对象中通过定义 serialVersionUID 来防止执行不兼容的类更改。调用 Object 类中的 clone() 方法默认是浅克隆，浅克隆只能复制值类型，不能复制引用类型，因此更多的时候我们需要深克隆，深克隆通常的实现方式有两种：序列化和所有引用类型都实现克隆。

[点击此处下载本文源码](https://github.com/vipstone/java-%20interview/tree/master/interview-code/src/main/java/com/interview)

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持