Markup interfaces are interfaces without attributes and methods, and they are also a design idea. A tag interface in Java represents the characteristics of a class, and a class that implements the tag interface has the characteristics. if a class that implements the Serializable interface is implemented, the object of this class can be serialized and deserialized. The common tag interfaces in Java include Cloneable interface, Random Access interface and Remote interface. if (object name instanceof tag interface name) can be used to detect whether a class implements a tag interface.

A Brief Analysis of the Four Marker Interfaces

1. java.io.Serializable Markup Interface

The interface is used to mark whether the object of the class can be serialized or serialized. After serialization, objects can be stored persistently and transmitted over the network. If an object is written into a file as a byte stream, it is a serialization process, which implements the persistent storage of the object. Then your program can read the serialized object from the file and restore it to the original object for deserialization. If the object of the serialized class does not implement the Serializable interface, the NotSerializable Exception is thrown.

1) What are the characteristics of object serialization? When an object is serialized, the instance variable referenced by the object is serialized, and if the instance variable is an object, the object is serialized. Instance variables modified by transients also do not participate in serialization. When the object is deserialized, the variables modified by transients are reinitialized. The object is initially null, and the basic data types are initialized to 0, false equivalents. Static-modified static variables also do not participate in the serialization process. When de-serialized, its value is the value of the variable in the runtime virtual machine, not the value "stored" when the object is serialized.

2) What is serial Version UID and what is its use? When an object is serialized, the object will have a serialVersionUID, which is computed based on the structure information of the class. In deserialization, if the object has a different serialVersionUID, the virtual machine will throw an exception and the restore operation will fail. For example, the following attributes can be found in the source code of the ArrayList class (JDK8), so the objects of the ArrayList class in JDK8 have a consistent serialVersionUID value.

private static final long serialVersionUID = 8683452581122892189L;

3) Code testing

* First create a Baby class to implement the Serializable interface

package mytest;

import java.io.Serializable;

public class Baby implements Serializable{

private int age;

private String name;

public Baby(int age,String name) {

this.age=age;

this.name=name;

}

@Override

public String toString() {

return "Baby [age=" + age + ", name=" + name + "]";

}

}

* Then create a Mother class to implement the Serializable interface. Contains an instance variable age of type int, an instance variable mybaby of type Baby, a static variable name of type String, and a String instance variable modifier of transient s for serialization testing

package mytest;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

public class Mother implements Serializable {

private int i;

private Baby mybaby;// An instance variable is also an implementation Serializable Objects of classes of

private static String name;

private transient String nickname;

public Mother(int i, Baby mybaby, String nickname) {

this.i = i;

this.mybaby = mybaby;

this.nickname = nickname;

}

public void setName(String name) {

this.name = name;

}

public String toString() {

return "Mother [i=" + i + ", mybaby=" + mybaby +", name="+name+", nickname=" + nickname + "]";

}

public static void main(String[] args) {

Baby mybaby = new Baby(2, "Petty officer");

Mother mother = new Mother(25, mybaby, "Guan Ma");

mother.setName("Xiaohong");//At this point, the static variable name The reference string is"Xiaohong"

ObjectOutputStream oos = null;

try {

FileOutputStream fos = new FileOutputStream("MyMom.ser");

//"MyMom.ser"If it does not exist, it will be created automatically

oos = new ObjectOutputStream(fos);

oos.writeObject(mother);//Serialize and write objects referenced by variables MyMom.ser This document

} catch (Exception e) {

e.printStackTrace();

} finally {

if (oos != null) {

try {

oos.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

ObjectInputStream ois = null;

Mother reMother=null;

mother.setName("Not red");//Static variables name The reference string is modified to"Not red"

try {

FileInputStream fis = new FileInputStream("MyMom.ser");

ois = new ObjectInputStream(fis);

reMother = (Mother) ois.readObject();

//Will write MyMom.ser Object deserialization,Because the return is Object Objects, so you need to force them

} catch (Exception e) {

e.printStackTrace();

} finally {

if (ois != null) {

try {

ois.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

System.out.println(reMother);//Objects that are deserialized reMother Print out

}

}

* The output is as follows

Mother [i=25, mybaby=Baby [age=2, name = Xiaoguan], name = not Xiaohong, nickname=null]

It can be found that in the mobile object, the object referenced by the instance variable mybaby of Baby type is serialized, the static variable name of String type is not serialized, and the instance variable nickname modified by transient s is also initialized to null.

2. java.lang.Cloneable Markup Interface

This interface marks whether the object of a class has a secure clone method. In the java.lang.Object class, there is a clone() method as follows

protected native Object clone() throws CloneNotSupportedException;//This is not the case with the implementation of the native modifier representation method, which is implemented in C/C++ in different places.

Since the Object class is also the parent of all classes, all classes have a clone() method that inherits from Object. If a Class objects calls the Of clone method that inherits from Object but does not have an implements Cloneable interface, the virtual machine will throw CloneNotSupportedException.

1) The modifier of the Object class clone() is the effect of protected. Because the clone() method of Object is protected, the clone method inherited from Object by a class is only visible by default in the java.lang package, and the subclass can inherit, but the subclass can only call the protected clone() method to clone its own object (note that calling super.clone() will not clone the object of the parent class, or clone the object of this class). A class must redefine clone() as a public method to allow "all methods" to call the clone() method of the class to clone an instance of the class.

2) What is the relationship between the cloned object and the original object? Cloned objects have the same attributes as the original objects, but not the same objects. Their memory addresses are different, so changing the attributes of cloned objects will not affect the original objects. But the clone method inherited from Object is a shallow copy. If the original object contains a reference to a child object, the copy object and the original object will have a reference to the same child object. In this case, the change of the clone object to the child object will change the attributes of the child object in the original object. So if not all the subobjects referenced by the original object are immutable objects like String, the clone method must be redefined to create a deep copy.

3) Code validation

* First, create a People class to implement the Cloneable interface, with only people\_id and people\_name attributes

package mytest2;

public class People implements Cloneable {

private int people\_age;

private String people\_name;

public People(int people\_age, String people\_name) {

super();

this.people\_age = people\_age;

this. people\_name = people\_name;

}

@Override

public String toString() {

return " [people\_age=" +people\_age + ", people\_name=" + people\_name + "]";

}

}

* In creating an Adult class inherited from People, the Cloneable interface is implemented, with adult\_age and adult\_name attributes, and a Date type testDay attribute. Testing in main method

package mytest2;

import java.util.Date;

public class Adult extends People implements Cloneable {

int adult\_age;

String adult\_name;

Date testDay;

public Adult(int adult\_age, String adult\_name, int people\_age, String people\_name) {

super(people\_age, people\_name);

this.adult\_age = adult\_age;

this.adult\_name = adult\_name;

this.testDay= new Date();

}

public People superClone() throws CloneNotSupportedException {

return (People) super.clone();// Here the clone will still be one Adult Class object

}

@Override

public String toString() {

return "[adult\_age=" + adult\_age + ", adult\_name=" + adult\_name + ", testDay=" + testDay + super.toString()+"]";

}

public static void main(String[] args) {

People people = new People(26, "Petty officer");

// people.clone();//Parent class clone Method is not visible here, error reporting

Adult adult = new Adult(18, "Adult officials",10,"Petty officer");

Adult adultclone = null;

try {

adultclone = (Adult) adult.clone();// Object Of clone Method return value Object Object, which needs to be forced here

people = (People) adult.superClone();// What you get here should still be one. Adult object

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

System.out.println(adultclone);

System.out.println(people);

adultclone.testDay.setTime(System.currentTimeMillis()+1000000000);

/\* System.currentTimeMillis()We get milliseconds from 0:00 on January 1, 1970 to the current time.

Here, change the value of testDay to test whether the testday attribute of the original object will change, and what changes have taken place?\*/

System.out.println("--------------------");

System.out.println(adultclone);

System.out.println(adult);

}

}

* The output is

     [adult\_age=18, adult\_name=Adult officials, testDay=Wed Apr 17 13:05:17 CST 2019 [people\_age=10, people\_name=Petty officer]]  
     [adult\_age=18, adult\_name=Adult officials, testDay=Wed Apr 17 13:05:17 CST 2019 [people\_age=10, people\_name=Petty officer]]  
     --------------------  
     [adult\_age=18, adult\_name=Adult officials, testDay=Mon Apr 29 02:51:57 CST 2019 [people\_age=10, people\_name=Petty officer]]  
     [adult\_age=18, adult\_name=Adult officials, testDay=Mon Apr 29 02:51:57 CST 2019 [people\_age=10, people\_name=Petty officer]]

The output of the first line is the same as that of the second line. It can be seen that super.clone() is not a cloned parent class, but an object of the cloned Adult class. The output of the third and fourth lines has changed compared with the output of the first two lines. It can verify that the child object referenced by the cloned object has changed, and the child object referenced by the original object has changed the same.

4) Establishment of deep cloning

* Because the Date type has overridden the clone method and changed the clone() modifier to public, the clone method can be called directly from an instance of Date to clone an instance of Date. Date related source code (JDK8) is as follows:

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;

}

* Override the clone method in the Adult class and implement deep cloning. The code is as follows

public Object clone() {

Adult adultClone = null;

try {

adultClone = (Adult) super.clone();

adultClone.testDay = (Date) this.testDay.clone();//To clone the object's testDay Domain becomes original object test A Copy of Domain to Realize Deep Cloning

} catch (CloneNotSupportedException e) {}

return adultClone;

}

* After covering clone(), the test results are as follows

[adult\_age=18, adult\_name=Adult officials, testDay=Wed Apr 17 13:28:43 CST 2019 [people\_age=10, people\_name=Petty officer]]

[adult\_age=18, adult\_name=Adult officials, testDay=Wed Apr 17 13:28:43 CST 2019 [people\_age=10, people\_name=Petty officer]]

--------------------

[adult\_age=18, adult\_name=Adult officials, testDay=Mon Apr 29 03:15:23 CST 2019 [people\_age=10, people\_name=Petty officer]]

[adult\_age=18, adult\_name=Adult officials, testDay=Wed Apr 17 13:28:43 CST 2019 [people\_age=10, people\_name=Petty officer]]//Subobjects referenced by the original object testDay This time, nothing has changed!

3. java.util.Random Access Markup Interface

Random Access tag interface can be implemented if a container class object has the characteristics of fast random access to elements within the object. For example, the ArrayList class implements this interface, because the ArrayList is a dynamic array, and the bottom layer is implemented by the array, so the ArrayList object can quickly and randomly access the elements contained in the object according to the index, with the time complexity of O(1).

4. java.rmi.Remote Markup Interface

This tag interface is used for remote method invocation, namely RMI (Remote Method Invocation) technology. If a class wants to implement remote method calls, it needs to implement the interface first. RMI technology is also the principle of EJB technology.