对象序列化的目标：将对象保存到磁盘中，或允许在网络中直接传输对象。

对象序列化机制允许把内存中的Java对象转换成平台无关的二进制流，从而允许把这种二进制流持久的保存在磁盘上，通过网络将这种二进制流传输到另一个网络节点。其他程序一旦获得了这种二进制流，都可以讲这种二进制流恢复成原来的Java对象。

如果需要让某个对象支持序列化机制，则必须让它的类是可序列化的，则这个类必须实现如下两个接口之一：

· Serializable

· Externalizable

Serializable是一个标志接口，它只是表明该类的实例是可序列化的。

****一、 使用对象流实现序列化****

一旦某个类实现了Serializable接口，则该类的对象就是可序列化的，程序通过如下步骤创建可序列化对象：

1） 创建一个ObjectOutStream，这个输出流是一个处理流：

1. ObjectOutputStream oos = **new** ObjectOutputStream("object.txt");

2） 调用ObjectOutputStream对象的writeObject()方法输出可序列化对象：

1. **public** **class** Person **implements** java.io.Serializable
2. {
3. **public** String name;
5. **public** **int** age;
7. // 构造方法
9. // setter和getter方法
10. }

使用ObjectOutputStream将一个Person对象写入磁盘文件：

1. **public** **class** WriteObject
2. {
3. **public** **static** **void** main(String[] args)  **throws** Exception
4. {
5. ObjectOutputStream oos = **new** ObjectOutputStream("object.txt");
6. Person per = **new** Person("沉缘",25);
7. oos.writeObject(per);
8. }
9. }

通过ObjectOutputStream，我们将Person对象保存到了文件中（硬盘），我们可以看到在当前目录中已经有了object.txt文件。

如果希望从二进制流中恢复对象，则可以通过反序列化机制，步骤如下：

1） 创建一个ObjectInputStream输入流，这个输入流是一个处理流，所以必须建立在其他节点流的基础上。

1. FileInputStream fis = **new** FileInputStream("object.txt");
2. ObjectInputStream ois = **new** ObjectInputStream(fis);

2） 调用ObjectInputStream对象的readObject()方法读取流中的对象，该方法返回一个Object类型的Java对象，可对该对象进行强制转换：

1. Person per= (Person) ois.readObject();
2. ois.close();

实例：

1. **public** **class** ReadObject
2. {
3. **public** **static** **void** main(String[] args)
4. {
5. **try**(
6. // 创建一个ObjectInputStream输入流
7. ObjectInputStream ois = **new** ObjectInputStream(
8. **new** FileInputStream("object.txt")))
9. {
10. // 从输入流中读取一个Java对象，并将其强制类型转换为Person类
11. Person p = (Person)ois.readObject();
12. System.out.println("名字为：" + p.getName()
13. + "\n年龄为：" + p.getAge());
14. }
15. **catch** (Exception ex)
16. {
17. ex.printStackTrace();
18. }
19. }
20. }

反序列化读取的仅仅是Java对象的数据，而不是Java类，因此采用反序列化恢复Java对象时，必须提供该对象所属类的class文件，否则将会引发ClassNotFoundException异常。

反序列化机制无须通过构造器来初始化Java对象。

****二、 对象引用的序列化****

在对象的嵌套过程中，比如Teacher类中有Person对象，如果希望Teacher对象是可序列化的，则Person对象也必须是可序列化的。

1. **class** Teacher **implements** java.io.Serializable
2. {
3. **private** String name;
5. **private** Person student;
7. //构造方法
9. //setter、getter方法
10. }

序列化机制的算法：

· 所有保存到磁盘中的对象都有一个序列化编号。

· 当程序试图序列化一个对象时，程序将先检查该对象是否已经被序列化过，只有该对象从未被序列化过，系统才会将该对象转换成字节序列输出。

· 如果某个对象已经被序列化过，程序将只是直接输出一个序列化编号，而不是再次重新序列化该对象。

下面程序序列化两个Teacher对象，两个Teacher对象都持有一个引用到同一个Person对象的引用，而且程序两次调用writeObject()方法输出同一个Teacher对象。

1. **public** **class** WriteTeacher
2. {
3. **public** **static** **void** main(String[] args) **throws** Exception{
4. ObjectOutputStream oos = **new** ObjectOutputStream("object.txt");
5. Person per = **new** Person("沉缘",25);
6. Teacher t1 = **new** Teacher("无情",<span style="font-family: SimSun;">per</span>);
7. Teacher t2 = **new** Teacher("无缘",per);
9. oos.writeObject(t1);
10. oos.writeObject(t2);
11. oos.writeObject(per);
12. oos.writeObject(t2);
14. oos.close();
15. }
16. }

上述程序，我们看着是序列化了四个对象，实际上只有三个，而且序列中的两个Teacher对象的student引用实际上时用一个Person对象。

接下来，我们读取引用：

1. **public** **class** ReadTeacher
2. {
3. **public** **static** **void** main(String[] args)
4. {
5. **try**(
6. // 创建一个ObjectInputStream输出流
7. ObjectInputStream ois = **new** ObjectInputStream(
8. **new** FileInputStream("teacher.txt")))
9. {
10. // 依次读取ObjectInputStream输入流中的四个对象
11. Teacher t1 = (Teacher)ois.readObject();
12. Teacher t2 = (Teacher)ois.readObject();
13. Person p = (Person)ois.readObject();
14. Teacher t3 = (Teacher)ois.readObject();
15. // 输出true
16. System.out.println("t1的student引用和p是否相同："
17. + (t1.getStudent() == p));
18. // 输出true
19. System.out.println("t2的student引用和p是否相同："
20. + (t2.getStudent() == p));
21. // 输出true
22. System.out.println("t2和t3是否是同一个对象："
23. + (t2 == t3));
24. }
25. **catch** (Exception ex)
26. {
27. ex.printStackTrace();
28. }
29. }
30. }

此时，我们应该注意到一个问题，那就是，序列化一个可变对象时，只有第一次使用writeObject（）方法输出时才会将该对象转换成字节序列并输出，当程序再次调用writeObject（）方法时，程序只是输出前面的序列化编号，及时后面对象的Field值已改变，改变的Field值也不会被输出。

1. **public** **class** SerializeMutable
2. {
3. **public** **static** **void** main(String[] args)
4. {
6. **try**(
7. // 创建一个ObjectOutputStream输入流
8. ObjectOutputStream oos = **new** ObjectOutputStream(
9. **new** FileOutputStream("mutable.txt"));
10. // 创建一个ObjectInputStream输入流
11. ObjectInputStream ois = **new** ObjectInputStream(
12. **new** FileInputStream("mutable.txt")))
13. {
14. Person per = **new** Person("孙悟空", 500);
15. // 系统会per对象转换字节序列并输出
16. oos.writeObject(per);
17. // 改变per对象的name Field
18. per.setName("猪八戒");
19. // 系统只是输出序列化编号，所以改变后的name不会被序列化
20. oos.writeObject(per);
21. Person p1 = (Person)ois.readObject();    //①
22. Person p2 = (Person)ois.readObject();    //②
23. // 下面输出true，即反序列化后p1等于p2
24. System.out.println(p1 == p2);
25. // 下面依然看到输出"孙悟空"，即改变后的Field没有被序列化
26. System.out.println(p2.getName());
27. }
28. **catch** (Exception ex)
29. {
30. ex.printStackTrace();
31. }
32. }
33. }

三、 自定义序列化

通过在Field（属性）前使用transient关键字，可以指定Java序列化时无须理会该Field。

1. **public** **class** Person
2. **implements** java.io.Serializable
3. {
4. **private** String name;
5. **private** **transient** **int** age;
6. // 注意此处没有提供无参数的构造器!
7. **public** Person(String name , **int** age)
8. {
9. System.out.println("有参数的构造器");
10. **this**.name = name;
11. **this**.age = age;
12. }
13. // 省略name与age的setter和getter方法
15. }

测试该Person对象：

1. public class TransientTest
2. {
3. public static void main(String[] args)
4. {
5. try(
6. // 创建一个ObjectOutputStream输出流
7. ObjectOutputStream oos = new ObjectOutputStream(
8. new FileOutputStream("transient.txt"));
9. // 创建一个ObjectInputStream输入流
10. ObjectInputStream ois = new ObjectInputStream(
11. new FileInputStream("transient.txt")))
12. {
13. Person per = new Person("孙悟空", 500);
14. // 系统会per对象转换字节序列并输出
15. oos.writeObject(per);
16. Person p = (Person)ois.readObject();
17. System.out.println(p.getName());
18. System.out.println(p.getAge());
19. }
20. catch (Exception ex)
21. {
22. ex.printStackTrace();
23. }
24. }
25. }

输出：

有参数的构造器

孙悟空0

观察输出，获取的age值为0， 说明在序列化时，该age属性并未被序列化。

四、 Externalizable接口

该接口提供的序列化机制，完全由程序员决定存储和恢复对象数据。Externalizable接口中两个需实现的方法。

|  |  |
| --- | --- |
| void | readExternal(**ObjectInput** in)  The object implements the readExternal method to restore its contents by calling the methods of DataInput for primitive types and readObject for objects, strings and arrays. |
| void | writeExternal(**ObjectOutput** out)  The object implements the writeExternal method to save its contents by calling the methods of DataOutput for its primitive values or calling the writeObject method of ObjectOutput for objects, strings, and arrays. |

我们举个例子，看下如何使用该接口来序列化对象。

1. **public** **class** Person
2. **implements** java.io.Externalizable
3. {
4. **private** String name;
5. **private** **int** age;
6. // 注意此处没有提供无参数的构造器!
7. **public** Person(String name , **int** age)
8. {
9. System.out.println("有参数的构造器");
10. **this**.name = name;
11. **this**.age = age;
12. }
13. // 省略name与age的setter和getter方法
15. // name的setter和getter方法
16. **public** **void** setName(String name)
17. {
18. **this**.name = name;
19. }
20. **public** String getName()
21. {
22. **return** **this**.name;
23. }
25. // age的setter和getter方法
26. **public** **void** setAge(**int** age)
27. {
28. **this**.age = age;
29. }
30. **public** **int** getAge()
31. {
32. **return** **this**.age;
33. }
35. **public** **void** writeExternal(java.io.ObjectOutput out)
36. **throws** IOException
37. {
38. // 将name Field的值反转后写入二进制流
39. out.writeObject(**new** StringBuffer(name).reverse());
40. out.writeInt(age);
41. }
42. **public** **void** readExternal(java.io.ObjectInput in)
43. **throws** IOException, ClassNotFoundException
44. {
45. // 将读取的字符串反转后赋给name Field
46. **this**.name = ((StringBuffer)in.readObject()).reverse().toString();
47. **this**.age = in.readInt();
48. }
49. }

两种序列化机制对比：



对象序列化需要注意：

1.  对象的类名、Field都会被序列化； 方法、static Field、transient Field都不会被序列化。

2. 实现Serializable接口的类如果需要让某个Field不被序列化，则可以在该Field前添加transient私事符。

3. 保证序列化对象的Field类型也是可序列化的。

4. 反序列化对象时必须有序列化对象的class文件。

5. 当通过文件、网络来读取序列化后的对象时，必须按实际写入的顺序读取。

五、 版本

在对象进行序列化或者反序列化操作的时候，要考虑JDK版本问题。如果序列化的JDK版本和反序列化的版本不一致，则可能出现异常。

因此，可以在序列化操作中引入一个serialVersionUID的长了，通过此常量验证版本的一致性。

1. **import** java.io.Serializable ;
2. **public** **class** Person **implements** Serializable{
3. **private** **static** **final** **long** serialVersionUID = 1L;
4. **private** String name ;   // 声明name属性，但是此属性不被序列化
5. **private** **int** age ;       // 声明age属性
6. **public** Person(String name,**int** age){ // 通过构造设置内容
7. **this**.name = name ;
8. **this**.age = age ;
9. }
10. **public** String toString(){   // 覆写toString()方法
11. **return** "姓名：" + **this**.name + "；年龄：" + **this**.age ;
12. }
13. };