**01、先来点理论**

Java 序列化是 JDK 1.1 时引入的一组开创性的特性，用于将 Java 对象转换为字节数组，便于存储或传输。此后，仍然可以将字节数组转换回 Java 对象原有的状态。

序列化的思想是“冻结”对象状态，然后写到磁盘或者在网络中传输；反序列化的思想是“解冻”对象状态，重新获得可用的 Java 对象。

再来看看序列化 Serializbale 接口的定义：

public interface Serializable {

}

明明就一个空的接口嘛，竟然能够保证实现了它的“类的对象”被序列化和反序列化？

**02、再来点实战**

在回答上述问题之前，我们先来创建一个类（只有两个字段，和对应的 getter/setter），用于序列化和反序列化。

class Wanger {

private String name;

private int 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;

}

}

再来创建一个测试类，通过 ObjectOutputStream 将“18 岁的王二”写入到文件当中，实际上就是一种序列化的过程；再通过 ObjectInputStream 将“18 岁的王二”从文件中读出来，实际上就是一种反序列化的过程。

public class Test {

public static void main(String[] args) {

// 初始化

Wanger wanger = new Wanger();

wanger.setName("王二");

wanger.setAge(18);

System.out.println(wanger);

// 把对象写到文件中

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("chenmo"));){

oos.writeObject(wanger);

} catch (IOException e) {

e.printStackTrace();

}

// 从文件中读出对象

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(new File("chenmo")));){

Wanger wanger1 = (Wanger) ois.readObject();

System.out.println(wanger1);

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

不过，由于 Wanger 没有实现 Serializbale 接口，所以在运行测试类的时候会抛出异常，堆栈信息如下：

java.io.NotSerializableException: com.cmower.java\_demo.xuliehua.Wanger

at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184)

at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348)

at com.cmower.java\_demo.xuliehua.Test.main(Test.java:21)

顺着堆栈信息，我们来看一下 ObjectOutputStream 的 writeObject0() 方法。其部分源码如下：

if (obj instanceof String) {

writeString((String) obj, unshared);

} else if (cl.isArray()) {

writeArray(obj, desc, unshared);

} else if (obj instanceof Enum) {

writeEnum((Enum<?>) obj, desc, unshared);

} else if (obj instanceof Serializable) {

writeOrdinaryObject(obj, desc, unshared);

} else {

if (extendedDebugInfo) {

throw new NotSerializableException(

cl.getName() + "\n" + debugInfoStack.toString());

} else {

throw new NotSerializableException(cl.getName());

}

}

也就是说，ObjectOutputStream 在序列化的时候，会判断被序列化的对象是哪一种类型，字符串？数组？枚举？还是 Serializable，如果全都不是的话，抛出 NotSerializableException。

假如 Wanger 实现了 Serializable 接口，就可以序列化和反序列化了。

class Wanger implements Serializable{

private static final long serialVersionUID = -2095916884810199532L;

private String name;

private int age;

}

具体怎么序列化呢？

以 ObjectOutputStream 为例吧，它在序列化的时候会依次调用 writeObject()→writeObject0()→writeOrdinaryObject()→writeSerialData()→invokeWriteObject()→defaultWriteFields()。

private void defaultWriteFields(Object obj, ObjectStreamClass desc)

throws IOException

{

Class<?> cl = desc.forClass();

desc.checkDefaultSerialize();

int primDataSize = desc.getPrimDataSize();

desc.getPrimFieldValues(obj, primVals);

bout.write(primVals, 0, primDataSize, false);

ObjectStreamField[] fields = desc.getFields(false);

Object[] objVals = new Object[desc.getNumObjFields()];

int numPrimFields = fields.length - objVals.length;

desc.getObjFieldValues(obj, objVals);

for (int i = 0; i < objVals.length; i++) {

try {

writeObject0(objVals[i],

fields[numPrimFields + i].isUnshared());

}

}

}

那怎么反序列化呢？

以 ObjectInputStream 为例，它在反序列化的时候会依次调用 readObject()→readObject0()→readOrdinaryObject()→readSerialData()→defaultReadFields()。

private void defaultWriteFields(Object obj, ObjectStreamClass desc)

throws IOException

{

Class<?> cl = desc.forClass();

desc.checkDefaultSerialize();

int primDataSize = desc.getPrimDataSize();

desc.getPrimFieldValues(obj, primVals);

bout.write(primVals, 0, primDataSize, false);

ObjectStreamField[] fields = desc.getFields(false);

Object[] objVals = new Object[desc.getNumObjFields()];

int numPrimFields = fields.length - objVals.length;

desc.getObjFieldValues(obj, objVals);

for (int i = 0; i < objVals.length; i++) {

try {

writeObject0(objVals[i],

fields[numPrimFields + i].isUnshared());

}

}

}

我想看到这，你应该会恍然大悟的“哦”一声了。Serializable 接口之所以定义为空，是因为它只起到了一个标识的作用，告诉程序实现了它的对象是可以被序列化的，但真正序列化和反序列化的操作并不需要它来完成。

**03、再来点注意事项**

开门见山的说吧，static 和 transient 修饰的字段是不会被序列化的。

为什么呢？我们先来证明，再来解释原因。

首先，在 Wanger 类中增加两个字段。

class Wanger implements Serializable {

private static final long serialVersionUID = -2095916884810199532L;

private String name;

private int age;

public static String pre = "沉默";

transient String meizi = "王三";

@Override

public String toString() {

return "Wanger{" + "name=" + name + ",age=" + age + ",pre=" + pre + ",meizi=" + meizi + "}";

}

}

其次，在测试类中打印序列化前和反序列化后的对象，并在序列化后和反序列化前改变 static字段的值。具体代码如下：

// 初始化

Wanger wanger = new Wanger();

wanger.setName("王二");

wanger.setAge(18);

System.out.println(wanger);

// 把对象写到文件中

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("chenmo"));){

oos.writeObject(wanger);

} catch (IOException e) {

e.printStackTrace();

}

// 改变 static 字段的值

Wanger.pre ="不沉默";

// 从文件中读出对象

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(new File("chenmo")));){

Wanger wanger1 = (Wanger) ois.readObject();

System.out.println(wanger1);

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

// Wanger{name=王二,age=18,pre=沉默,meizi=王三}

// Wanger{name=王二,age=18,pre=不沉默,meizi=null}

从结果的对比当中，我们可以发现：

1）序列化前，pre 的值为“沉默”，序列化后，pre 的值修改为“不沉默”，反序列化后，pre 的值为“不沉默”，而不是序列化前的状态“沉默”。

为什么呢？因为序列化保存的是对象的状态，而 static 修饰的字段属于类的状态，因此可以证明序列化并不保存 static 修饰的字段。

2）序列化前，meizi 的值为“王三”，反序列化后，meizi 的值为 null，而不是序列化前的状态“王三”。

为什么呢？transient 的中文字义为“临时的”（论英语的重要性），它可以阻止字段被序列化到文件中，在被反序列化后，transient 字段的值被设为初始值，比如 int 型的初始值为 0，对象型的初始值为 null。

如果想要深究源码的话，你可以在 ObjectStreamClass 中发现下面这样的代码：

private static ObjectStreamField[] getDefaultSerialFields(Class<?> cl) {

Field[] clFields = cl.getDeclaredFields();

ArrayList<ObjectStreamField> list = new ArrayList<>();

int mask = Modifier.STATIC | Modifier.TRANSIENT;

int size = list.size();

return (size == 0) ? NO\_FIELDS :

list.toArray(new ObjectStreamField[size]);

}

看到 Modifier.STATIC | Modifier.TRANSIENT，是不是感觉更好了呢？

**04、再来点干货**

除了 Serializable 之外，Java 还提供了一个序列化接口 Externalizable（念起来有点拗口）。

两个接口有什么不一样的吗？试一试就知道了。

首先，把 Wanger 类实现的接口 Serializable 替换为 Externalizable。

class Wanger implements Externalizable {

private String name;

private int age;

public Wanger() {

}

public String getName() {

return name;

}

@Override

public String toString() {

return "Wanger{" + "name=" + name + ",age=" + age + "}";

}

@Override

public void writeExternal(ObjectOutput out) throws IOException {

}

@Override

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {

}

}

实现 Externalizable 接口的 Wanger 类和实现 Serializable 接口的 Wanger 类有一些不同：

1）新增了一个无参的构造方法。

使用 Externalizable 进行反序列化的时候，会调用被序列化类的无参构造方法去创建一个新的对象，然后再将被保存对象的字段值复制过去。否则的话，会抛出以下异常：

java.io.InvalidClassException: com.cmower.java\_demo.xuliehua1.Wanger; no valid constructor

at java.io.ObjectStreamClass$ExceptionInfo.newInvalidClassException(ObjectStreamClass.java:150)

at java.io.ObjectStreamClass.checkDeserialize(ObjectStreamClass.java:790)

at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1782)

at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1353)

at java.io.ObjectInputStream.readObject(ObjectInputStream.java:373)

at com.cmower.java\_demo.xuliehua1.Test.main(Test.java:27)

2）新增了两个方法 writeExternal() 和 readExternal()，实现 Externalizable 接口所必须的。

然后，我们再在测试类中打印序列化前和反序列化后的对象。

// 初始化

Wanger wanger = new Wanger();

wanger.setName("王二");

wanger.setAge(18);

System.out.println(wanger);

// 把对象写到文件中

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("chenmo"));) {

oos.writeObject(wanger);

} catch (IOException e) {

e.printStackTrace();

}

// 从文件中读出对象

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(new File("chenmo")));) {

Wanger wanger1 = (Wanger) ois.readObject();

System.out.println(wanger1);

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

// Wanger{name=王二,age=18}

// Wanger{name=null,age=0}

从输出的结果看，反序列化后得到的对象字段都变成了默认值，也就是说，序列化之前的对象状态没有被“冻结”下来。

为什么呢？因为我们没有为 Wanger 类重写具体的 writeExternal() 和 readExternal() 方法。那该怎么重写呢？

@Override

public void writeExternal(ObjectOutput out) throws IOException {

out.writeObject(name);

out.writeInt(age);

}

@Override

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {

name = (String) in.readObject();

age = in.readInt();

}

1）调用 ObjectOutput 的 writeObject() 方法将字符串类型的 name 写入到输出流中；

2）调用 ObjectOutput 的 writeInt() 方法将整型的 age 写入到输出流中；

3）调用 ObjectInput 的 readObject() 方法将字符串类型的 name 读入到输入流中；

4）调用 ObjectInput 的 readInt() 方法将字符串类型的 age 读入到输入流中；

再运行一次测试了类，你会发现对象可以正常地序列化和反序列化了。

序列化前：Wanger{name=王二,age=18}  
序列化后：Wanger{name=王二,age=18}

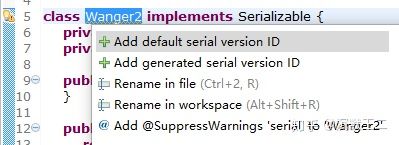
**05、再来点甜点**

让我先问问你吧，你知道 private static final long serialVersionUID = -2095916884810199532L; 这段代码的作用吗？

嗯……

serialVersionUID 被称为序列化 ID，它是决定 Java 对象能否反序列化成功的重要因子。在反序列化时，Java 虚拟机会把字节流中的 serialVersionUID 与被序列化类中的 serialVersionUID进行比较，如果相同则可以进行反序列化，否则就会抛出序列化版本不一致的异常。

当一个类实现了 Serializable 接口后，IDE 就会提醒该类最好产生一个序列化 ID，就像下面这样：



1）添加一个默认版本的序列化 ID：

private static final long serialVersionUID = 1L。

2）添加一个随机生成的不重复的序列化 ID。

private static final long serialVersionUID = -2095916884810199532L;

3）添加 @SuppressWarnings 注解。

@SuppressWarnings("serial")

怎么选择呢？

首先，我们采用第二种办法，在被序列化类中添加一个随机生成的序列化 ID。

class Wanger implements Serializable {

private static final long serialVersionUID = -2095916884810199532L;

private String name;

private int age;

// 其他代码忽略

}

然后，序列化一个 Wanger 对象到文件中。

// 初始化

Wanger wanger = new Wanger();

wanger.setName("王二");

wanger.setAge(18);

System.out.println(wanger);

// 把对象写到文件中

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("chenmo"));) {

oos.writeObject(wanger);

} catch (IOException e) {

e.printStackTrace();

}

这时候，我们悄悄地把 Wanger 类的序列化 ID 偷梁换柱一下，嘿嘿。

// private static final long serialVersionUID = -2095916884810199532L;

private static final long serialVersionUID = -2095916884810199533L;

好了，准备反序列化吧。

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(new File("chenmo")));) {

Wanger wanger = (Wanger) ois.readObject();

System.out.println(wanger);

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

哎呀，出错了。

java.io.InvalidClassException: local class incompatible: stream classdesc

serialVersionUID = -2095916884810199532,

local class serialVersionUID = -2095916884810199533

at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1521)

at com.cmower.java\_demo.xuliehua1.Test.main(Test.java:27)

异常堆栈信息里面告诉我们，从持久化文件里面读取到的序列化 ID 和本地的序列化 ID 不一致，无法反序列化。

那假如我们采用第三种方法，为 Wanger 类添加个 @SuppressWarnings("serial") 注解呢？

@SuppressWarnings("serial")

class Wanger3 implements Serializable {

// 省略其他代码

}

好了，再来一次反序列化吧。可惜依然报错。

java.io.InvalidClassException: local class incompatible: stream classdesc

serialVersionUID = -2095916884810199532,

local class serialVersionUID = -3818877437117647968

at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1521)

at com.cmower.java\_demo.xuliehua1.Test.main(Test.java:27)

异常堆栈信息里面告诉我们，本地的序列化 ID 为 -3818877437117647968，和持久化文件里面读取到的序列化 ID 仍然不一致，无法反序列化。这说明什么呢？使用 @SuppressWarnings("serial") 注解时，该注解会为被序列化类自动生成一个随机的序列化 ID。

由此可以证明，Java 虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，还有一个非常重要的因素就是序列化 ID 是否一致。

也就是说，如果没有特殊需求，采用默认的序列化 ID（1L）就可以，这样可以确保代码一致时反序列化成功。

class Wanger implements Serializable {

private static final long serialVersionUID = 1L;

// 省略其他代码

}