Serializable接口

# java.io. Serializable接口

什么是Serializable接口？

一个对象序列化的接口，一个类只有实现了Serializable接口，它的对象才能被序列化

什么是序列化？

将对象的状态信息转换为可以存储或传输的形式的过程，在序列化期间，对象将其当前状态写入到临时存储区或持久性存储区，之后，便可以通过从存储区中读取或反序列化对象的状态信息，来重新创建该对象

什么情况下需要序列化？

当我们需要把对象的状态信息通过网络进行传输，或者需要将对象的状态信息持久化，以便将来使用时都需要把对象进行序列化。

Serializable主要用来支持两种主要的特性：

1、Java的RMI（**remote method invocation**），RMI允许像在本机上一样操作远程机器上的对象，当发送消息给远程对象时，就需要用到**序列化机制来发送参数和接受返回值**；

2、Java的JavaBean，Bean的状态信息通常是在设计时配置的，Bean的状态信息必须被保存下来，以便当程序运行时能恢复这些状态信息，这也需要Serializable序列化机制。

**将 Java 对象序列化为二进制文件的 Java 序列化技术是 Java 系列技术中一个较为重要的技术点，在大部分情况下，开发人员只需要了解被序列化的类需要实现 Serializable 接口，使用 ObjectInputStream 和 ObjectOutputStream 进行对象的读写。**然而在有些情况下，光知道这些还远远不够，文章列举了笔者遇到的一些真实情境，它们与 Java 序列化相关，通过分析情境出现的原因，使读者轻松牢记 Java 序列化中的一些高级认识。

# 本文将逐一的介绍几个情境，顺序如下面的列表。

•序列化 ID 的问题：ID必须一致；

•静态变量序列化：静态变量属于类的状态，不会被序列化；

•父类的序列化与 Transient 关键字：

•对敏感字段加密

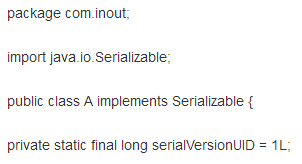
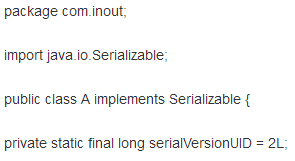
•序列化存储规则

## •序列化 ID 的问题

情境：两个客户端 A 和 B 试图通过网络传递对象数据，A 端将对象 C 序列化为二进制数据再传给 B，B 反序列化得到 C。

问题：C 对象的全类路径假设为 com.inout.Test，在 A 和 B 端都有这么一个类文件，功能代码完全一致。也都实现了 Serializable 接口，但是反序列化时总是提示不成功。

**解决：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，一个非常重要的一点是两个类的序列化 ID 是否一致**（就是 private static final long serialVersionUID = 1L）。清单 1 中，虽然两个类的功能代码完全一致，但是序列化 ID 不同，他们无法相互序列化和反序列化。

序列化 ID 在 Eclipse 下提供了两种生成策略，一个是固定的 1L，一个是随机生成一个不重复的 long 类型数据（实际上是使用 JDK 工具生成），在这里有一个建议，如果没有特殊需求，就是用默认的 1L 就可以，这样可以确保代码一致时反序列化成功。那么随机生成的序列化 ID 有什么作用呢，**有些时候，通过改变序列化 ID 可以用来限制某些用户的使用。**

## •静态变量序列化

清单 2. 静态变量序列化问题代码

public class **Test** implements Serializable {

private static final long serialVersionUID = 1L;

**public static int staticVar = 5;**

public static void main(String[] args) {

try {

//初始时staticVar为5

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("result.obj"));

**out.writeObject(new Test());**

out.close();

**//序列化后修改为10**

**Test.staticVar = 10;**

**ObjectInputStream oin = new ObjectInputStream(new FileInputStream("result.obj"));**

Test t = (Test) oin.readObject();

oin.close();

//再读取，通过t.staticVar打印新的值

**System.out.println(t.staticVar);**

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

}

清单 2 中的 main 方法，将对象序列化后，修改静态变量的数值，再将序列化对象读取出来，然后通过读取出来的对象获得静态变量的数值并打印出来。依照清单 2，这个 System.out.println(t.staticVar) 语句输出的是 10 还是 5 呢？

**最后的输出是 10，**对于无法理解的读者认为，打印的 staticVar 是从读取的对象里获得的，应该是保存时的状态才对。之所以打印 10 的原因在于序列化时，并不保存静态变量，这其实比较容易理解，**序列化保存的是对象的状态，静态变量属于类的状态，因此 序列化并不保存静态变量**。

## •父类的序列化与 Transient 关键字

**父类的序列化与 Transient 关键字**

情境：一个子类实现了 Serializable 接口，它的父类都没有实现 Serializable 接口，序列化该子类对象，然后反序列化后输出父类定义的某变量的数值，**该变量数值与序列化时的数值不同**。

解决：**要想将父类对象也序列化，就需要让父类也实现Serializable 接口**。如果父类不实现的话的，就需要有默认的无参的构造函数。在父类没有实现 Serializable 接口时，虚拟机是不会序列化父对象的，而一个 Java 对象的构造必须先有父对象，才有子对象，反序列化也不例外。所以反序列化时，为了构造父对象，只能调用父类的无参构造函数作为默认的父对象。因此当我们取父对象的变量值时，它的值是调用父类无参构造函数后的值。如果你考虑到这种序列化的情况，在父类无参构造函数中对变量进行初始化，否则的话，父类变量值都是默认声明的值，如 int 型的默认是 0，string 型的默认是 null。

**Transient 关键字的作用是控制变量的序列化，在变量声明前加上该关键字，可以阻止该变量被序列化到文件中，在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null。**

特性使用案例

我们熟悉**使用 Transient 关键字可以使得字段不被序列化**，那么还有别的方法吗？根据父类对象序列化的规则，我们可以将不需要被序列化的字段抽取出来放到父类中，子类实现 Serializable 接口，父类不实现，根据父类序列化规则，父类的字段数据将不被序列化。

## •对敏感字段加密

对敏感字段加密

情境：服务器端给客户端发送序列化对象数据，对象中有一些数据是敏感的，比如密码字符串等，希望对该密码字段在序列化时，进行加密，而客户端如果拥有解密的密钥，只有在客户端进行反序列化时，才可以对密码进行读取，这样可以一定程度保证序列化对象的数据安全。

解决：**在序列化过程中，虚拟机会试图调用对象类里的 writeObject 和 readObject 方法，进行用户自定义的序列化和反序列化**，如果没有这样的方法，则默认调用是 ObjectOutputStream 的 defaultWriteObject 方法以及 ObjectInputStream 的 defaultReadObject 方法。用户自定义的 writeObject 和 readObject 方法可以**允许用户控制序列化的过程**，比如可以在序列化的过程中动态改变序列化的数值。基于这个原理，可以在实际应用中得到使用，用于敏感字段的加密工作，清单 3 展示了这个过程。

清单 3. 静态变量序列化问题代码

private static final long serialVersionUID = 1L;

private String password = "pass";

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

private void **writeObject**(ObjectOutputStream out) {

try {

PutField putFields = out.putFields();

System.out.println("原密码:" + password);

password = "encryption";//模拟加密

putFields.put("password", password);

System.out.println("加密后的密码" + password);

out.writeFields();

} catch (IOException e) {

e.printStackTrace();

}

}

private void **readObject**(ObjectInputStream in) {

try {

GetField readFields = in.readFields();

Object object = readFields.get("password", "");

System.out.println("要解密的字符串:" + object.toString());

password = "pass";//模拟解密,需要获得本地的密钥

} catch (IOException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

**ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("result.obj"));**

out.writeObject(new Test());

out.close();

ObjectInputStream oin = new ObjectInputStream(new FileInputStream("result.obj"));

**Test t = (Test) oin.readObject();**

System.out.println("解密后的字符串:" + t.getPassword());

oin.close();

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

在清单 3 的 writeObject 方法中，对密码进行了加密，**在 readObject 中则对 password 进行解密，只有拥有密钥的客户端，才可以正确的解析出密码，确保了数据的安全**。执行清单 3 后控制台输出如图 3 所示。

特性使用案例

**RMI 技术是完全基于 Java 序列化技术的，服务器端接口调用所需要的参数对象来至于客户端，它们通过网络相互传输**。这就涉及 RMI 的安全传输的问题。一些敏感的字段，如用户名密码（用户登录时需要对密码进行传输），我们希望对其进行加密，这时，就可以采用本节介绍的方法在客户端对密码进行加密，服务器端进行解密，确保数据传输的安全性。

## •序列化存储规则

序列化存储规则

情境：问题代码如清单 4 所示。

清单 4. 存储规则问题代码

ObjectOutputStream out = new ObjectOutputStream(

new FileOutputStream("result.obj"));

Test test = new Test();

//试图将对象两次写入文件

out.writeObject(test);

out.flush();

System.out.println(new File("result.obj").length());

out.writeObject(test);

out.close();

System.out.println(new File("result.obj").length());

ObjectInputStream oin = new ObjectInputStream(new FileInputStream("result.obj"));

//从文件依次读出两个文件

Test t1 = (Test) oin.readObject();

Test t2 = (Test) oin.readObject();

oin.close();

//判断两个引用是否指向同一个对象

System.out.println(t1 == t2);

清单 3 中对同一对象两次写入文件，打印出写入一次对象后的存储大小和写入两次后的存储大小，然后从文件中反序列化出两个对象，比较这两个对象是否为同一对象。**一般的思维是，两次写入对象，文件大小会变为两倍的大小，反序列化时，由于从文件读取，生成了两个对象，判断相等时应该是输入 false 才对**，但是最后结果为true。

我们看到，第二次写入对象时文件只增加了 5 字节，并且两个对象是相等的，这是为什么呢？

解答：Java 序列化机制为了节省磁盘空间，具有特定的存储规则，当写入文件的为同一对象时，并不会再将对象的内容进行存储，而只是再次存储一份引用，上面增加的 5 字节的存储空间就是新增引用和一些控制信息的空间。反序列化时，恢复引用关系，使得清单 3 中的 t1 和 t2 指向唯一的对象，二者相等，输出 true。该存储规则极大的节省了存储空间。

特性案例分析

查看清单 5 的代码。

清单 5. 案例代码

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("result.obj"));

Test test = new Test();

test.i = 1;

out.writeObject(test);

out.flush();

test.i = 2;

out.writeObject(test);

out.close();

ObjectInputStream oin = new ObjectInputStream(new FileInputStream("result.obj"));

Test t1 = (Test) oin.readObject();

Test t2 = (Test) oin.readObject();

System.out.println(t1.i);

System.out.println(t2.i);

清单5的目的是希望将 test 对象两次保存到 result.obj 文件中，写入一次以后修改对象属性值再次保存第二次，然后从 result.obj 中再依次读出两个对象，输出这两个对象的 i 属性值。**案例代码的目的原本是希望一次性传输对象修改前后的状态。**

结果两个输出的都是 1， 原因就是第一次写入对象以后，第二次再试图写的时候，虚拟机根据引用关系知道已经有一个相同对象已经写入文件，因此只保存第二次写的引用，所以读取时，都是第一次保存的对象。读者在使用一个文件多次 writeObject 需要特别注意这个问题。