序列化

1. 预备知识
   1. 标记接口

什么是标记接口（Java Mark Interface）

The tag/ marker interface pattern is a design pattern in computer science, used with languages that provide run-time type information about objects. It provides a means to associate metadata with a class where the language does not have explicit support for such metadata.

* + 1. Serializable

下面的代码是我从JDK源代码中摘出来的：

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() + " "

+ debugInfoStack.toString());

} else {

throw new NotSerializableException(cl.getName());

}

}

* + 1. Cloneable

<https://en.wikipedia.org/wiki/Marker_interface_pattern>

* 1. Serializable接口

Serializability of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

类通过实现 java.io.Serializable 接口以启用其序列化功能。未实现此接口的类将无法使其任何状态序列化或反序列化。可序列化类的所有子类型本身都是可序列化的。序列化接口没有方法或字段，仅用于标识可序列化的语义。

To allow subtypes of non-serializable classes to be serialized, the subtype may assume responsibility for saving and restoring the state of the supertype's public, protected, and (if accessible) package fields. The subtype may assume this responsibility only if the class it extends has an accessible no-arg constructor to initialize the class's state. It is an error to declare a class Serializable if this is not the case. The error will be detected at runtime.

要允许不可序列化类的子类型序列化，可以假定该子类型负责保存和恢复超类型的公用 (public)、受保护的 (protected) 和（如果可访问）包 (package) 字段的状态。仅在子类型扩展的类有一个可访问的无参数构造方法来初始化该类的状态时，才可以假定子类型有此职责。如果不是这种情况，则声明一个类为可序列化类是错误的。该错误将在运行时检测到。

During deserialization, the fields of non-serializable classes will be initialized using the public or protected no-arg constructor of the class. A no-arg constructor must be accessible to the subclass that is serializable. The fields of serializable subclasses will be restored from the stream.

在反序列化过程中，将使用该类的公用或受保护的无参数构造方法初始化不可序列化类的字段。可序列化的子类必须能够访问无参数构造方法。可序列化子类的字段将从该流中恢复。

When traversing a graph, an object may be encountered that does not support the Serializable interface. In this case the NotSerializableException will be thrown and will identify the class of the non-serializable object.

当遍历一个图形时，可能会遇到不支持 Serializable 接口的对象。在此情况下，将抛出 NotSerializableException，并将标识不可序列化对象的类。

Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

在序列化和反序列化过程中需要特殊处理的类必须使用下列准确签名来实现特殊方法：

private void writeObject(java.io.ObjectOutputStream out)

throws IOException

private void readObject(java.io.ObjectInputStream in)

throws IOException, ClassNotFoundException;

private void readObjectNoData()

throws ObjectStreamException;

The writeObject method is responsible for writing the state of the object for its particular class so that the corresponding readObject method can restore it. The default mechanism for saving the Object's fields can be invoked by calling out.defaultWriteObject. The method does not need to concern itself with the state belonging to its superclasses or subclasses. State is saved by writing the individual fields to the ObjectOutputStream using the writeObject method or by using the methods for primitive data types supported by DataOutput.

writeObject 方法负责写入特定类的对象的状态，以便相应的 readObject 方法可以恢复它。通过调用 out.defaultWriteObject 可以调用保存 Object 的字段的默认机制。该方法本身不需要涉及属于其超类或子类的状态。通过使用 writeObject 方法或使用 DataOutput 支持的用于基本数据类型的方法将各个字段写入 ObjectOutputStream，状态可以被保存。

The readObject method is responsible for reading from the stream and restoring the classes fields. It may call in.defaultReadObject to invoke the default mechanism for restoring the object's non-static and non-transient fields. The defaultReadObject method uses information in the stream to assign the fields of the object saved in the stream with the correspondingly named fields in the current object. This handles the case when the class has evolved to add new fields. The method does not need to concern itself with the state belonging to its superclasses or subclasses. State is restored by reading data from the ObjectInputStream for the individual fields and making assignments to the appropriate fields of the object. Reading primitive data types is supported by DataInput.

readObject 方法负责从流中读取并恢复类字段。它可以调用 in.defaultReadObject 来调用默认机制，以恢复对象的非静态和非瞬态字段。defaultReadObject 方法使用流中的信息来分配流中通过当前对象中相应指定字段保存的对象的字段。这用于处理类演化后需要添加新字段的情形。该方法本身不需要涉及属于其超类或子类的状态。通过使用 writeObject 方法或使用 DataOutput 支持的用于基本数据类型的方法将各个字段写入 ObjectOutputStream，状态可以被保存。

The readObjectNoData method is responsible for initializing the state of the object for its particular class in the event that the serialization stream does not list the given class as a superclass of the object being deserialized. This may occur in cases where the receiving party uses a different version of the deserialized instance's class than the sending party, and the receiver's version extends classes that are not extended by the sender's version. This may also occur if the serialization stream has been tampered; hence, readObjectNoData is useful for initializing deserialized objects properly despite a "hostile" or incomplete source stream.

在序列化流不列出给定类作为将被反序列化对象的超类的情况下，readObjectNoData 方法负责初始化特定类的对象状态。这在接收方使用的反序列化实例类的版本不同于发送方，并且接收者版本扩展的类不是发送者版本扩展的类时发生。在序列化流已经被篡改时也将发生；因此，不管源流是“敌意的”还是不完整的，readObjectNoData 方法都可以用来正确地初始化反序列化的对象。

Serializable classes that need to designate an alternative object to be used when writing an object to the stream should implement this special method with the exact signature:

将对象写入流时需要指定要使用的替代对象的可序列化类，应使用准确的签名来实现此特殊方法：

ANY-ACCESS-MODIFIER Object writeReplace() throws ObjectStreamException;

This writeReplace method is invoked by serialization if the method exists and it would be accessible from a method defined within the class of the object being serialized. Thus, the method can have private, protected and package-private access. Subclass access to this method follows java accessibility rules.

Classes that need to designate a replacement when an instance of it is read from the stream should implement this special method with the exact signature.

此 writeReplace 方法将由序列化调用，前提是如果此方法存在，而且它可以通过被序列化对象的类中定义的一个方法访问。因此，该方法可以拥有私有 (private)、受保护的 (protected) 和包私有 (package-private) 访问。子类对此方法的访问遵循 java 访问规则。

在从流中读取类的一个实例时需要指定替代的类应使用的准确签名来实现此特殊方法。

ANY-ACCESS-MODIFIER Object readResolve() throws ObjectStreamException;

This readResolve method follows the same invocation rules and accessibility rules as writeReplace.

The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an [InvalidClassException](https://docs.oracle.com/javase/10/docs/api/java/io/InvalidClassException.html" \o "class in java.io). A serializable class can declare its own serialVersionUID explicitly by declaring a field named "serialVersionUID" that must be static, final, and of type long:

此 readResolve 方法遵循与 writeReplace 相同的调用规则和访问规则。

序列化运行时使用一个称为 serialVersionUID 的版本号与每个可序列化类相关联，该序列号在反序列化过程中用于验证序列化对象的发送者和接收者是否为该对象加载了与序列化兼容的类。如果接收者加载的该对象的类的 serialVersionUID 与对应的发送者的类的版本号不同，则反序列化将会导致 InvalidClassException。可序列化类可以通过声明名为 "serialVersionUID" 的字段（该字段必须是静态 (static)、最终 (final) 的 long 型字段）显式声明其自己的 serialVersionUID：

ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;

If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is *strongly recommended* that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization. Therefore, to guarantee a consistent serialVersionUID value across different java compiler implementations, a serializable class must declare an explicit serialVersionUID value. It is also strongly advised that explicit serialVersionUID declarations use the private modifier where possible, since such declarations apply only to the immediately declaring class--serialVersionUID fields are not useful as inherited members. Array classes cannot declare an explicit serialVersionUID, so they always have the default computed value, but the requirement for matching serialVersionUID values is waived for array classes.

如果可序列化类未显式声明 serialVersionUID，则序列化运行时将基于该类的各个方面计算该类的默认 serialVersionUID 值，如“Java(TM) 对象序列化规范”中所述。不过， 强烈建议 所有可序列化类都显式声明 serialVersionUID 值，原因是计算默认的 serialVersionUID 对类的详细信息具有较高的敏感性，根据编译器实现的不同可能千差万别，这样在反序列化过程中可能会导致意外的 InvalidClassException。因此，为保证 serialVersionUID 值跨不同 java 编译器实现的一致性，序列化类必须声明一个明确的 serialVersionUID 值。还强烈建议使用 private 修饰符显示声明 serialVersionUID（如果可能），原因是这种声明仅应用于直接声明类 -- serialVersionUID 字段作为继承成员没有用处。数组类不能声明一个明确的 serialVersionUID，因此它们总是具有默认的计算值，但是数组类没有匹配 serialVersionUID 值的要求。

1. 为什么要使用序列化

1）Java平台允许我们在内存中创建可复用的Java对象，但一般情况下，只有当JVM处于运行时，这些对象才可能存在，即，这些对象的生命周期不会比JVM的生命周期更长。但在现实应用中，就可能要求在JVM停止运行之后能够保存(持久化)指定的对象，并在将来重新读取被保存的对象。Java对象序列化就能够帮助我们实现该功能。

使用Java对象序列化，在保存对象时，会把其状态保存为一组字节，在未来，再将这些字节组装成对象。必须注意地是，对象序列化保存的是对象的”状态”，即它的成员变量。由此可知，**对象序列化不会关注类中的静态变量**。

2）除了在持久化对象时会用到对象序列化之外，当使用RMI(远程方法调用)，或在网络中传递对象时，都会用到对象序列化。Java序列化API为处理对象序列化提供了一个标准机制，该API简单易用。

1. 序列化相关知识

1、在Java中，只要一个类实现了java.io.Serializable接口，那么它就可以被序列化。

2、通过ObjectOutputStream和ObjectInputStream对对象进行序列化及反序列化

3、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）

4、序列化并不保存静态变量。

5、要想将父类对象也序列化，就需要让父类也实现Serializable 接口。

6、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。

7、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

1. ArrayList的序列化

在介绍ArrayList序列化之前，先来考虑一个问题：

***如何自定义的序列化和反序列化策略***

带着这个问题，我们来看java.util.ArrayList的源码

code 3

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class ArrayList<E> extends AbstractList<E>          implements List<E>, RandomAccess, Cloneable, java.io.Serializable  {      private static final long serialVersionUID = 8683452581122892189L;      transient Object[] elementData; // non-private to simplify nested class access      private int size;  } |

笔者省略了其他成员变量，从上面的代码中可以知道ArrayList实现了java.io.Serializable接口，那么我们就可以对它进行序列化及反序列化。因为elementData是transient的，所以我们认为这个成员变量不会被序列化而保留下来。我们写一个Demo，验证一下我们的想法：

code 4

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public static void main(String[] args) throws IOException, ClassNotFoundException {          List<String> stringList = new ArrayList<String>();          stringList.add("hello");          stringList.add("world");          stringList.add("hollis");          stringList.add("chuang");          System.out.println("init StringList" + stringList);          ObjectOutputStream objectOutputStream = new ObjectOutputStream(new FileOutputStream("stringlist"));          objectOutputStream.writeObject(stringList);            IOUtils.close(objectOutputStream);          File file = new File("stringlist");          ObjectInputStream objectInputStream = new ObjectInputStream(new FileInputStream(file));          List<String> newStringList = (List<String>)objectInputStream.readObject();          IOUtils.close(objectInputStream);          if(file.exists()){              file.delete();          }          System.out.println("new StringList" + newStringList);      }  //init StringList[hello, world, hollis, chuang]  //new StringList[hello, world, hollis, chuang] |

了解ArrayList的人都知道，ArrayList底层是通过数组实现的。那么数组elementData其实就是用来保存列表中的元素的。通过该属性的声明方式我们知道，他是无法通过序列化持久化下来的。那么为什么code 4的结果却通过序列化和反序列化把List中的元素保留下来了呢？

**writeObject和readObject方法**

在ArrayList中定义了两个方法： writeObject和readObject。

这里先给出结论:

*在序列化过程中，如果被序列化的类中定义了writeObject 和 readObject 方法，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。*

来看一下这两个方法的具体实现：

code 5

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | private void readObject(java.io.ObjectInputStream s)          throws java.io.IOException, ClassNotFoundException {          elementData = EMPTY\_ELEMENTDATA;            // Read in size, and any hidden stuff          s.defaultReadObject();            // Read in capacity          s.readInt(); // ignored            if (size > 0) {              // be like clone(), allocate array based upon size not capacity              ensureCapacityInternal(size);                Object[] a = elementData;              // Read in all elements in the proper order.              for (int i=0; i<size; i++) {                  a[i] = s.readObject();              }          }      } |

code 6

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | private void writeObject(java.io.ObjectOutputStream s)          throws java.io.IOException{          // Write out element count, and any hidden stuff          int expectedModCount = modCount;          s.defaultWriteObject();            // Write out size as capacity for behavioural compatibility with clone()          s.writeInt(size);            // Write out all elements in the proper order.          for (int i=0; i<size; i++) {              s.writeObject(elementData[i]);          }            if (modCount != expectedModCount) {              throw new ConcurrentModificationException();          }      } |

那么为什么ArrayList要用这种方式来实现序列化呢？

**why transient**

ArrayList实际上是动态数组，每次在放满以后自动增长设定的长度值，如果数组自动增长长度设为100，而实际只放了一个元素，那就会序列化99个null元素。为了保证在序列化的时候不会将这么多null同时进行序列化，ArrayList把元素数组设置为transient。

**why writeObject and readObject**

前面说过，为了防止一个包含大量空对象的数组被序列化，为了优化存储，所以，ArrayList使用transient来声明elementData。 但是，作为一个集合，在序列化过程中还必须保证其中的元素可以被持久化下来，所以，通过重写writeObject 和 readObject方法的方式把其中的元素保留下来。

writeObject方法把elementData数组中的元素遍历的保存到输出流（ObjectOutputStream）中。

readObject方法从输入流（ObjectInputStream）中读出对象并保存赋值到elementData数组中。

至此，我们先试着来回答刚刚提出的问题：

*如何自定义的序列化和反序列化策略*

答：可以通过在被序列化的类中增加writeObject 和 readObject方法。那么问题又来了：

*虽然ArrayList中写了writeObject 和 readObject 方法，但是这两个方法并没有显示的被调用啊。*

***那么如果一个类中包含writeObject 和 readObject 方法，那么这两个方法是怎么被调用的呢?***

**ObjectOutputStream**

从code 4中，我们可以看出，对象的序列化过程通过ObjectOutputStream和ObjectInputputStream来实现的，那么带着刚刚的问题，我们来分析一下ArrayList中的writeObject 和 readObject 方法到底是如何被调用的呢？

为了节省篇幅，这里给出ObjectOutputStream的writeObject的调用栈：

writeObject ---> writeObject0 --->writeOrdinaryObject--->writeSerialData--->invokeWriteObject

这里看一下invokeWriteObject：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | void invokeWriteObject(Object obj, ObjectOutputStream out)          throws IOException, UnsupportedOperationException      {          if (writeObjectMethod != null) {              try {                  writeObjectMethod.invoke(obj, new Object[]{ out });              } catch (InvocationTargetException ex) {                  Throwable th = ex.getTargetException();                  if (th instanceof IOException) {                      throw (IOException) th;                  } else {                      throwMiscException(th);                  }              } catch (IllegalAccessException ex) {                  // should not occur, as access checks have been suppressed                  throw new InternalError(ex);              }          } else {              throw new UnsupportedOperationException();          }      } |

其中writeObjectMethod.invoke(obj, new Object[]{ out });是关键，通过反射的方式调用writeObjectMethod方法。官方是这么解释这个writeObjectMethod的：

*class-defined writeObject method, or null if none*

在我们的例子中，这个方法就是我们在ArrayList中定义的writeObject方法。通过反射的方式被调用了。

至此，我们先试着来回答刚刚提出的问题：

***如果一个类中包含writeObject 和 readObject 方法，那么这两个方法是怎么被调用的?***

答：在使用ObjectOutputStream的writeObject方法和ObjectInputStream的readObject方法时，会通过反射的方式调用。

至此，我们已经介绍完了ArrayList的序列化方式。那么，不知道有没有人提出这样的疑问：

***Serializable明明就是一个空的接口，它是怎么保证只有实现了该接口的方法才能进行序列化与反序列化的呢？***

Serializable接口的定义：

|  |  |
| --- | --- |
| 1  2 | public interface Serializable {  } |

读者可以尝试把code 1中的继承Serializable的代码去掉，再执行code 2，会抛出java.io.NotSerializableException。

其实这个问题也很好回答，我们再回到刚刚ObjectOutputStream的writeObject的调用栈：

writeObject ---> writeObject0 --->writeOrdinaryObject--->writeSerialData--->invokeWriteObject

writeObject0方法中有这么一段代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | 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());                  }              } |

在进行序列化操作时，会判断要被序列化的类是否是Enum、Array和Serializable类型，如果不是则直接抛出NotSerializableException

1. 单例与序列化

单例模式真的能够实现实例的唯一性吗？

答案是否定的，很多人都知道使用反射可以破坏单例模式，除了反射以外，使用序列化与反序列化也同样会破坏单例。

**序列化对单例的破坏**

首先来写一个单例的类：

code 1

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | package com.hollis;  import java.io.Serializable;  /\*\*   \* Created by hollis on 16/2/5.   \* 使用双重校验锁方式实现单例   \*/  public class Singleton implements Serializable{      private volatile static Singleton singleton;      private Singleton (){}      public static Singleton getSingleton() {          if (singleton == null) {              synchronized (Singleton.class) {                  if (singleton == null) {                      singleton = new Singleton();                  }              }          }          return singleton;      }  } |

接下来是一个测试类：

code 2

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | package com.hollis;  import java.io.\*;  /\*\*   \* Created by hollis on 16/2/5.   \*/  public class SerializableDemo1 {      //为了便于理解，忽略关闭流操作及删除文件操作。真正编码时千万不要忘记      //Exception直接抛出      public static void main(String[] args) throws IOException, ClassNotFoundException {          //Write Obj to file          ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("tempFile"));          oos.writeObject(Singleton.getSingleton());          //Read Obj from file          File file = new File("tempFile");          ObjectInputStream ois =  new ObjectInputStream(new FileInputStream(file));          Singleton newInstance = (Singleton) ois.readObject();          //判断是否是同一个对象          System.out.println(newInstance == Singleton.getSingleton());      }  }  //false |

输出结构为false，说明：

*通过对Singleton的序列化与反序列化得到的对象是一个新的对象，这就破坏了Singleton的单例性。*

这里，在介绍如何解决这个问题之前，我们先来深入分析一下，为什么会这样？在反序列化的过程中到底发生了什么。

**ObjectInputStream**

对象的序列化过程通过ObjectOutputStream和ObjectInputputStream来实现的，那么带着刚刚的问题，分析一下ObjectInputputStream 的readObject 方法执行情况到底是怎样的。

为了节省篇幅，这里给出ObjectInputStream的readObject的调用栈：

readObject--->readObject0--->readOrdinaryObject--->checkResolve

这里看一下重点代码，readOrdinaryObject方法的代码片段：  
code 3

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | private Object readOrdinaryObject(boolean unshared)          throws IOException      {          //此处省略部分代码            Object obj;          try {              obj = desc.isInstantiable() ? desc.newInstance() : null;          } catch (Exception ex) {              throw (IOException) new InvalidClassException(                  desc.forClass().getName(),                  "unable to create instance").initCause(ex);          }            //此处省略部分代码            if (obj != null &&              handles.lookupException(passHandle) == null &&              desc.hasReadResolveMethod())          {              Object rep = desc.invokeReadResolve(obj);              if (unshared && rep.getClass().isArray()) {                  rep = cloneArray(rep);              }              if (rep != obj) {                  handles.setObject(passHandle, obj = rep);              }          }            return obj;      } |

code 3 中主要贴出两部分代码。先分析第一部分：

code 3.1

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Object obj;  try {      obj = desc.isInstantiable() ? desc.newInstance() : null;  } catch (Exception ex) {      throw (IOException) new InvalidClassException(desc.forClass().getName(),"unable to create instance").initCause(ex);  } |

这里创建的这个obj对象，就是本方法要返回的对象，也可以暂时理解为是ObjectInputStream的readObject返回的对象。

*isInstantiable：如果一个serializable/externalizable的类可以在运行时被实例化，那么该方法就返回true。针对serializable和externalizable我会在其他文章中介绍。*

*desc.newInstance：该方法通过反射的方式调用无参构造方法新建一个对象。*

所以。到目前为止，也就可以解释，为什么序列化可以破坏单例了？

*答：序列化会通过反射调用无参数的构造方法创建一个新的对象。*

那么，接下来我们再看刚开始留下的问题，如何防止序列化/反序列化破坏单例模式。

**防止序列化破坏单例模式**

先给出解决方案，然后再具体分析原理：

只要在Singleton类中定义readResolve就可以解决该问题：

code 4

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | package com.hollis;  import java.io.Serializable;  /\*\*   \* Created by hollis on 16/2/5.   \* 使用双重校验锁方式实现单例   \*/  public class Singleton implements Serializable{      private volatile static Singleton singleton;      private Singleton (){}      public static Singleton getSingleton() {          if (singleton == null) {              synchronized (Singleton.class) {                  if (singleton == null) {                      singleton = new Singleton();                  }              }          }          return singleton;      }        private Object readResolve() {          return singleton;      }  } |

具体原理，我们回过头继续分析code 3中的第二段代码:

code 3.2

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | if (obj != null &&              handles.lookupException(passHandle) == null &&              desc.hasReadResolveMethod())          {              Object rep = desc.invokeReadResolve(obj);              if (unshared && rep.getClass().isArray()) {                  rep = cloneArray(rep);              }              if (rep != obj) {                  handles.setObject(passHandle, obj = rep);              }          } |

hasReadResolveMethod:如果实现了serializable 或者 externalizable接口的类中包含readResolve则返回true

invokeReadResolve:通过反射的方式调用要被反序列化的类的readResolve方法。

所以，原理也就清楚了，主要在Singleton中定义readResolve方法，并在该方法中指定要返回的对象的生成策略，就可以方式单例被破坏。

**总结**

在涉及到序列化的场景时，要格外注意他对单例的破坏。

1. 关于 Java 对象序列化您不知道的 5 件事
   1. 序列化允许[重构](http://www.amazon.cn/gp/product/B003BY6PLK/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&tag=importnew-23&linkCode=as2&camp=536&creative=3200&creativeASIN=B003BY6PLK)

序列化允许一定数量的类变种，甚至重构之后也是如此，ObjectInputStream 仍可以很好地将其读出来。

Java Object Serialization 规范可以自动管理的关键任务是：

* 将新字段添加到类中
* 将字段从 static 改为非 static
* 将字段从 transient 改为非 transient

取决于所需的向后兼容程度，转换字段形式（从非 static 转换为 static 或从非 transient 转换为 transient）或者删除字段需要额外的消息传递。

### 重构序列化类

既然已经知道序列化允许重构，我们来看看当把新字段添加到 Person 类中时，会发生什么事情。

如清单 3 所示，PersonV2 在原先 Person 类的基础上引入一个表示性别的新字段。

##### 清单 3. 将新字段添加到序列化的 Person 中

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | enum Gender  {      MALE, FEMALE  }    public class Person      implements java.io.Serializable  {      public Person(String fn, String ln, int a, Gender g)      {          this.firstName = fn; this.lastName = ln; this.age = a; this.gender = g;      }        public String getFirstName() { return firstName; }      public String getLastName() { return lastName; }      public Gender getGender() { return gender; }      public int getAge() { return age; }      public Person getSpouse() { return spouse; }        public void setFirstName(String value) { firstName = value; }      public void setLastName(String value) { lastName = value; }      public void setGender(Gender value) { gender = value; }      public void setAge(int value) { age = value; }      public void setSpouse(Person value) { spouse = value; }        public String toString()      {          return "[Person: firstName=" + firstName +              " lastName=" + lastName +              " gender=" + gender +              " age=" + age +              " spouse=" + spouse.getFirstName() +              "]";      }        private String firstName;      private String lastName;      private int age;      private Person spouse;      private Gender gender;  } |

序列化使用一个 hash，该 hash 是根据给定源文件中几乎所有东西 — 方法名称、字段名称、字段类型、访问修改方法等 — 计算出来的，序列化将该 hash 值与序列化流中的 hash 值相比较。

为了使 Java 运行时相信两种类型实际上是一样的，第二版和随后版本的 Person 必须与第一版有相同的序列化版本 hash（存储为 private static final serialVersionUID 字段）。因此，我们需要 serialVersionUID 字段，它是通过对原始（或 V1）版本的 Person 类运行 JDK serialver命令计算出的。

一旦有了 Person 的 serialVersionUID，不仅可以从原始对象 Person 的序列化数据创建 PersonV2 对象（当出现新字段时，新字段被设为缺省值，最常见的是“null”），还可以反过来做：即从 PersonV2 的数据通过反序列化得到 Person，这毫不奇怪。

* 1. 序列化并不安全

让 Java 开发人员诧异并感到不快的是，序列化二进制格式完全编写在文档中，并且完全可逆。实际上，只需将二进制序列化流的内容转储到控制台，就足以看清类是什么样子，以及它包含什么内容。

这对于安全性有着不良影响。例如，当通过 RMI 进行远程方法调用时，通过连接发送的对象中的任何 private 字段几乎都是以明文的方式出现在套接字流中，这显然容易招致哪怕最简单的安全问题。

幸运的是，序列化允许 “hook” 序列化过程，并在序列化之前和反序列化之后保护（或模糊化）字段数据。可以通过在 Serializable 对象上提供一个 writeObject 方法来做到这一点。

### 模糊化序列化数据

假设 Person 类中的敏感数据是 age 字段。毕竟，女士忌谈年龄。 我们可以在序列化之前模糊化该数据，将数位循环左移一位，然后在反序列化之后复位。（您可以开发更安全的算法，当前这个算法只是作为一个例子。）

为了 “hook” 序列化过程，我们将在 Person 上实现一个 writeObject 方法；为了 “hook” 反序列化过程，我们将在同一个类上实现一个readObject 方法。重要的是这两个方法的细节要正确 — 如果访问修改方法、参数或名称不同于清单 4 中的内容，那么代码将不被察觉地失败，Person 的 age 将暴露。

##### 清单 4. 模糊化序列化数据

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | public class Person      implements java.io.Serializable  {      public Person(String fn, String ln, int a)      {          this.firstName = fn; this.lastName = ln; this.age = a;      }        public String getFirstName() { return firstName; }      public String getLastName() { return lastName; }      public int getAge() { return age; }      public Person getSpouse() { return spouse; }        public void setFirstName(String value) { firstName = value; }      public void setLastName(String value) { lastName = value; }      public void setAge(int value) { age = value; }      public void setSpouse(Person value) { spouse = value; }        private void writeObject(java.io.ObjectOutputStream stream)          throws java.io.IOException      {          // "Encrypt"/obscure the sensitive data          age = age << 2;          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream)          throws java.io.IOException, ClassNotFoundException      {          stream.defaultReadObject();            // "Decrypt"/de-obscure the sensitive data          age = age << 2;      }        public String toString()      {          return "[Person: firstName=" + firstName +              " lastName=" + lastName +              " age=" + age +              " spouse=" + (spouse!=null ? spouse.getFirstName() : "[null]") +              "]";      }        private String firstName;      private String lastName;      private int age;      private Person spouse;  } |

如果需要查看被模糊化的数据，总是可以查看序列化数据流/文件。而且，由于该格式被完全文档化，即使不能访问类本身，也仍可以读取序列化流中的内容。

* 1. 序列化的数据可以被签名和密封

上一个技巧假设您想模糊化序列化数据，而不是对其加密或者确保它不被修改。当然，通过使用 writeObject 和 readObject 可以实现密码加密和签名管理，但其实还有更好的方式。

如果需要对整个对象进行加密和签名，最简单的是将它放在一个 javax.crypto.SealedObject 和/或 java.security.SignedObject 包装器中。两者都是可序列化的，所以将对象包装在 SealedObject 中可以围绕原对象创建一种 “包装盒”。必须有对称密钥才能解密，而且密钥必须单独管理。同样，也可以将 SignedObject 用于数据验证，并且对称密钥也必须单独管理。

结合使用这两种对象，便可以轻松地对序列化数据进行密封和签名，而不必强调关于数字签名验证或加密的细节。很简洁，是吧？

* 1. 序列化允许将代理放在流中

很多情况下，类中包含一个核心数据元素，通过它可以派生或找到类中的其他字段。在此情况下，没有必要序列化整个对象。可以将字段标记为 transient，但是每当有方法访问一个字段时，类仍然必须显式地产生代码来检查它是否被初始化。

如果首要问题是序列化，那么最好指定一个 flyweight 或代理放在流中。为原始 Person 提供一个 writeReplace 方法，可以序列化不同类型的对象来代替它。类似地，如果反序列化期间发现一个 readResolve 方法，那么将调用该方法，将替代对象提供给调用者。

### 打包和解包代理

writeReplace 和 readResolve 方法使 Person 类可以将它的所有数据（或其中的核心数据）打包到一个 PersonProxy 中，将它放入到一个流中，然后在反序列化时再进行解包。

##### 清单 5. 你完整了我，我代替了你

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68 | class PersonProxy      implements java.io.Serializable  {      public PersonProxy(Person orig)      {          data = orig.getFirstName() + "," + orig.getLastName() + "," + orig.getAge();          if (orig.getSpouse() != null)          {              Person spouse = orig.getSpouse();              data = data + "," + spouse.getFirstName() + "," + spouse.getLastName() + ","                + spouse.getAge();          }      }        public String data;      private Object readResolve()          throws java.io.ObjectStreamException      {          String[] pieces = data.split(",");          Person result = new Person(pieces[0], pieces[1], Integer.parseInt(pieces[2]));          if (pieces.length > 3)          {              result.setSpouse(new Person(pieces[3], pieces[4], Integer.parseInt                (pieces[5])));              result.getSpouse().setSpouse(result);          }          return result;      }  }    public class Person      implements java.io.Serializable  {      public Person(String fn, String ln, int a)      {          this.firstName = fn; this.lastName = ln; this.age = a;      }        public String getFirstName() { return firstName; }      public String getLastName() { return lastName; }      public int getAge() { return age; }      public Person getSpouse() { return spouse; }        private Object writeReplace()          throws java.io.ObjectStreamException      {          return new PersonProxy(this);      }        public void setFirstName(String value) { firstName = value; }      public void setLastName(String value) { lastName = value; }      public void setAge(int value) { age = value; }      public void setSpouse(Person value) { spouse = value; }        public String toString()      {          return "[Person: firstName=" + firstName +              " lastName=" + lastName +              " age=" + age +              " spouse=" + spouse.getFirstName() +              "]";      }        private String firstName;      private String lastName;      private int age;      private Person spouse;  } |

注意，PersonProxy 必须跟踪 Person 的所有数据。这通常意味着代理需要是 Person 的一个内部类，以便能访问 private 字段。有时候，代理还需要追踪其他对象引用并手动序列化它们，例如 Person 的 spouse。

这种技巧是少数几种不需要读/写平衡的技巧之一。例如，一个类被重构成另一种类型后的版本可以提供一个 readResolve 方法，以便静默地将被序列化的对象转换成新类型。类似地，它可以采用 writeReplace 方法将旧类序列化成新版本。

* 1. 信任，但要验证

认为序列化流中的数据总是与最初写到流中的数据一致，这没有问题。但是，正如一位美国前总统所说的，“信任，但要验证”。

对于序列化的对象，这意味着验证字段，以确保在反序列化之后它们仍具有正确的值，“以防万一”。为此，可以实现 ObjectInputValidation接口，并覆盖 validateObject() 方法。如果调用该方法时发现某处有错误，则抛出一个 InvalidObjectException。