Java 基础提升篇：深入分析 Java 的序列化与反序列化

初遇

序列化是一种对象持久化的手段。普遍应用在网络传输、RMI 等场景中。本文通过分析 ArrayList 的序列化来介绍 Java 序列化的相关内容。主要涉及到以下几个问题：

- 怎么实现 Java 的序列化
- 为什么实现了 java.io.Serializable 接口才能被序列化
- transient 的作用是什么
- 怎么自定义序列化策略
- 自定义的序列化策略是如何被调用的
- ArrayList 对序列化的实现有什么好处

Java 对象的序列化

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**。

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，在本文的后续章节中将会陆续讲到。

```
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;
}
```

如何对 Java 对象进行序列化与反序列化

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。这里先来一段代码：

code 1 创建一个 User 类，用于序列化及反序列化

```
import java.io.Serializable;
import java.util.Date;
/**
 * Created by hollis on 16/2/2.
 */
public class User implements Serializable{
    private String name;
    private int age;
    private Date birthday;
    private transient String gender;
    private static final long serialVersionUID = -6849794470754667710L;
    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;
    }
    public Date getBirthday() {
        return birthday;
    }
    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }
```

```
    public String getGender() {
        return gender;
    }
    public void setGender(String gender) {
        this.gender = gender;
    }
    @Override
    public String toString() {
        return "User{" +
                "name='" + name + '/'' +
                ", age=" + age +
                ", gender=" + gender +
                ", birthday=" + birthday +
                '}';
    }
}
```

code 2 对 User 进行序列化及反序列化的 Demo

```
import org.apache.commons.io.FileUtils;
import org.apache.commons.io.IOUtils;
import java.io.*;
import java.util.Date;
/**
 * Created by hollis on 16/2/2.
 */
public class SerializableDemo {
    public static void main(String[] args) {
        //Initializes The Object
        User user = new User();
        user.setName("hollis");
        user.setGender("male");
        user.setAge(23);
        user.setBirthday(new Date());
        System.out.println(user);
        //Write Obj to File
        ObjectOutputStream oos = null;
        try {
            oos = new ObjectOutputStream(new FileOutputStream("tempFile"));
```

```
            oos.writeObject(user);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            IOUtils.closeQuietly(oos);
        }
        //Read Obj from File
        File file = new File("tempFile");
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(new FileInputStream(file));
            User newUser = (User) ois.readObject();
            System.out.println(newUser);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            IOUtils.closeQuietly(ois);
            try {
                FileUtils.forceDelete(file);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
//output
//User{name='hollis', age=23, gender=male, birthday=Tue Feb 02 17:37:38 CST 2016}
//User{name='hollis', age=23, gender=null, birthday=Tue Feb 02 17:37:38 CST 2016}
```

序列化及反序列化相关知识

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。

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

```
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

```
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 中定义了来个方法： 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

```
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

```
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：

```
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：

```
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：

```
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 。

总结

1) 如果一个类想被序列化，需要实现 Serializable 接口。否则将抛出 NotSerializableException 异常，这是因为，在序列化操作过程中会对类型进行检查 要求被序列化的类必须属于 Enum、Array 和 Serializable 类型其中的任何一种。
2) 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。
3) 在类中增加 writeObject 和 readObject 方法可以实现自定义序列化策略。