

深入分析Java的序列化与反序列化

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

序列化应用场景

当你想把的内存中的对象状态保存到一个文件中或者数据库中时候。

当你想用套接字在网络上传送对象的时候。网络底层传输的是比特流

当你想通过RMI传输对象的时候。

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

```
* Created by hollis on 16/2/2.
*/
public class User implements Serializable{ 实现Serializable就可以序列化
    private String name;
    private int age;
    private Date birthday;
    private transient String gender; 被transient修饰的属性, 不参与序列化过程, 不被序列化到文件中。
    private static final long serialVersionUID = -6849794470754667710L; static修饰的也不被序列化

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    } 剩下的三个属性的get,set方法这里省略了

    @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

```

package com.hollis;
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 序列化内存中对象状态到文件中
        ObjectOutputStream oos = null;
        oos = new ObjectOutputStream(new FileOutputStream("tempFile"));
        oos.writeObject(user);
        //Read Obj from File 从文件中反序列化出对象
        File file = new File("tempFile");
        ObjectInputStream ois = null;
        ois = new ObjectInputStream(new FileInputStream(file));
        User newUser = (User) ois.readObject();
        System.out.println(newUser)
        //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}
        user==newUser 结果是false
    }
}

```

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`。

通过自定义序列化规则，可以“打破”`transient`的规则，比如`arraylist`

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] // stringList原始对象不等于反序列化后的newStringList对象

```

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

```
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(); 从输入流 (ObjectInputStream) 中读出对象并保存赋值到elementData 数组
                                   中。因为反序列化时候被transient修饰的变量，默认是null
        }
    }
}
```

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]); 把 elementData 数组中的元素遍历的保存到输出流ObjectOutputStream
    }

    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。

什么时候需要设计自定义序列化策略呢？

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 ObjectInputStream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();
    }
}
```

通过反射序列化对象判断是不是重写了writeObject策略，利用反射的动态调用特性，反射执行对象的序列化方法。为什么不使用obj.writeObject？
ObjectOutputStream.write序列化对象过程中，通过反射调用对象的writeObject方法，执行自定义的序列化方案。

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

参考资料

[Java 序列化的高级认识 \(https://www.ibm.com/developerworks/cn/java/j-lo-serial/\)](https://www.ibm.com/developerworks/cn/java/j-lo-serial/)

序列化与反序列化

保存的是对象“状态”并不是完整对象

序列化 (Serialization)是将对象的状态信息转换为可以存储或传输的形式过程。一般将一个对象存储至一个储存媒介，例如档案或是记忆体缓冲等。在网络传输过程中，可以是字节或是XML等格式。而字节的或XML编码格式可以还原完全相等的对象。这个相反的过程又称为反序列化。

Java对象的序列化与反序列化

在Java中，我们可以通过多种方式来创建对象，并且只要对象没有被回收我们都可以复用该对象。但是，我们创建出来的这些Java对象都是存在于JVM的堆内存中的。只有JVM处于运行状态的时候，这些对象才可能存在。一旦JVM停止运行，这些对象的状态也就随之而丢失了。

但是在真实的应用场景中，我们需要将这些对象持久化下来，并且能够在需要的时候把对象重新读取出来。Java的对象序列化可以帮助我们实现该功能。

对象序列化机制（object serialization）是Java语言内建的一种对象持久化方式，通过对象序列化，可以把对象的状态保存为字节数组，并且可以在有需要的时候将这个字节数组通过反序列化的方式再转换成对象。对象序列化可以很容易的在JVM中的活动对象和字节数组（流）之间进行转换。

在Java中，对象的序列化与反序列化被广泛应用到RMI(远程方法调用)及网络传输中。

相关接口及类

Java为了方便开发人员将Java对象进行序列化及反序列化提供了一套方便的API来支持。其中包括以下接口和类：



java.io.Serializable 序列化接口

java.io.Externalizable

ObjectOutput ObjectInput

ObjectOutputStream

ObjectInputStream

Serializable 接口

ObjectOutputStream.writeObject 序列化对象过程中，会判断被序列化的对象是否实现 Serializable 接口或者 Enum 类，否则抛出异常，对象不能序列化

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。**(该接口并没有方法和字段，为什么只有实现了该接口的类的对象才能被序列化呢？)
(<http://www.hollischuang.com/archives/1140#What%20Serializable%20Did>))

当试图对一个对象进行序列化的时候，如果遇到不支持 `Serializable` 接口的对象。在此情况下，将抛出 `NotSerializableException`。

如果要序列化的类有父类，要想同时将在父类中定义过的变量持久化下来，那么父类也应该集成 `java.io.Serializable` 接口。

下面是一个实现了 `java.io.Serializable` 接口的类

```


*/
public class User1 implements Serializable {

    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;
    }

    @Override
    public String toString() {
        return "User{" +
            "name='" + name + '\'' +
            ", age=" + age +
            '}';
    }
}


```

通过下面的代码进行序列化及反序列化

```
package com.hollischaung.serialization.SerializableDemos;

import org.apache.commons.io.FileUtils;
import org.apache.commons.io.IOUtils;

import java.io.*;

/**
 * Created by hollis on 16/2/17.
 * SerializableDemo1 结合SerializableDemo2 说明 一个类要想被序列化必须实现Serializable接口
 */
public class SerializableDemo1 {

    public static void main(String[] args) {
        //Initializes The Object
        User1 user = new User1();
        user.setName("hollis");
        user.setAge(23);
        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);
        }
    }
}
```

更多关于Serializable的使用，请参考代码实例 (<https://github.com/hollischaung/java-demo/tree/master/src/main/java/com/hollischaung/serialization/SerializableDemos>).

Externalizable接口

除了Serializable 之外，java中还提供了另一个序列化接口 Externalizable

为了了解Externalizable接口和Serializable接口的区别，先来看代码，我们把上面的代码改成使用Externalizable的形式。



```
package com.hollischaung.serialization.ExternalizableDemos;
```

```
import java.io.Externalizable;  
import java.io.IOException;  
import java.io.ObjectInput;  
import java.io.ObjectOutput;
```

```
/**  
 * Created by hollis on 16/2/17.  
 * 实现Externalizable接口  
 */
```

```
public class User1 implements Externalizable {
```




```
    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;  
    }
```




```

package com.hollischaung.serialization.ExternalizableDemos;

import java.io.*;

/**
 * Created by hollis on 16/2/17.
 */
public class ExternalizableDemo1 {

    //为了便于理解和节省篇幅，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
    //IOException直接抛出
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        //Write Obj to file
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("tempFile"));
        User1 user = new User1();
        user.setName("hollis");
        user.setAge(23);
        oos.writeObject(user);
        //Read Obj from file
        File file = new File("tempFile");
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file));
        User1 newInstance = (User1) ois.readObject();
        //output
        System.out.println(newInstance);
    }
}

//OutPut:
//User{name='null', age=0}

```

通过上面的实例可以发现，对User1类进行序列化及反序列化之后得到的对象的所有属性的值都变成了默认值。也就是说，之前的那个对象的状态并没有被持久化下来。这就是Externalizable接口和Serializable接口的区别：

Externalizable 继承了 Serializable，该接口中定义了两个抽象方法：writeExternal() 与 readExternal()。当使用Externalizable接口来进行序列化与反序列化的时候需要开发人员重写 writeExternal() 与 readExternal() 方法。由于上面的代码中，并没有在这两个方法中定义序列化实现细节，所以输出的内容为空。还有一点值得注意：在使用Externalizable进行序列化的时候，在读取对象时，会调用被序列化类的无参构造器去创建一个新的对象，然后再将被保存对象的字段的值分别填充到新对象中。所以，实现Externalizable接口的类必须要提供一个public的无参的构造器。

按照要求修改之后代码如下：

```
package com.hollischaung.serialization.ExternalizableDemos;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

/**
 * Created by hollis on 16/2/17.
 * 实现Externalizable接口, 并实现writeExternal 和readExternal 方法
 */
public class User2 implements Externalizable {

    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;
    }
}
```

```
package com.hollischaung.serialization.ExternalizableDemos;
```

```
import java.io.*;
```

```
/**  
 * Created by hollis on 16/2/17.  
 */
```

```
public class ExternalizableDemo2 {
```

```
    //为了便于理解和节省篇幅，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
```

```
    //IOException直接抛出
```

```
    public static void main(String[] args) throws IOException, ClassNotFoundException {
```

```
        //Write Obj to file
```

```
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("tempFile"));
```

```
        User2 user = new User2();
```

```
        user.setName("hollis");
```

```
        user.setAge(23);
```

```
        oos.writeObject(user);
```

```
        //Read Obj from file
```

```
        File file = new File("tempFile");
```

```
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file));
```

```
        User2 newInstance = (User2) ois.readObject();
```

```
        //output
```

```
        System.out.println(newInstance);
```

```
    }
```

```
}
```

```
//OutPut:
```

```
//User{name='hollis', age=23}
```

这次，就可以把之前的对象状态持久化下来了。

如果 User 类中没有无参数的构造函数，在运行时会抛出异常：
java.io.InvalidClassException

更多 Externalizable 接口使用实例请参考 [代码实例](https://github.com/hollischaung/java-demo/tree/master/src/main/java/com/hollischaung/serialization/ExternalizableDemos) (https://github.com/hollischaung/java-demo/tree/master/src/main/java/com/hollischaung/serialization/ExternalizableDemos).

ObjectOutput和ObjectInput 接口

ObjectInput接口 扩展自 DataInput 接口以包含对象的读操作。

DataInput 接口用于从二进制流中读取字节，并根据所有 Java 基本类型数据进行重构。同时还提供根据 UTF-8 修改版格式的数据重构 String 的工具。

对于此接口中的所有数据读取例程来说，如果在读取所需字节数之前已经到达文件末尾 (end of file)，则将抛出 EOFException (IOException 的一种)。如果因为到达文件末尾以外的其他原因无法读取字节，则将抛出 IOException 而不是 EOFException。尤其是，在输入流已关闭的情况下，将抛出 IOException。

ObjectOutput 扩展 DataOutput 接口以包含对象的写入操作。

DataOutput 接口用于将数据从任意 Java 基本类型转换为一系列字节，并将这些字节写入二进制流。同时还提供了一个将 String 转换成 UTF-8 修改版格式并写入所得到的系列字节的工具。

对于此接口中写入字节的所有方法，如果由于某种原因无法写入某个字节，则抛出 IOException。

ObjectOutputStream类和ObjectInputStream类

通过前面的代码片段中我们也能知道，我们一般使用ObjectOutputStream的 writeObject 方法把一个对象进行持久化。再使用ObjectInputStream的 readObject 从持久化存储中把对象读取出来。

更多关于ObjectInputStream和ObjectOutputStream的相关知识欢迎阅读我的另外两篇博文：[深入分析Java的序列化与反序列化 \(http://www.hollischuang.com/archives/1140\)](http://www.hollischuang.com/archives/1140)、[单例与序列化的那些事儿 \(http://www.hollischuang.com/archives/1144\)](http://www.hollischuang.com/archives/1144)。

Transient 关键字

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。关于Transient 关键字的拓展知识欢迎阅读[深入分析Java的序列化与反序列化 \(http://www.hollischuang.com/archives/1140\)](http://www.hollischuang.com/archives/1140)。

序列化ID

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`）

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。

参考资料

>

单例与序列化的那些事儿 (<https://www.hollischuang.com/archives/1144>)

单例模式，是设计模式中最简单的一种。通过单例模式可以保证系统中一个类只有一个实例而且该实例易于外界访问，从而方便对实例个数的控制并节约系统资源。如果希望在系统中某个类的对象只能存在一个，单例模式是最好的解决方案。关于单例模式的使用方式，可以阅读[单例模式的七种写法](http://www.hollischuang.com/archives/205) (<http://www.hollischuang.com/archives/205>)。

但是，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

反序列化就是通过反射调用对象的空参构造器，即使私有构造器也可以

首先来写一个单例的类：

code 1

```

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

```

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：

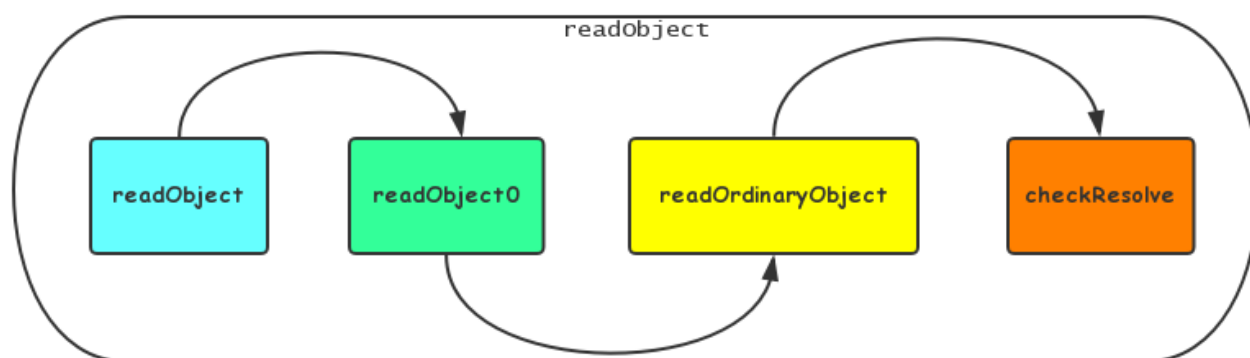
通过对Singleton的序列化与反序列化得到的对象是一个新的对象，这就破坏了Singleton的单例性。

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ObjectInputStream来实现的，那么带着刚刚的问题，分析一下ObjectInputStream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：



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


```
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);
        }
    }

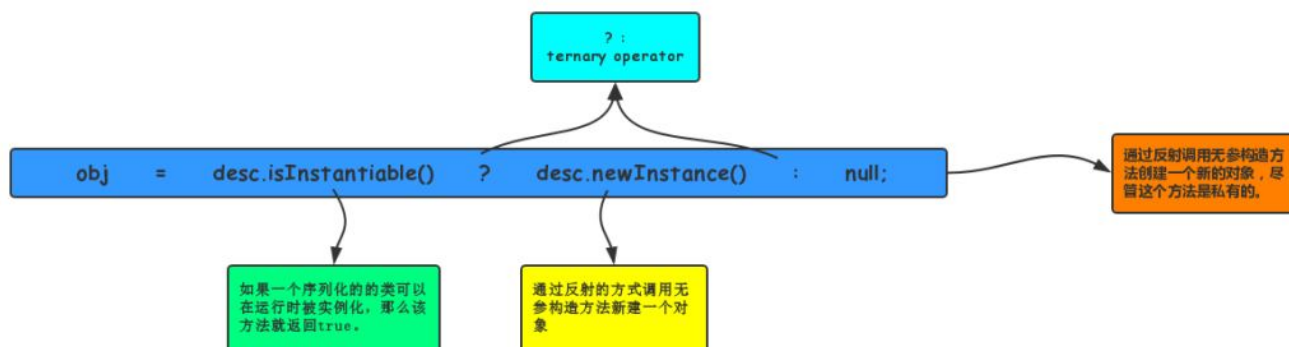
    return obj;
}
```

code 3 中主要贴出两部分代码。先分析第一部分：

code 3.1

```
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。对象的属性也通过反射赋值，这就是反射的好处，运行时对成员变量赋值，即使没有提供get,set方法

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

```

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() { readResovle在反序列化过程中给会被调用，赋值给反序列化的对象
        return singleton;
    }
}

```

还是运行以下测试类：

```

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());
    }
}
//true

```

本次输出结果为true。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

```
if (obj != null &&
    handles.lookupException(passHandle) == null &&
    desc.hasReadResolveMethod())
{
    Object rep = desc.invokeReadResolve(obj); 反序列化过程中执行自定义的
    ReadResolve方法，赋值给反序列化后的对象引用过
    if (unshared && rep.getClass().isArray()) {
        rep = cloneArray(rep);
    }
    if (handles.getObject(passHandle, obj) != rep);
}
```

编译时根本无法知道该对象或类可能属于哪些类，程序只依靠运行时信息来发现该对象和类的真实信息。反射优点，只有运行时刻才知道object具体是哪一个类，在调用他的resolve方法

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，就可以防止单例被破坏。

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。

推荐阅读

深入分析Java的序列化与反序列化 (<http://www.hollischuang.com/archives/1140>).