Java 中序列化使用总结

作者：leopoldwu
原文链接：https://ld246.com/article/1498751975069
来源网站：链滴

Java中序列化使用总结

[TOC]

---

1. 序列化与反序列化

transient关键字与序列化有关，所以先介绍以下序列化：

序列化是一种处理对象流的机制，它可以将对象转化成二进制字节流以便进行存储或网络传输。

反序列化正好与序列化相反，它能够将二进制字节流重新转化成对象。

1.1 序列化流程

- 根据某种序列化算法（可自定义，也可以使用JDK提供的算法）将对象转化成字节流
- 将字节流写入到数据流载体中
- 使用输出流存储或传输

1.2 反序列化流程

- 从存储介质中或者网络上获取对象的字节流
- 把字节流读取到流载体中
- 通过反序列化将字节流翻译成对象

1.3 示例

```
// 实现Serializable使对象能够序列化
public class SerializableClass implements Serializable{
    private static final long serialVersionUID = 1L;
    public int id;
    public String name;
}
// 序列化
public static void serializeClass() throws IOException{
        SerializableClass sc = new SerializableClass();
        FileOutputStream fos = new FileOutputStream("temp.out");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(sc);
        oos.flush();
        oos.close();
}
// 反序列化
public static void deserializeClass() throws IOException, ClassNotFoundException{
        FileInputStream fis = new FileInputStream("temp.out");
        ObjectInputStream ois = new ObjectInputStream(fis);
        SerializableClass sc = (SerializableClass) ois.readObject();
```

```
        System.out.println(sc);
        ois.close();
}
```

1.4 序列化算法

序列化算法的操作过程一般如下：

- 将对象实例相关的类元数据输出
- 递归输出类的超类描述直到不再有超类
- 类元数据输出完成后，开始从最顶层的超类开始输出对象实例的实际数据值

1.5 继承关系序列化

如果父类有实现Serializable接口，则在子类实例被序列化时父类也会被序列化，如果没有实现该接则父类不会被序列化。

1.5 注意问题

最好自己给可序列化类添加一个serialVersionUID，虽然不添加系统也会自动生成，但是如果我们不己添加，当我们序列了一个实例，假设我们把它保存在文件中，接着我们对原本的类增加或移除字段不能修改字段类型），然后在次读取我们保存在文件中的实例，这时会出现版本不一致的错误。

如果我们把一个实例先转化成它的父类型，然后在进行序列化（实际上序列化的对象依然是子类型而是父类型），如果在另一个程序中不存在子类型，只存在它的父类型，那么这时反序列化时会出现错。

2. transient关键字使用

2.1 需要序列化的类成员

静态变量因为是属于类的属性，不属于某个具体的实例，所以静态变量是不需要序列化的；

方法只是一系列操作的集合，也不依赖于对象，所以方法也不需要序列化

普通的成员变量与具体的实例有关，所以需要序列化的是普通的成员变量。

2.2 transient使用

假如我们不希望某个字段序列化，可以在该字段上加上transient关键字，这样在实例序列化时该字段不会被序列化，这样有助于保护一些敏感信息。

3. 自定义序列化

3.1 Serializable接口

3.1.1 自定义序列化方法

前面介绍的都是默认的序列化，如果需要对序列化进行自定义需要用到下面两个方法：

```
private void writeObject(ObjectOutputStream out) throws IOException
private void readObject(ObjectInputStream in) throws IOException,ClassNotFoundException
```

这两个方法是private，所以我们在序列化和反序列化操作时是不需要自己调用这两个方法的，和上面操作是相同的，这两个方法是通过反射的方式被调用的。

可以通过调用 out.defaultWriteObject() 将对象数据以默认的方式写入到数据流中，同样可以通过 in.efaultReadObject()从数据流中读取默认的对象数据，另外可以在这两个方法中进行一些其他操作。

3.1.2 限制序列化对象的数量

序列化会使我们的单例模式失效，为了解决这个问题需要限制序列化对象的数量，可以通过如下两个法解决：

```
// 使用该方法返回的对象作为反序列化的对象
private Object readResolve()
// 使用该方法的返回对象作为序列化的对象
private Object writeReplace()
```

具体例子如下：

```
public class Factory implements Serializable{
     private static Factory instance;
     private Factory(){}
     public static Factory getInstance(){
         if(instance == null){
             instance = new Factory();
         }
         return instance;
     }
     private Object readResolve(){
        return getInstance();
     }
    private Object writeReplace(){
         return getInstance();
    }
}
```

3.2 Externalizable接口

Externalizable接口继承自Serialzable接口，它内部定义了两个方法用于制定序列化和反序列化策略方法如下：

```
// 用于将对象写入输出流中
public void writeExternal(ObjectOutput out) throws IOException;
// JVM使用无参构成方法实例化一个对象，然后调用此方法反序列化一个新对象，所以序列化类必须无参构造方法
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
```

当我们的序列化类继承了这个接口后需要实上面的两个方法，在方法通过把属性写入到输出流或者从入流读取到属性中的方法进行序列化和反序列化，同样我们不需要自己调用这两个方法。

另外，还可以使用writeReplace 和 readResolve这两个方法替代上面的这两个方法。