

Serialization in Java



六尺帐篷 (/u/f8e9b1c246f1)

2017.07.27 12:31 字数 2627 阅读 34 评论 0 喜欢 1

(/u/f8e9b1c246f1)

[编辑文章 \(/writer#/notebooks/10012174/notes/15024562\)](#)

- Serializable in Java
- Class Refactoring with Serialization and serialVersionUID
- Java Externalizable Interface
- Java Serialization Methods
- Serialization with Inheritance
- Serialization Proxy Pattern

我们知道Java对象的生存周期跟GC有关，更宽泛一点讲，JVM关闭了，对象自然也就被销毁了。但是有的时候，我们需要将某些对象保存起来，或者进行传输，以便以后JVM启动的时候，又可以重新获取到对象。这个技术就是对象持久化技术。

Java中的Serialization可以将一个对象转成字节流，我们可以将这个字节流通过网络传输到其他地方，或者保存到文件中，或者存到数据库中。这样就相当于将对象保存下来了。

Java中的Deserialization 就是序列化的反过程，从将字节流中的内容转化成java对象。

Serializable in Java

如果你想要将一个对象序列化，你所需要的做的就是实现java.io.Serializable接口。这个接口是一个marker interface，没有成员变量，没有方法。

Java中对象的序列化是通过ObjectInputStream and ObjectOutputStream两个流实现的。我们将一个对象写入字节流，就需要ObjectOutputStream，从一个字节流读取对象就需要使用ObjectInputStream。

下面我们看一个简单的序列化的例子：



```
import java.io.Serializable;

public class Employee implements Serializable {

    private String name;
    private int id;
    transient private int salary;

    @Override
    public String toString(){
        return "Employee{name="+name+",id="+id+",salary="+salary+"}";
    }

    //getter and setter methods
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getSalary() {
        return salary;
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }

}
```

可以看到这就是一个简单的实现了java.io.Serializable的接口的类，定义了几个属性和getter和setter。

我们看到对于salary变量我们使用了transient修饰符，这个修饰符适用于：如果对于对象中的某些成员变量，我们不想将它序列化，就可以用transient修饰符修饰它，这样它就不会被序列化。

下面，我们将一个序列化的对象保存到文件中，然后从文件反序列读取这个对象，我们需要使用到java.io.Serializable这两个类



```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

/**
 * A simple class with generic serialize and deserialize method implementations
 *
 * @author 刘德华
 *
 */
public class SerializationUtil {

    // deserialize to Object from given file
    public static Object deserialize(String fileName) throws IOException,
        ClassNotFoundException {
        FileInputStream fis = new FileInputStream(fileName);
        ObjectInputStream ois = new ObjectInputStream(fis);
        Object obj = ois.readObject();
        ois.close();
        return obj;
    }

    // serialize the given object and save it to file
    public static void serialize(Object obj, String fileName)
        throws IOException {
        FileOutputStream fos = new FileOutputStream(fileName);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(obj);

        fos.close();
    }
}
```

我们来测试一下序列化的结果

```
import java.io.IOException;

public class SerializationTest {

    public static void main(String[] args) {
        String fileName="employee.ser";
        Employee emp = new Employee();
        emp.setId(100);
        emp.setName("Pankaj");
        emp.setSalary(5000);

        //serialize to file
        try {
            SerializationUtil.serialize(emp, fileName);
        } catch (IOException e) {
            e.printStackTrace();
            return;
        }

        Employee empNew = null;
        try {
            empNew = (Employee) SerializationUtil.deserialize(fileName);
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }

        System.out.println("emp Object::"+emp);
        System.out.println("empNew Object::"+empNew);
    }
}
```

运行结果:

```
emp Object::Employee{name=Pankaj,id=100,salary=5000}
empNew Object::Employee{name=Pankaj,id=100,salary=0}
```



因为salary被声明为transient变量，所以这个成员变量没有被序列化写入到文件中，所以反序列的时候，这个值为初始化的默认0。

类似的是，对于静态变量，static变量，也不会被序列化，因为static变量是属于类的信息，并不属于对象的信息。

Class Refactoring with Serialization and serialVersionUID

java的序列化对于某些对象的变化会忽略掉。

对于下面这些类的变化，不会影响反序列的过程：

- 向类中添加一个新的变量
- 将一个变量从transient转为no-transient
- 将static变量变成 no-static

想要让这些类的变化反映到反序列化的过程中，我们需要在类中定义一个serialVersionUID。我们来写一个测试的方法测试反序列已经序列化的类。

```
import java.io.IOException;

public class DeserializationTest {

    public static void main(String[] args) {

        String fileName="employee.ser";
        Employee empNew = null;

        try {
            empNew = (Employee) SerializationUtil.deserialize(fileName);
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }

        System.out.println("empNew Object::"+empNew);

    }

}
```

我们在运行之前，改变一下类，首先先给类添加一个password属性，并设置getter和setter



```
import java.io.Serializable;

public class Employee implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 7239983150140796558L;
    //private static final long serialVersionUID = -6470090944414208496L;
    private String name;
    private int id;
    transient private int salary;
    private String password;

    @Override
    public String toString(){
        return "Employee(name="+name+",id="+id+",salary="+salary+")";
    }

    //getter and setter methods
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getSalary() {
        return salary;
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

}
```

运行程序，我们发现对反序列没有影响，和没加之前一样

```
java.io.InvalidClassException: Employee; local class incompatible: stream classdesc serialVersionUID = 7239983150140796558L, local class serialVersionUID = -6470090944414208496L
    at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:616)
    at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1829)
    at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1713)
    at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1986)
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1535)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:422)
    at SerializationUtil.deserialize(SerializationUtil.java:22)
    at DeserializationTest.main(DeserializationTest.java:13)
Exception in thread "main" java.lang.NullPointerException
    at DeserializationTest.main(DeserializationTest.java:19)
```

出现错误的原因就是因为，之前序列化的类serialVersionUID 和现在这个新的类的 serialVersionUID 是不同的，实际上如果一个类没有显示定义serialVersionUID，它就会自动根据这个类的成员变量方法，类名等信息，自动计算一个值分配给这个class一个 serialVersionUID。如果你使用IDE的话，你会得到一个警告“The serializable class Employee does not declare a static final serialVersionUID field of type long”

如果我们想要在类改变之后，反序列化的时候，能感知到类的改变，我们就需要显示的制定一个serialVersionUID。



例如，在这个例子中，我们生成一个serialVersionUID，然后将其序列化，运行SerializationTest 程序，然后添加password属性，在运行反序列化的程序，我们发现这次不会报错，而且新添加的属性也会序列化了

Java Externalizable Interface

我们发现序列化的过程是程序自动进行的。有些时候，我们想在序列化的过程中进行一些操作，比如筛选和转换的工作。我们就可以实现 java.io.Externalizable接口，这个接口提供了两个方法，分别可以让我们序列化的时候，和反序列化的时候的操作。
writeExternal() and readExternal()

```
package Externalizable;

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

public class Person implements Externalizable{

    private int id;
    private String name;
    private String gender;

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeInt(id);
        out.writeObject(name+"xyz");
        out.writeObject("abc"+gender);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException,
        ClassNotFoundException {
        id=in.readInt();
        //read in the same order as written
        name=(String) in.readObject();
        if(!name.endsWith("xyz")) throw new IOException("corrupted data");
        name=name.substring(0, name.length()-3);
        gender=(String) in.readObject();
        if(!gender.startsWith("abc")) throw new IOException("corrupted data");
        gender=gender.substring(3);
    }

    @Override
    public String toString(){
        return "Person{id="+id+",name="+name+",gender="+gender+"}";
    }
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }
}
```



可以看到我们将对象转换到流之前，先将成员变量的值改变了，当读取的时候，再将对象的值变回来。通过这种方式，我们可以维持数据的完整性。如果跟我们预期的值不一致的时候，我们就可以在读取的时候抛出异常，说明这个序列化对象可能受到了破坏，下面我们写一个测试程序来验证

```
package Externalizable;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class ExternalizationTest {

    public static void main(String[] args) {

        String fileName = "person.ser";
        Person person = new Person();
        person.setId(1);
        person.setName("Pankaj");
        person.setGender("Male");

        try {
            FileOutputStream fos = new FileOutputStream(fileName);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(person);
            oos.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        FileInputStream fis;
        try {
            fis = new FileInputStream(fileName);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Person p = (Person)ois.readObject();
            ois.close();
            System.out.println("Person Object Read="+p);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }

    }

}
```

运行结果：

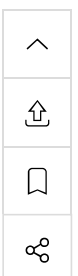
```
Person Object Read=Person{id=1,name=Pankaj,gender=Male}
```

目前，我们掌握了两种序列化对象的方法，分别是java.io.Serializable和java.io.Externalizable，似乎java.io.Externalizable更灵活，可以在写入流之前和读成对象之前分别进行操作。但实际上，我们更推荐使用java.io.Serializable。先卖个小关子，等你读完了这篇文章，就知道为什么了

Java Serialization Methods

我们已经知道java的序列化过程是自动的，只要我实现了Serializable接口，实现这个过程的类有 ObjectInputStream and ObjectOutputStream。但是如果我们有的一些数据比较敏感，比如密码，账户余额，我们不想暴露这些信息，想要save或者retrieving之前先进行一些编码转码工作，那么我们就需要用到序列化的四个方法了，这些方法，可以帮我们改变序列化的行为。

- **readObject(ObjectInputStream ois):** If this method is present in the class, ObjectInputStream readObject() method will use this method for reading the object from stream.



- **writeObject(ObjectOutputStream oos)**: If this method is present in the class, ObjectOutputStream writeObject() method will use this method for writing the object to stream. One of the common usage is to obscure the object variables to maintain data integrity.
- **Object writeReplace()**: If this method is present, then after serialization process this method is called and the object returned is serialized to the stream.
- **Object readResolve()**: If this method is present, then after deserialization process, this method is called to return the final object to the caller program. One of the usage of this method is to implement Singleton pattern with Serialized classes.

通常为了安全性，这些方法的实现都是声明为private，子类无法去重写这些函数。这样做仅仅是为了保证序列化过程的安全性。

Serialization with Inheritance

有些时候，我们可能需要继承一个没有实现Serializable接口的类，如果我们依赖自动的序列化行为，超类有一些成员变量就不会被转换到流中，当我们获取对象的时候，也就无法获取到。

这种情况下，我们就可以用到上面提到的readObject() and writeObject()来解决问题了。通过实现这两个方法，我们可以将超类的状态也序列化，将来获取的时候，就可以获取到了。

```
package Inheritance;

public class SuperClass {

    private int id;
    private String value;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getValue() {
        return value;
    }
    public void setValue(String value) {
        this.value = value;
    }
}
```

SuperClass是一个简单的java bean，但没有实现Serializable接口




```
package Inheritance;

import java.io.IOException;
import java.io.InvalidObjectException;
import java.io.ObjectInputStream;
import java.io.ObjectInputValidation;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class SubClass extends SuperClass implements Serializable, ObjectInputValidation{

    private static final long serialVersionUID = -1322322139926390329L;

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString(){
        return "SubClass{id="+getId()+",value="+getValue()+",name="+getName()+"}";
    }

    //adding helper method for serialization to save/initialize super class state
    private void readObject(ObjectInputStream ois) throws ClassNotFoundException, IOException{
        ois.defaultReadObject();

        //notice the order of read and write should be same
        setId(ois.readInt());
        setValue((String) ois.readObject());
    }

    private void writeObject(ObjectOutputStream oos) throws IOException{
        oos.defaultWriteObject();

        oos.writeInt(getId());
        oos.writeObject(getValue());
    }

    @Override
    public void validateObject() throws InvalidObjectException {
        //validate the object here
        if(name == null || "".equals(name)) throw new InvalidObjectException("name can't be null");
        if(getId() <=0) throw new InvalidObjectException("ID can't be negative or zero");
    }
}
```

我们实现了一个ObjectInputValidation接口，并且实现了 validateObject() 方法，我们可以对数据进行一些转换，以保证数据的完整性不发生变化

我们写一个测试类，来看看我们是否能从流中获取超类的状态信息



```
package Inheritance;

import java.io.IOException;

public class InheritanceSerializationTest {

    public static void main(String[] args) {
        String fileName = "subclass.ser";

        SubClass subClass = new SubClass();
        subClass.setId(10);
        subClass.setValue("Data");
        subClass.setName("Pankaj");

        try {
            SerializationUtil.serialize(subClass, fileName);
        } catch (IOException e) {
            e.printStackTrace();
            return;
        }

        try {
            SubClass subNew = (SubClass) SerializationUtil.deserialize(fileName);
            System.out.println("SubClass read = "+subNew);
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }
    }
}
```

运行结果：

```
SubClass read = SubClass{id=10,value=Data,name=Pankaj}
```

我们发现通过这种方式，我们可以实现序列化超类的信息，即使超类没有实现序列化的接口。这个技术在我们需要序列化一些第三方库的时候，很有用。

Serialization Proxy Pattern

序列化技术存在下面这些问题：

- 类的结构不能改变太多，除非我们重新进行序列化。所以，有时候即使我们已经不需要最初定义的那些变量了，我们仍然需要保留他们。
- 同时，序列化也会带来很多的安全性的问题。一个黑客可能可以通过改变输出的流序列从而来攻击系统。例如，一个用户角色被序列化之后，黑客将它的流改变成admin角色，这样就会造成很大的危险。

为了解决上面的安全性问题，java序列化使用代理模式来增强安全性。在这个模式中，一个内部私有类被用作代理类。这个类持有主类的状态。这个模式主要通过readResolve() and writeReplace() 这两个方法实现的

我们先看代码再来分析



```
import java.io.InvalidObjectException;
import java.io.ObjectInputStream;
import java.io.Serializable;

public class Data implements Serializable{

    private static final long serialVersionUID = 2087368867376448459L;

    private String data;

    public Data(String d){
        this.data=d;
    }

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }

    @Override
    public String toString(){
        return "Data{data="+data+"}";
    }

    //serialization proxy class
    private static class DataProxy implements Serializable{

        private static final long serialVersionUID = 8333905273185436744L;

        private String dataProxy;
        private static final String PREFIX = "ABC";
        private static final String SUFFIX = "DEFG";

        public DataProxy(Data d){
            //obscuring data for security
            this.dataProxy = PREFIX + d.data + SUFFIX;
        }

        private Object readResolve() throws InvalidObjectException {
            if(dataProxy.startsWith(PREFIX) && dataProxy.endsWith(SUFFIX)){
                return new Data(dataProxy.substring(3, dataProxy.length() -4));
            }else throw new InvalidObjectException("data corrupted");
        }

    }

    //replacing serialized object to DataProxy object
    private Object writeReplace(){
        return new DataProxy(this);
    }

    private void readObject(ObjectInputStream ois) throws InvalidObjectException{
        throw new InvalidObjectException("Proxy is not used, something fishy");
    }
}
```

- data 和dataProxy类都必须实现Serializable接口
- DataProxy类需要持有序列化对象的状态
- DataProxy是一个内部的私有类，所以其他类都不可以访问这个类
- DataProxy只有一个构造函数，就是将data类作为参数，也就是被代理对象需要传给dataProxy
- data类需要覆盖writeReplace() 这个方法会在写入流之后调用，这个方法返回dataproxy对象，所以当data类被序列化之后，返回的流是dataproxy类。由于dataproxy类在外部是无法访问的，所以它不可以被直接使用。
- dataproxy类需要实现readResolve方法，并且返回data对象。所以当有一个data类被反序列化的时候，内部的dataproxy会被反序列化，之后就会调用readResolve方法，返回data对象给我们



- 最后实现readObject方法，并且抛出异常，当攻击者想要改变data对象的流的时候，就会被这个方法抛出异常。

我们写一个测试类，来测试这个代理模式是否可用

```
import java.io.IOException;

import SerializationUtil;

public class SerializationProxyTest {

    public static void main(String[] args) {
        String fileName = "data.ser";

        Data data = new Data("Pankaj");

        try {
            SerializationUtil.serialize(data, fileName);
        } catch (IOException e) {
            e.printStackTrace();
        }

        try {
            Data newData = (Data) SerializationUtil.deserialize(fileName);
            System.out.println(newData);
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }
    }
}
```

运行结果

```
Data{data=Pankaj}
```

如果你打开data.ser文件，你会发现dataproxy对象被存在文件中。

Java (/nb/10012174)


© 著作权归作者所有



六尺帐篷 (/u/f8e9b1c246f1)

写了 245418 字，被 15240 人关注，获得了 1429 个喜欢 (/u/f8e9b1c246f1)

喜欢 1



更多分享

(<http://cwb.assets.jianshu.io/notes/images/1502456>)

被以下专题收入，发现更多相似内容


投稿管理


- + 收入我的专题
- 


Android知识 (/c/3fde3b545a35?utm_source=desktop&utm_medium=notes-included-collection)
- 


Android... (/c/5139d555c94d?utm_source=desktop&utm_medium=notes-included-collection)
- 

Android开发 (/c/d1591c322c89?utm_source=desktop&utm_medium=notes-included-collection)









-  Android... (/c/58b4c20abf2f?utm_source=desktop&utm_medium=notes-included-collection)
-  程序员 (/c/NEt52a?utm_source=desktop&utm_medium=notes-included-collection)
-  java (/c/4f285596f858?utm_source=desktop&utm_medium=notes-included-collection)

^

📌

🔖

🔗