008.对象序列化

序列化的含义和意义

对象序列化的目标是将对象保存到磁盘中，或允许在网络中直接传输对象。

对象序列化机制允许把内存中的Java对象转换成平台无关的二进制流，

从而允许把这种二进制流持久地保存在磁盘上，或通过网络将这种二进制流传输到另一个网络节点。

其他程序一旦获得了这种二进制流，都可以将这种二进制流恢复成原来的Java对象。

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。

如果需要让某个对象支持序列化机制，方式有二：

方式一：实现Serializable接口，通过序列化流

实现Serializable接口，通过ObjectOutputStream和ObjectInputStream将对象序列化和反序列化。

Serializable接口是标记接口，是个空接口，用于标识该类可以被序列化。

实例 demo：使用对象流实现序列化

在Java中，只要一个类实现了 java.io.Serializable 接口，它就可以被序列化（**枚举类直接就可以被序列化**）。

```
// Gender类，表示性别
// 每个枚举类型都会默认继承类java.lang.Enum，而Enum类实现了Serializable接口，所以枚举类型对象都是默认可以被序列化的。
public enum Gender {
    MALE, FEMALE
}

// Person 类实现了 Serializable 接口，它包含三个字段。另外，它还重写了该类的 toString() 方法，以方便打印 Person 实例中的内容。
public class Person implements Serializable {
    private String name = null;
    private Integer age = null;
    private Gender gender = null;

    public Person() {
        System.out.println("none-arg constructor");
    }

    public Person(String name, Integer age, Gender gender) {
        System.out.println("arg constructor");
        this.name = name;
        this.age = age;
        this.gender = gender;
    }

    // 省略 set get 方法

    @Override
    public String toString() {
```

```
        return "[" + name + ", " + age + ", " + gender + "]";
    }
}

// SimpleSerial，是个简单的序列化程序，它先将Person对象保存到文件person.out中，然后再从该文件中读出被存储的Person对象，并打印该对象。
public class SimpleSerial {
    public static void main(String[] args) throws Exception {
        File file = new File("D:\\person.out");
        ObjectOutputStream oout = new ObjectOutputStream(new FileOutputStream(file));
     // 注意这里使用的是 ObjectOutputStream 对象输出流封装其他的输出流
        Person person = new Person("John", 101, Gender.MALE);
        oout.writeObject(person);
        oout.close();

        ObjectInputStream oin = new ObjectInputStream(new FileInputStream(file));  // 使用对象输入流读取序列化的对象
        Object newPerson = oin.readObject(); // 没有强制转换到Person类型
        oin.close();
        System.out.println(newPerson);
    }
}
```

进一步分析：

当重新读取被保存的Person对象时，并没有调用Person的任何构造器，

看起来就像是直接使用字节将Person对象还原出来的。

当Person对象被保存到person.out文件后，可以在其它地方去读取该文件以还原对象，

但必须**确保该读取程序的 CLASSPATH 中包含有 Person.class**

哪怕在读取Person对象时并没有显示地使用Person类，如上例所示，

否则会抛出 ClassNotFoundException。

简单的来说，Java 对象序列化就是把对象写入到输出流中，用来存储或传输；反序列化就是从输入流中读取对象。

序列化一个对象首先要创造某些OutputStream对象(如FileOutputStream、ByteArrayOutputStream等)，

然后将其封装在一个ObjectOutputStream对象中，在调用writeObject()方法即可序列化一个对象

反序列化的过程需要创造InputStream对象(如FileInputstream、ByteArrayInputStream等)，

然后将其封装在ObjectInputStream中，在调用readObject()即可

注意对象的序列化是基于字节的，不能使用基于字符的流。

对象引用的序列化

当程序序列化一个Teacher对象时，如果该Teacher对象持有一个Person对象的引用，

为了在反序列化时可以正常恢复该Teacher对象，程序会顺带将该Person对象也进行序列化，

所以Person类也必须是可序列化的，否则Teacher类将不可序列化。

当使用Java序列化机制序列化可变对象时一定要注意，

只有第一次调用writeObject()方法来输出对象时才会将对象转换成字节序列，并写入到ObjectOutputStream；

在后面程序中即使该对象的Field发生了改变，再次调用writeObject()方法输出该对象时，改变后的Field也不会被输出。

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。

Transient 关键字只能用于修饰Field，不可修饰Java程序中的其他成分。

方式二：实现Externalizable接口，重写writeExternal和readExternal方法

Externalizable接口继承了Serializable接口，替我们封装了两个方法，一个用于序列化，一个用于反序列化。

这种方式是将属性序列化，注意这种方式transient修饰词将失去作用，也就是说被transient修饰的属性，

只要你在writeExternal方法中序列化了该属性，照样也会得到序列化。

```
public class Person implements Externalizable {
    private String name = null;
    transient private Integer age = null;
    private Gender gender = null;

    public Person() {
        System.out.println("none-arg constructor");
    }

    public Person(String name, Integer age, Gender gender) {
        System.out.println("arg constructor");
        this.name = name;
        this.age = age;
        this.gender = gender;
    }

    private void writeObject(ObjectOutputStream out) throws IOException {
        out.defaultWriteObject();
        out.writeInt(age);
    }

    private void readObject(ObjectInputStream in) throws IOException,
ClassNotFoundException {
        in.defaultReadObject();
        age = in.readInt();
    }
```

```
    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    }
    ...
}
```

Externalizable 继承于 Serializable，当使用该接口时，**序列化的细节需要由程序员去完成。**

如上所示的代码，由于实现的writeExternal()与readExternal()方法未作任何处理，那么该序列化行为将不会保存/读取任何一个字段。这也就是为什么输出结果中所有字段的值均为空。

另外，使用 Externalizable 接口进行序列化时，**读取对象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，这就是为什么在此次序列化过程中Person类的无参构造器会被调用。

由于这个原因，实现 Externalizable 接口的类必须要提供一个无参构造器，且它的访问权限为public。

对上述Person类做进一步的修改，使其能够对name与age字段进行序列化，但忽略 gender 字段：

```
public class Person implements Externalizable {
    private String name = null;
    transient private Integer age = null;
    private Gender gender = null;

    public Person() {
        System.out.println("none-arg constructor");
    }

    public Person(String name, Integer age, Gender gender) {
        System.out.println("arg constructor");
        this.name = name;
        this.age = age;
        this.gender = gender;
    }

    private void writeObject(ObjectOutputStream out) throws IOException {
        out.defaultWriteObject();
        out.writeInt(age);
    }

    private void readObject(ObjectInputStream in) throws IOException,
ClassNotFoundException {
        in.defaultReadObject();
        age = in.readInt();
    }
```

```
    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(name);
        out.writeInt(age);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        name = (String) in.readObject();
        age = in.readInt();
    }
    ...
}
```

readResolve()方法——单例模式的反序列化

当使用Singleton模式时，应该是期望某个类的实例应该是唯一的，但如果该类是可序列化的，那么情况可能略有不同。

其实这个 case 没必要说太多，知道就行，因为哪里就这么巧，一个能序列化的类（实现了Serializable/Externalizable接口的类），就恰恰是单例的呢？

看下面例子，把 Person 类改造为能序列化的类，然后用反序列攻击单例

```
public class SerializationTest {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        Person person = Person.getInstance();
        ObjectOutputStream objectOutputStream = new ObjectOutputStream(new
FileOutputStream("person"));
        objectOutputStream.writeObject(person);

        ObjectInputStream objectInputStream = new ObjectInputStream(new
FileInputStream("person"));
        Person person1 = (Person) objectInputStream.readObject();

        System.out.println(person == person1); // false
    }
}
```

比较两个 person 实例地址，是 false，说明生成了两个对象，违背了单例类的初衷，

那么为了能在序列化过程仍能保持单例的特性，可以在Person类中添加一个readResolve()方法，

在该方法中直接返回Person的单例对象

```
 public Object readResolve() {
    return 单实例对象;
}
```

原理是当从 I/O 流中读取对象时，ObjectInputStream 类里有 readResolve() 方法，该方法会被自动调用，

期间经过种种逻辑，最后会调用到可序列化类里的 readResolve()方法，

这样可以用 readResolve() 中返回的单例对象直接替换在反序列化过程中创建的对象，实现单例特性。

也就是说，无论如何，反序列化都会额外创建对象，只不过使用 readResolve() 方法可以替换之。

Java序列化对象版本号

Java的序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性的。在进行反序列化时，JVM会把传来的字节流中的serialVersionUID与本地相应实体（类）的serialVersionUID进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常。

显式地定义serialVersionUID有两种用途：

1）在某些场合，希望类的不同版本对序列化兼容，因此需要确保类的不同版本具有相同的serialVersionUID；

在某些场合，不希望类的不同版本对序列化兼容，因此需要确保类的不同版本具有不同的serialVersionUID。

2）当你序列化了一个类实例后，希望更改一个字段或添加一个字段，不设置serialVersionUID，

所做的任何更改都将导致无法反序化旧有实例，并在反序列化时抛出一个异常。

如果你添加了serialVersionUID，在反序列旧有实例时，新添加或更改的字段值将设为初始化值

对象为null，基本类型为相应的初始默认值，字段被删除将不设置。

序列化的安全性

服务器端给客户端发送序列化对象数据，序列化二进制格式的数据写在文档中，并且完全可逆。

一抓包就能就看到类是什么样子，以及它包含什么内容。如果对象中有一些数据是敏感的，比如密码字符串等，则要对字段在序列化时，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。

比如可以通过使用 writeObject 和 readObject 实现密码加密和签名管理，Java提供了更好的方式。

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。

实验：

把对象，保存到硬盘，然后篡改一下，反序列化出来，看看可以不可以！

解决上面的问题，使用SealedObject

KeyGenerator对象介绍：

keyGenerator对象位于javax.crypto包下

jdk 1.6 doc介绍:KeyGenerator 此类提供(对称加密算法:AES,DES 等等)密钥生成器的功能

获得keyGenerator：

一般是通过此类的静态方法getInstance()方法获得,

此类的全局变量都为私有变量,因此不讨论

方法：

1. getAlgorithm();获得算法名称
2. getInstance();通过指定算法,亦可指定提供者来构造KeyGenerator对象,有多个重载方法
3. getProvider();返回此算法实现的提供商
4. init(SecureRandom sRandoom);用于初始化KeyGenerator对象,通过随机源的方式
5. init(int size);通过指定生成秘钥的大小,来初始化的方式
6. init(AlgorithmParameterSpec params);通过指定参数集来初始化
7. init(AlgorithmParameterSpec params,SecureRandom sRandoom);通过指定参数集和随机数源的方式生成
8. init(int arg0, SecureRandom arg1);通过指定大小和随机源的方式产生
9. generatorKey();生成秘钥 // 返回SecertKey对象

支持的算法有：

AES

ARCFOUR

Blowfish

DES //*DES算法*为密码体制中的对称密码体制，又被称为美国数据加密标准，是1972年美国IBM公司研制的对称密码体制加密*算法*。

DESede //*DESede*是由*DES*对称加密算法改进后的一种对称加密算法。使用 168 位的密钥对资料进行三次加密的一种机制；

HmacMD5

HmacSHA1,HmacSHA256,HmacSHA384,HmacSHA512

RC2

Cipher为加密和解密提供密码功能。

获得Cipher：

一般是通过此类的静态方法getInstance()方法获得,

Cipher的工作模式设置在init里

public static final int ENCRYPT_MODE 用于将 Cipher 初始化为加密模式的常量。

```
public static final int DECRYPT_MODE   用于将 Cipher 初始化为解密模式的常量。
public static final int WRAP_MODE      用于将 Cipher 初始化为密钥包装模式的常量。
public static final int UNWRAP_MODE    用于将 Cipher 初始化为密钥解包模式的常量。
public static final int PUBLIC_KEY     用于表示要解包的密钥为“公钥”的常量。
public static final int PRIVATE_KEY    用于表示要解包的密钥为“私钥”的常量。
public static final int SECRET_KEY     用于表示要解包的密钥为“秘密密钥”的常量。
```

```java
public class IOTest1 {
    public static void main(String[] args) {
        ObjectOutputStream objout = null;
        ObjectInputStream objin = null;
        ObjectOutputStream objoutKey = null;
        ObjectInputStream objGetKeyin = null;
        try {
//            先序列化，加密
            OutputStream out = new FileOutputStream("d:\\security");
            objout = new ObjectOutputStream(out);
            Person person = new Person();
            person.name = "zhangsan";
            person.age = 18;

            KeyGenerator keyGenerator = KeyGenerator.getInstance("DESede");
            SecretKey encryptKey = keyGenerator.generateKey();
            Cipher cipher = Cipher.getInstance("DESede");
            cipher.init(Cipher.ENCRYPT_MODE,encryptKey);
            SealedObject so = new SealedObject(person,cipher);
            objout.writeObject(so);

            OutputStream outKey = new FileOutputStream("d:\\keyfile");
            objoutKey = new ObjectOutputStream(outKey);
            objoutKey.writeObject(encryptKey);


            //反序列化
            InputStream in = new FileInputStream("d:\\security");
            objin = new ObjectInputStream(in);
            SealedObject sors = (SealedObject) objin.readObject();

            InputStream getKeyin = new FileInputStream("d:\\keyfile");
            objGetKeyin = new ObjectInputStream(getKeyin);
            SecretKey openKey = (SecretKey) objGetKeyin.readObject();

            Person person1 = (Person) sors.getObject(openKey);

            System.out.println(person1);



        } catch (Exception e) {
            e.printStackTrace();
        }finally {
            if(objout!=null){
                try {
                    objout.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
```

```
            if(objin!=null){
                try {
                    objin.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if(objoutKey!=null){
                try {
                    objoutKey.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if(objGetKeyin!=null){
                try {
                    objGetKeyin.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

class Person implements Serializable {
    private static final long serialVersionUID = -9973655236729699 90L;
    public String name;
    public int age;

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

解决上面的问题，使用SignedObject

```
//反序列化，解密
InputStream in = new FileInputStream("d:\\security");
objin = new ObjectInputStream(in);
SignedObject so2 = (SignedObject) objin.readObject();


InputStream getKeyin = new FileInputStream("d:\\publickey");
objGetKeyin = new ObjectInputStream(getKeyin);
PublicKey publicKey2 = (PublicKey) objGetKeyin.readObject();

//用so2上的一个方法verify，判断盒子里的对象有没有没篡改
Signature verifySignature = Signature.getInstance("DSA");
if (so2.verify(publicKey2, verifySignature)) {
    //内容没被篡改
    System.out.println("内容没被篡改");
    Person person1 = (Person) so2.getObject();
    System.out.println(person1);
}else{
```

```
    System.out.println("内容被篡改过！");
}
```

要知道序列化的是什么样儿的对象（成员）

序列化并不保存静态变量

通过序列化操作，可以实现对任何可 Serializable 对象的深度复制（deep copy），
这意味着复制的是整个对象的关系网，而不仅仅是基本对象及其引用

如果父类没有实现Serializable接口，但其子类实现了此接口，那么这个子类是可以序列化的，
但是在反序列化的过程中会调用父类的无参构造函数，所以在其直接父类（注意是直接父类）中必须有一个无参的构造函数。

反序列化后，何时不是同一个对象

只要将对象序列化到单一流中，就可以恢复出与我们写出时一样的对象网，而且只要在同一流中，对象都是同一个。
否则，反序列化后的对象地址和原对象地址不同，只是内容相同

如果将一个对象序列化入某文件，那么之后又对这个对象进行修改，然后再把修改的对象重新写入该文件，
那么修改无效，文件保存的序列化的对象仍然是最原始的。
这是因为，序列化输出过程跟踪了写入流的对象，而试图将同一个对象写入流时，
并不会导致该对象被复制，而只是将一个句柄写入流，该句柄指向流中相同对象的第一个对象出现的位置。
为了避免这种情况，在后续的 writeObject() 之前调用 out.reset() 方法，这个方法的作用是清除流中保存的写入对象的记录。