<http://blog.csdn.net/dreamtdp/article/details/15378329>

**1、序列化是干什么的？**  
       简单说就是为了保存在内存中的各种对象的状态（也就是实例变量，不是方法），并且可以把保存的对象状态再读出来。虽然你可以用你自己的各种各样的方法来保存object states，但是Java给你提供一种应该比你自己好的保存对象状态的机制，那就是序列化。  
  
**2、什么情况下需要序列化**   
    a）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；  
    b）当你想用套接字在网络上传送对象的时候；  
    c）当你想通过RMI传输对象的时候；  
  
**3、当对一个对象实现序列化时，究竟发生了什么？**  
    在没有序列化前，每个保存在堆（Heap）中的对象都有相应的状态（state），即实例变量（instance ariable）比如：

java 代码

1. Foo  myFoo = **new** Foo();
2. myFoo .setWidth(37);
3. myFoo.setHeight(70);

       当 通过下面的代码序列化之后，MyFoo对象中的width和Height实例变量的值（37，70）都被保存到foo.ser文件中，这样以后又可以把它 从文件中读出来，重新在堆中创建原来的对象。当然保存时候不仅仅是保存对象的实例变量的值，JVM还要保存一些小量信息，比如类的类型等以便恢复原来的对 象。

java 代码

1. FileOutputStream fs = **new** FileOutputStream("foo.ser");
2. ObjectOutputStream os = **new** ObjectOutputStream(fs);
3. os.writeObject(myFoo);

**4、实现序列化（保存到一个文件）的步骤**a）Make a FileOutputStream

**java 代码**

1. **FileOutputStream fs = new FileOutputStream("foo.ser");**

       b）Make a ObjectOutputStream

java 代码

1. ObjectOutputStream os =  **new** ObjectOutputStream(fs);

       c）write the object

java 代码

1. os.writeObject(myObject1);
2. os.writeObject(myObject2);
3. os.writeObject(myObject3);

    d) close the ObjectOutputStream

java 代码

1. os.close();

**5、举例说明**

**java 代码**

1. **import java.io.\*;**
3. **public class  Box implements Serializable**
4. **{**
5. **private int width;**
6. **private int height;**
8. **public void setWidth(int width){**
9. **this.width  = width;**
10. **}**
11. **public void setHeight(int height){**
12. **this.height = height;**
13. **}**
15. **public static void main(String[] args){**
16. **Box myBox = new Box();**
17. **myBox.setWidth(**50**);**
18. **myBox.setHeight(**30**);**
20. **try{**
21. **FileOutputStream fs = new FileOutputStream(**"foo.ser"**);**
22. **ObjectOutputStream os =  new ObjectOutputStream(fs);**
23. **os.writeObject(myBox);**
24. **os.close();**
25. **}catch(Exception ex){**
26. **ex.printStackTrace();**
27. **}**
28. **}**
30. **}**

6、相关注意事项  
    a）序列化时，只对对象的状态进行保存，而不管对象的方法；  
    b）当一个父类实现序列化，子类自动实现序列化，不需要显式实现Serializable接口；  
    c）当一个对象的实例变量引用其他对象，序列化该对象时也把引用对象进行序列化；  
    d）并非所有的对象都可以序列化，,至于为什么不可以，有很多原因了,比如：  
        1.安全方面的原因，比如一个对象拥有private，public等field，对于一个要传输的对象，比如写到文件，或者进行rmi传输  等等，在序列化进行传输的过程中，这个对象的private等域是不受保护的。  
       2. 资源分配方面的原因，比如socket，thread类，如果可以序列化，进行传输或者保存，也无法对他们进行重新的资源分  配，而且，也是没有必要这样实现。

这个确实是个问题.当年在学习**[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/dreamtdp/article/details/_blank)**基础的时候,也问过类似的问题.对呀.为何要进行序列化呢.不进行序列化,我的程序不跑的好好的吗?你想要什么结果,我也能给解决不是.我想说确实是这样,如果你的程序与网络无关,那很好你已经可以摒弃它了.

    那下面我来简单分析下为何java需要进行序列化呢.

    首先我们要明白,序列化是做什么作用的.java序列化: 以特定的方式对类实例的瞬时状态进行编码保存的一种操作.(可能不是很精确,咱不是搞学术的,看懂即可).从此定义可以看出,序列化作用的对象是类的实例.对实例进行序列化,就是保存实例当前在内存中的状态.包括实例的每一个属性的值和引用等.

    既然后序列化,便会有反序列化.反序列化的作用便是将序列化后的编码解码成类实例的瞬时状态.申请等同的内存保存该实例.

    从上述定义可以发现,序列化就是为了保存java的类对象的状态的.保存这个状态的作用主要用于不同jvm之间进行类实例间的共享.在ORMaping中的缓存机制,进行缓存同步时,便是常见的java序列化的应用之一.在进行远程方法调用,远程过程调用时,采用序列化对象的传输也是一种应用...当你想从一个jvm中调用另一个jvm的对象时,你就可以考虑使用序列化了.

**简而言之:序列化的作用就是为了不同jvm之间共享实例对象的一种解决方案.由java提供此机制,效率之高,是其他解决方案无法比拟的.自家的东西嘛.**

**理解Java对象序列化**

关于Java序列化的文章早已是汗牛充栋了，本文是对我个人过往学习，理解及应用Java序列化的一个总结。此文内容涉及Java序列化的基本原理，以及多种方法对序列化形式进行定制。在撰写本文时，既参考了Thinking in Java, Effective Java，JavaWorld，developerWorks中的相关文章和其它网络资料，也加入了自己的实践经验与理解，文、码并茂，希望对大家有所帮助。(2012.02.14最后更新)  
  
1. 什么是Java对象序列化  
    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。  
  
2. 简单示例  
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。此处将创建一个可序列化的类Person，本文中的所有示例将围绕着该类或其修改版。  
    Gender类，是一个枚举类型，表示性别

public enum Gender {  
    MALE, FEMALE  
}

如果熟悉Java枚举类型的话，应该知道每个枚举类型都会默认继承类java.lang.Enum，而该类实现了Serializable接口，所以枚举类型对象都是默认可以被序列化的。  
    Person类，实现了Serializable接口，它包含三个字段：name，String类型；age，Integer类型；gender，Gender类型。另外，还重写该类的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;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public Integer getAge() {  
        return age;  
    }  
  
    public void setAge(Integer age) {  
        this.age = age;  
    }  
  
    public Gender getGender() {  
        return gender;  
    }  
  
    public void setGender(Gender gender) {  
        this.gender = gender;  
    }  
  
    @Override  
    public String toString() {  
        return "[" + name + ", " + age + ", " + gender + "]";  
    }  
}

    SimpleSerial，是一个简单的序列化程序，它先将一个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("person.out");  
  
        ObjectOutputStream oout = new ObjectOutputStream(new FileOutputStream(file));  
        Person person = new Person("John", 101, Gender.MALE);  
        oout.writeObject(person);  
        oout.close();  
  
        ObjectInputStream oin = new ObjectInputStream(new FileInputStream(file));  
        Object newPerson = oin.readObject(); // 没有强制转换到Person类型  
        oin.close();  
        System.out.println(newPerson);  
    }  
}

上述程序的输出的结果为：

arg constructor  
[John, 31, MALE]

    此时必须注意的是，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文件中之后，我们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。  
  
3. Serializable的作用  
    为什么一个类实现了Serializable接口，它就可以被序列化呢？在上节的示例中，使用ObjectOutputStream来持久化对象，在该类中有如下代码：

private void writeObject0(Object obj, boolean unshared) throws IOException {  
    IMG_256  
    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());  
        }  
    }  
    IMG_257  
}

从上述代码可知，如果被写对象的类型是String，或数组，或Enum，或Serializable，那么就可以对该对象进行序列化，否则将抛出NotSerializableException。  
  
4. 默认序列化机制  
    如果仅仅只是让某个类实现Serializable接口，而没有其它任何处理的话，则就是使用默认序列化机制。使用默认机制，在序列化对象时，不仅会序列化当前对象本身，还会对该对象引用的其它对象也进行序列化，同样地，这些其它对象引用的另外对象也将被序列化，以此类推。所以，如果一个对象包含的成员变量是容器类对象，而这些容器所含有的元素也是容器类对象，那么这个序列化的过程就会较复杂，开销也较大。  
  
5. 影响序列化  
    在现实应用中，有些时候不能使用默认序列化机制。比如，希望在序列化过程中忽略掉敏感数据，或者简化序列化过程。下面将介绍若干影响序列化的方法。  
  
5.1 transient关键字  
    当某个字段被声明为transient后，默认序列化机制就会忽略该字段。此处将Person类中的age字段声明为transient，如下所示，

public class Person implements Serializable {  
    IMG_258  
    transient private Integer age = null;  
    IMG_259  
}

再执行SimpleSerial应用程序，会有如下输出：

arg constructor  
[John, null, MALE]

可见，age字段未被序列化。  
  
5.2 writeObject()方法与readObject()方法  
    对于上述已被声明为transitive的字段age，除了将transitive关键字去掉之外，是否还有其它方法能使它再次可被序列化？方法之一就是在Person类中添加两个方法：writeObject()与readObject()，如下所示：

public class Person implements Serializable {  
    IMG_260  
    transient private Integer age = null;  
    IMG_261  
  
    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();  
    }  
}

在writeObject()方法中会先调用ObjectOutputStream中的defaultWriteObject()方法，该方法会执行默认的序列化机制，如5.1节所述，此时会忽略掉age字段。然后再调用writeInt()方法显示地将age字段写入到ObjectOutputStream中。readObject()的作用则是针对对象的读取，其原理与writeObject()方法相同。  
    再次执行SimpleSerial应用程序，则又会有如下输出：

arg constructor  
[John, 31, MALE]

必须注意地是，writeObject()与readObject()都是private方法，那么它们是如何被调用的呢？毫无疑问，是使用反射。详情可见ObjectOutputStream中的writeSerialData方法，以及ObjectInputStream中的readSerialData方法。  
  
5.3 Externalizable接口  
    无论是使用transient关键字，还是使用writeObject()和readObject()方法，其实都是基于Serializable接口的序列化。JDK中提供了另一个序列化接口--Externalizable，使用该接口之后，之前基于Serializable接口的序列化机制就将失效。此时将Person类修改成如下，

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 {  
  
    }  
    IMG_262  
}

此时再执行SimpleSerial程序之后会得到如下结果：

arg constructor  
none-arg constructor  
[null, null, null]

从该结果，一方面可以看出Person对象中任何一个字段都没有被序列化。另一方面，如果细心的话，还可以发现这此次序列化过程调用了Person类的无参构造器。  
    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，若使用Externalizable进行序列化，当读取对象时，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。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的构造器，且它的访问权限为public。  
    对上述Person类作进一步的修改，使其能够对name与age字段进行序列化，但要忽略掉gender字段，如下代码所示：

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();  
    }  
    IMG_263  
}

执行SimpleSerial之后会有如下结果：

arg constructor  
none-arg constructor  
[John, 31, null]

5.4 readResolve()方法  
    当我们使用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，那么情况可能会略有不同。此时对第2节使用的Person类进行修改，使其实现Singleton模式，如下所示：

public class Person implements Serializable {  
  
    private static class InstanceHolder {  
        private static final Person instatnce = new Person("John", 31, Gender.MALE);  
    }  
  
    public static Person getInstance() {  
        return InstanceHolder.instatnce;  
    }  
  
    private String name = null;  
  
    private Integer age = null;  
  
    private Gender gender = null;  
  
    private Person() {  
        System.out.println("none-arg constructor");  
    }  
  
    private Person(String name, Integer age, Gender gender) {  
        System.out.println("arg constructor");  
        this.name = name;  
        this.age = age;  
        this.gender = gender;  
    }  
    IMG_264  
}

同时要修改SimpleSerial应用，使得能够保存/获取上述单例对象，并进行对象相等性比较，如下代码所示：

public class SimpleSerial {  
  
    public static void main(String[] args) throws Exception {  
        File file = new File("person.out");  
        ObjectOutputStream oout = new ObjectOutputStream(new FileOutputStream(file));  
        oout.writeObject(Person.getInstance()); // 保存单例对象  
        oout.close();  
  
        ObjectInputStream oin = new ObjectInputStream(new FileInputStream(file));  
        Object newPerson = oin.readObject();  
        oin.close();  
        System.out.println(newPerson);  
  
        System.out.println(Person.getInstance() == newPerson); // 将获取的对象与Person类中的单例对象进行相等性比较  
    }  
}

执行上述应用程序后会得到如下结果：

arg constructor  
[John, 31, MALE]  
false

值得注意的是，从文件person.out中获取的Person对象与Person类中的单例对象并不相等。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，如下所示：

public class Person implements Serializable {  
  
    private static class InstanceHolder {  
        private static final Person instatnce = new Person("John", 31, Gender.MALE);  
    }  
  
    public static Person getInstance() {  
        return InstanceHolder.instatnce;  
    }  
  
    private String name = null;  
  
    private Integer age = null;  
  
    private Gender gender = null;  
  
    private Person() {  
        System.out.println("none-arg constructor");  
    }  
  
    private Person(String name, Integer age, Gender gender) {  
        System.out.println("arg constructor");  
        this.name = name;  
        this.age = age;  
        this.gender = gender;  
    }  
  
    private Object readResolve() throws ObjectStreamException {  
        return InstanceHolder.instatnce;  
    }  
    IMG_265  
}

再次执行本节的SimpleSerial应用后将有如下输出：

arg constructor  
[John, 31, MALE]  
true

    无论是实现Serializable接口，或是Externalizable接口，当从I/O流中读取对象时，readResolve()方法都会被调用到。实际上就是用readResolve()中返回的对象直接替换在反序列化过程中创建的对象，而被创建的对象则会被垃圾回收掉。