**序列化和反序列化**

## 一、序列化和反序列化的概念

**把对象转换为字节序列的过程称为对象的序列化。**

**把字节序列恢复为对象的过程称为对象的反序列化。**

对象（object）是类（class）的一个实例（instance）。一个类中包含了变量（field）和函数（method）两个部分。同一个 类的不同对象只是变量不同，所以Java API序列化过程中只保存了对象的变量部分。同样，由于静态变量（static field）是由同一个类的各个对象共用的，所以序列化过程中也不保存。 由于还原对象时需要在程序中动态创建该对象，所以程序也需要知道该对象的类定义，所以如果对象由一个程序序列化保存之后，由另外一个程序反序列化还原 时，类文件也需要传送给该程序。这就需要扩展Java API序列化的功能，对其进行自定义。Java的远程方法调用(Remote Method Invocation, RMI)功能，就是以Java API序列化为基础，并进行了扩展

**1 对象的序列化主要有两种用途**

　　1） 把对象的字节序列永久地保存到硬盘上，通常存放在一个文件中；

2） 在网络上传送对象的字节序列。

### **2 序列化的用途**

**（1）对象持久化（persistence）**

　　对象持久化是指延长对象的存在时间。通常状况下，当程序结束时，程序中的对象不再存在。如果通过序列化功能，将对象保存到文件中，就可以延长对象的存在时间，在下次程序运行是再恢复该对象。序列化将对象保存在文件中，是实现对象持久化的一种方式。持久化还有很多种方式，比如Hibernate框架就提供了一整套对象持久化的方案。

**（2）对象复制**

　　通过序列化，将对象保存在内存中，可以再通过此数据得到多个对象的副本。

**（3）对象传输**

　　通过序列化，将对象转化字节流后，可以通过网络发送给另外的Java程序。

**3 在很多应用中，需要对某些对象进行序列化，让它们离开内存空间，入住物理硬盘，以便长期保存。**比如最常见的是Web服务器中的Session对象，当有 10万用户并发访问，就有可能出现10万个Session对象，内存可能吃不消，于是Web容器就会把一些seesion先序列化到硬盘中，等要用了，再把保存在硬盘中的对象还原到内存中。

**4 当两个进程在进行远程通信时**，彼此可以发送各种类型的数据。无论是何种类型的数据，都会以二进制序列的形式在网络上传送。发送方需要把这个Java对象转换为字节序列，才能在网络上传送；接收方则需要把字节序列再恢复为Java对象。

## 二、JDK类库中的序列化API

**1 java.io.ObjectOutputStream代表对象输出流**，它的writeObject(Object obj)方法可对参数指定的obj对象进行序列化，把得到的字节序列写到一个目标输出流中。

**2 java.io.ObjectInputStream代表对象输入流**，它的readObject()方法从一个源输入流中读取字节序列，再把它们反序列化为一个对象，并将其返回。

**3 只有实现了Serializable和Externalizable**接口的类的对象才能被序列化。Externalizable接口继承自 Serializable接口，实现Externalizable接口的类完全由自身来控制序列化的行为，而仅实现Serializable接口的类可以 采用默认的序列化方式 。

**4 对象序列化包括如下步骤：**

　　1） 创建一个对象输出流，它可以包装一个其他类型的目标输出流，如文件输出流；

　　2） 通过对象输出流的writeObject()方法写对象。

**5 对象反序列化的步骤如下：**

　　1） 创建一个对象输入流，它可以包装一个其他类型的源输入流，如文件输入流；

　　2） 通过对象输入流的readObject()方法读取对象。

**6 对象序列化和反序列范例：定义一个Person类，实现Serializable接口**

1 import java.io.Serializable;

2

3 /\*\*

4 \* <p>ClassName: Person<p>

5 \* <p>Description:测试对象序列化和反序列化<p>

6 \* @author xudp

7 \* @version 1.0 V

8 \* @createTime 2014-6-9 下午02:33:25

9 \*/

10 public class Person implements Serializable {

11

12 /\*\*

13 \* 序列化ID

14 \*/

15 private static final long serialVersionUID = -5809782578272943999L;

16 private int age;

17 private String name;

18 private String sex;

19

20 public int getAge() {

21 return age;

22 }

23

24 public String getName() {

25 return name;

26 }

27

28 public String getSex() {

29 return sex;

30 }

31

32 public void setAge(int age) {

33 this.age = age;

34 }

35

36 public void setName(String name) {

37 this.name = name;

38 }

39

40 public void setSex(String sex) {

41 this.sex = sex;

42 }

43 }

**7 序列化和反序列化Person类对象**

1 import java.io.File;

2 import java.io.FileInputStream;

3 import java.io.FileNotFoundException;

4 import java.io.FileOutputStream;

5 import java.io.IOException;

6 import java.io.ObjectInputStream;

7 import java.io.ObjectOutputStream;

8 import java.text.MessageFormat;

9

10 /\*\*

11 \* <p>ClassName: TestObjSerializeAndDeserialize<p>

12 \* <p>Description: 测试对象的序列化和反序列<p>

13 \* @author xudp

14 \* @version 1.0 V

15 \* @createTime 2014-6-9 下午03:17:25

16 \*/

17 public class TestObjSerializeAndDeserialize {

18

19 public static void main(String[] args) throws Exception {

20 SerializePerson();//序列化Person对象

21 Person p = DeserializePerson();//反序列Perons对象

22 System.out.println(MessageFormat.format("name={0},age={1},sex={2}",

23 p.getName(), p.getAge(), p.getSex()));

24 }

25

26 /\*\*

27 \* MethodName: SerializePerson

28 \* Description: 序列化Person对象

29 \* @author xudp

30 \* @throws FileNotFoundException

31 \* @throws IOException

32 \*/

33 private static void SerializePerson() throws FileNotFoundException,

34 IOException {

35 Person person = new Person();

36 person.setName("gacl");

37 person.setAge(25);

38 person.setSex("男");

39 // ObjectOutputStream 对象输出流，将Person对象存储到E盘的Person.txt文件中，完成对Person对象的序列化操作

40 ObjectOutputStream oo = new ObjectOutputStream(new FileOutputStream(

41 new File("E:/Person.txt")));

42 oo.writeObject(person);

43 System.out.println("Person对象序列化成功！");

44 oo.close();

45 }

46

47 /\*\*

48 \* MethodName: DeserializePerson

49 \* Description: 反序列Perons对象

50 \* @author xudp

51 \* @return

52 \* @throws Exception

53 \* @throws IOException

54 \*/

55 private static Person DeserializePerson() throws Exception, IOException {

56 ObjectInputStream ois = new ObjectInputStream(new FileInputStream(

57 new File("E:/Person.txt")));

58 Person person = (Person) ois.readObject();

59 System.out.println("Person对象反序列化成功！");

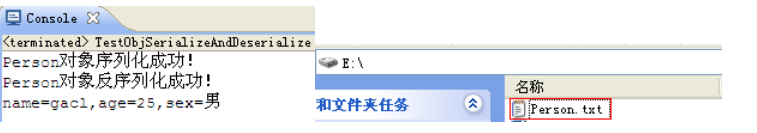
60 return person;

61 }

62

63 }

**8 代码运行结果如下：**



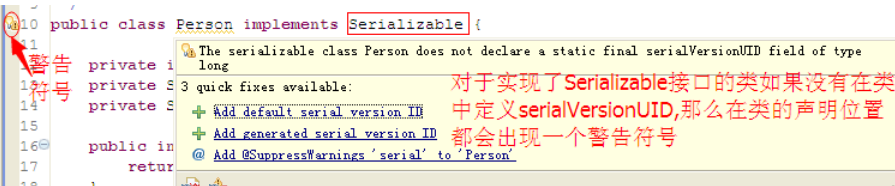
序列化Person成功后在E盘生成了一个Person.txt文件，而反序列化Person是读取E盘的Person.txt后生成了一个Person对象

## 三、serialVersionUID的作用

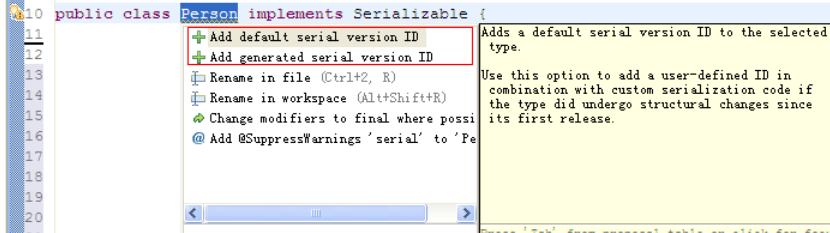
**1字​面​意​思​上​是​序​列​化​的​版​本​号​**

凡是实现Serializable接口的类都有一个表示序列化版本标识符的静态变量。

2 实现Serializable接口的类如果类中没有添加serialVersionUID，那么就会出现如下的警告提示



3 用鼠标点击就会弹出生成serialVersionUID的对话框，如下图所示：



**4 serialVersionUID有两种生成方式**

1. 

这种方式生成的serialVersionUID是1L，例如：

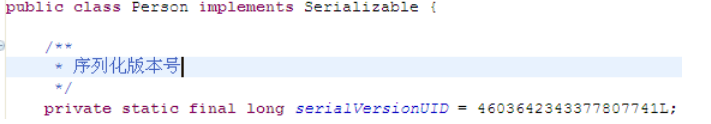
private static final long serialVersionUID = 1L;

1. 

这种方式生成的serialVersionUID是根据类名，接口名，方法和属性等来生成的，例如：

private static final long serialVersionUID = 4603642343377807741L;

**5 添加了之后就不会出现那个警告提示了，如下所示：**



**6 扯了那么多，那么serialVersionUID(序列化版本号)到底有什么用呢**

**（1）**例子来说明一下serialVersionUID的作用，看下面的代码：

1 import java.io.File;

2 import java.io.FileInputStream;

3 import java.io.FileNotFoundException;

4 import java.io.FileOutputStream;

5 import java.io.IOException;

6 import java.io.ObjectInputStream;

7 import java.io.ObjectOutputStream;

8 import java.io.Serializable;

9

10 public class TestSerialversionUID {

11

12 public static void main(String[] args) throws Exception {

13 SerializeCustomer();// 序列化Customer对象

14 Customer customer = DeserializeCustomer();// 反序列Customer对象

15 System.out.println(customer);

16 }

17

18 /\*\*

19 \* MethodName: SerializeCustomer

20 \* Description: 序列化Customer对象

21 \* @author xudp

22 \* @throws FileNotFoundException

23 \* @throws IOException

24 \*/

25 private static void SerializeCustomer() throws FileNotFoundException,

26 IOException {

27 Customer customer = new Customer("gacl",25);

28 // ObjectOutputStream 对象输出流

29 ObjectOutputStream oo = new ObjectOutputStream(new FileOutputStream(

30 new File("E:/Customer.txt")));

31 oo.writeObject(customer);

32 System.out.println("Customer对象序列化成功！");

33 oo.close();

34 }

35

36 /\*\*

37 \* MethodName: DeserializeCustomer

38 \* Description: 反序列Customer对象

39 \* @author xudp

40 \* @return

41 \* @throws Exception

42 \* @throws IOException

43 \*/

44 private static Customer DeserializeCustomer() throws Exception, IOException {

45 ObjectInputStream ois = new ObjectInputStream(new FileInputStream(

46 new File("E:/Customer.txt")));

47 Customer customer = (Customer) ois.readObject();

48 System.out.println("Customer对象反序列化成功！");

49 return customer;

50 }

51 }

52

53 /\*\*

54 \* <p>ClassName: Customer<p>

55 \* <p>Description: Customer实现了Serializable接口，可以被序列化<p>

56 \* @author xudp

57 \* @version 1.0 V

58 \* @createTime 2014-6-9 下午04:20:17

59 \*/

60 class Customer implements Serializable {

61 //Customer类中没有定义serialVersionUID

62 private String name;

63 private int age;

64

65 public Customer(String name, int age) {

66 this.name = name;

67 this.age = age;

68 }

69

70 /\*

71 \* @MethodName toString

72 \* @Description 重写Object类的toString()方法

73 \* @author xudp

74 \* @return string

75 \* @see java.lang.Object#toString()

76 \*/

77 @Override

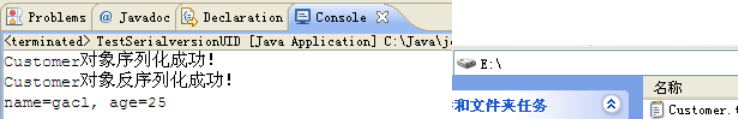
78 public String toString() {

79 return "name=" + name + ", age=" + age;

80 }

81 }

（2）运行结果：



序列化和反序列化都成功了。

（3）下面我们修改一下Customer类，添加多一个sex属性，如下：

1 class Customer implements Serializable {

2 //Customer类中没有定义serialVersionUID

3 private String name;

4 private int age;

5

6 //新添加的sex属性

7 private String sex;

8

9 public Customer(String name, int age) {

10 this.name = name;

11 this.age = age;

12 }

13

14 public Customer(String name, int age,String sex) {

15 this.name = name;

16 this.age = age;

17 this.sex = sex;

18 }

19

20 /\*

21 \* @MethodName toString

22 \* @Description 重写Object类的toString()方法

23 \* @author xudp

24 \* @return string

25 \* @see java.lang.Object#toString()

26 \*/

27 @Override

28 public String toString() {

29 return "name=" + name + ", age=" + age;

30 }

31 }

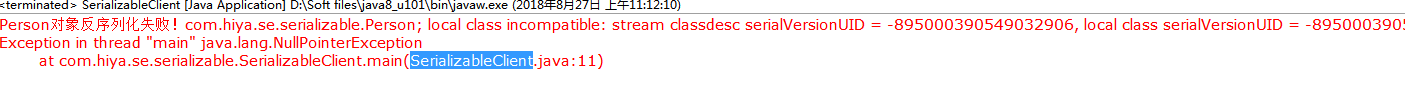
（4）然后执行反序列操作，此时就会抛出如下的异常信息：

1 Exception in thread "main" java.io.InvalidClassException: Customer;

2 local class incompatible:

3 stream classdesc serialVersionUID = -88175599799432325,

4 local class serialVersionUID = -5182532647273106745



（5）意思就是说，文件流中的class和classpath中的class，也就是修改过后的class，不兼容了，处于安全机制考虑，程序抛出了错误，并且拒绝载入。那么如果我们真的有需求要在序列化后添加一个字段或者方法呢？应该怎么办？那就是自己去指定serialVersionUID。在TestSerialversionUID例子中，没有指定Customer类的serialVersionUID的，那么java编译器会自动给这个class进行一个摘要算法，类似于指纹算法，只要这个文件 多一个空格，得到的UID就会截然不同的，可以保证在这么多类中，这个编号是唯一的。所以，添加了一个字段后，由于没有显指定 serialVersionUID，编译器又为我们生成了一个UID，当然和前面保存在文件中的那个不会一样了，于是就出现了2个序列化版本号不一致的错误。**因此，只要我们自己指定了serialVersionUID，就可以在序列化后，去添加一个字段，或者方法，而不会影响到后期的还原，还原后的对象照样可以使用，而且还多了方法或者属性可以用。**

（6）下面继续修改Customer类，给Customer指定一个serialVersionUID，修改后的代码如下

1 class Customer implements Serializable {

2 /\*\*

3 \* Customer类中定义的serialVersionUID(序列化版本号)

4 \*/

5 private static final long serialVersionUID = -5182532647273106745L;

6 private String name;

7 private int age;

8

9 //新添加的sex属性

10 //private String sex;

11

12 public Customer(String name, int age) {

13 this.name = name;

14 this.age = age;

15 }

16

17 /\*public Customer(String name, int age,String sex) {

18 this.name = name;

19 this.age = age;

20 this.sex = sex;

21 }\*/

22

23 /\*

24 \* @MethodName toString

25 \* @Description 重写Object类的toString()方法

26 \* @author xudp

27 \* @return string

28 \* @see java.lang.Object#toString()

29 \*/

30 @Override

31 public String toString() {

32 return "name=" + name + ", age=" + age;

33 }

34 }

（7）重新执行序列化操作，将Customer对象序列化到本地硬盘的Customer.txt文件存储，然后修改Customer类，添加sex属性，修改后的Customer类代码如下：

1 class Customer implements Serializable {

2 /\*\*

3 \* Customer类中定义的serialVersionUID(序列化版本号)

4 \*/

5 private static final long serialVersionUID = -5182532647273106745L;

6 private String name;

7 private int age;

8

9 //新添加的sex属性

10 private String sex;

11

12 public Customer(String name, int age) {

13 this.name = name;

14 this.age = age;

15 }

16

17 public Customer(String name, int age,String sex) {

18 this.name = name;

19 this.age = age;

20 this.sex = sex;

21 }

22

23 /\*

24 \* @MethodName toString

25 \* @Description 重写Object类的toString()方法

26 \* @author xudp

27 \* @return string

28 \* @see java.lang.Object#toString()

29 \*/

30 @Override

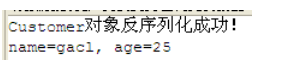
31 public String toString() {

32 return "name=" + name + ", age=" + age;

33 }

34 }

（8）执行反序列操作，这次就可以反序列成功了，如下所示：



## 四、serialVersionUID的取值

**1 serialVersionUID的取值是Java运行时环境根据类的内部细节自动生成的**。如果对类的源代码作了修改，再重新编译，新生成的类文件的serialVersionUID的取值有可能也会发生变化。

**2 类的serialVersionUID的默认值完全依赖于Java编译器的实现**，对于同一个类，用不同的Java编译器编译，有可能会导致不同的 serialVersionUID，也有可能相同。

3 **为了提高serialVersionUID的独立性和确定性，强烈建议在一个可序列化类中显示的定义serialVersionUID，为它赋予明确的值。**

**4 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；

（2） 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。