Java序列化机制和原理

Java序列化算法透析

Serialization（序列化）是一种将对象以一连串的字节描述的过程；反序列化deserialization是一种将这些字节重建成一个对象的过程。Java序列化API提供一种处理对象序列化的标准机制。在这里你能学到如何序列化一个对象，什么时候需要序列化以及Java序列化的算法，我们用一个实例来示范序列化以后的字节是如何描述一个对象的信息的。

序列化的必要性

Java中，一切都是对象，在分布式环境中经常需要将Object从这一端网络或设备传递到另一端。

这就需要有一种可以在两端传输数据的协议。Java序列化机制就是为了解决这个问题而产生。

如何序列化一个对象

一个对象能够序列化的前提是实现Serializable接口，Serializable接口没有方法，更像是个标记。

有了这个标记的Class就能被序列化机制处理。

```
import java.io.Serializable;

class TestSerial implements Serializable {

    public byte version = 100;

    public byte count = 0;

}
```

然后我们写个程序将对象序列化并输出。ObjectOutputStream能把Object输出成Byte流。

我们将Byte流暂时存储到temp.out文件里。

```
public static void main(String args[]) throws IOException {
    FileOutputStream fos = new FileOutputStream("temp.out");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    TestSerial ts = new TestSerial();
    oos.writeObject(ts);
    oos.flush();
    oos.close();
}
```

如果要从持久的文件中读取Bytes重建对象，我们可以使用ObjectInputStream。

```
public static void main(String args[]) throws IOException {
    FileInputStream fis = new FileInputStream("temp.out");
    ObjectInputStream oin = new ObjectInputStream(fis);
    TestSerial ts = (TestSerial) oin.readObject();
    System.out.println("version="+ts.version);
}
```

执行结果为 100.

对象的序列化格式

将一个对象序列化后是什么样子呢？打开刚才我们将对象序列化输出的temp.out文件

以16进制方式显示。内容应该如下：

```
AC ED 00 05 73 72 00 0A 53 65 72 69 61 6C 54 65
73 74 A0 0C 34 00 FE B1 DD F9 02 00 02 42 00 05
63 6F 75 6E 74 42 00 07 76 65 72 73 69 6F 6E 78
70 00 64
```

这一坨字节就是用来描述序列化以后的TestSerial对象的，我们注意到TestSerial类中只有两个域：

public byte version = 100;

public byte count = 0;

且都是byte型，理论上存储这两个域只需要2个byte，但是实际上temp.out占据空间为51bytes，也就是说除了数据以外，还包括了对序列化对象的其他描述

Java的序列化算法

序列化算法一般会按步骤做如下事情：

♦将对象实例相关的类元数据输出。

♦递归地输出类的超类描述直到不再有超类。

♦类元数据完了以后，开始从最顶层的超类开始输出对象实例的实际数据值。

♦从上至下递归输出实例的数据

我们用另一个更完整覆盖所有可能出现的情况的例子来说明：

```
class parent implements Serializable {
    int parentVersion = 10;
}
```

```
class contain implements Serializable{
    Int containVersion = 11;
}
```

```
public class SerialTest extends parent implements Serializable {
       int version = 66;
       contain con = new contain();
       public int getVersion() {
              return version;
       }
       public static void main(String args[]) throws IOException {
              FileOutputStream fos = new FileOutputStream("temp.out");
              ObjectOutputStream oos = new ObjectOutputStream(fos);
              SerialTest st = new SerialTest();
              oos.writeObject(st);
              oos.flush();
```

```
            oos.close();
        }
}
```

这个例子是相当的直白啦。SerialTest类实现了Parent超类，内部还持有一个Container对象。

序列化后的格式如下：

AC ED 00 05 73 72 00 0A 53 65 72 69 61 6C 54 65

73 74 05 52 81 5A AC 66 02 F6 02 00 02 49 00 07

76 65 72 73 69 6F 6E 4C 00 03 63 6F 6E 74 00 09

4C 63 6F 6E 74 61 69 6E 3B 78 72 00 06 70 61 72

65 6E 74 0E DB D2 BD 85 EE 63 7A 02 00 01 49 00

0D 70 61 72 65 6E 74 56 65 72 73 69 6F 6E 78 70

00 00 00 0A 00 00 00 42 73 72 00 07 63 6F 6E 74

61 69 6E FC BB E6 0E FB CB 60 C7 02 00 01 49 00

0E 63 6F 6E 74 61 69 6E 56 65 72 73 69 6F 6E 78

70 00 00 00 0B

我们来仔细看看这些字节都代表了啥。开头部分，见颜色：

序列化算法的第一步就是输出对象相关类的描述。例子所示对象为SerialTest类实例，

因此接下来输出SerialTest类的描述。见颜色：

接下来，算法输出其中的一个域，int version=66；见颜色：

然后，算法输出下一个域，contain con = new contain();这个有点特殊，是个对象。

描述对象类型引用时需要使用JVM的标准对象签名表示法，见颜色：

.接下来算法就会输出超类也就是Parent类描述了，见颜色：

下一步，输出parent类的域描述，int parentVersion=100;同见颜色：

到此为止，算法已经对所有的类的描述都做了输出。下一步就是把实例对象的实际值输出了。这时候是从parent Class的域开始的，见颜色：

还有SerialTest类的域：

再往后的bytes比较有意思，算法需要描述contain类的信息，要记住，

现在还没有对contain类进行过描述，见颜色：

.输出contain的唯一的域描述，int containVersion=11；

这时，序列化算法会检查contain是否有超类，如果有的话会接着输出。

最后，将contain类实际域值输出。

OK,我们讨论了java序列化的机制和原理，希望能对同学们有所帮助。

- AC ED: STREAM_MAGIC. 声明使用了序列化协议.
- 00 05: STREAM_VERSION. 序列化协议版本.
- 0x73: TC_OBJECT. 声明这是一个新的对象.
- 0x72: TC_CLASSDESC. 声明这里开始一个新Class。
- 00 0A: Class名字的长度.
- 53 65 72 69 61 6c 54 65 73 74: SerialTest,Class类名.
- 05 52 81 5A AC 66 02 F6: SerialVersionUID, 序列化ID，如果没有指定，

  则会由算法随机生成一个8byte的ID.
- 0x02: 标记号. 该值声明该对象支持序列化。
- 00 02: 该类所包含的域个数。
- 0x49: 域类型. 49 代表"I", 也就是Int.
- 00 07: 域名字的长度.
- 76 65 72 73 69 6F 6E: version,域名字描述.
- 0x4C: 域的类型.
- 00 03: 域名字长度.
- 63 6F 6E: 域名字描述，con
- 0x74: TC_STRING. 代表一个new String.用String来引用对象。
- 00 09: 该String长度.
- 4C 63 6F 6E 74 61 69 6E 3B: Lcontain;, JVM的标准对象签名表示法.
- 0x78: TC_ENDBLOCKDATA,对象数据块结束的标志
- 0x72: TC_CLASSDESC. 声明这个是个新类.
- 00 06: 类名长度.
- 70 61 72 65 6E 74: parent,类名描述。
- 0E DB D2 BD 85 EE 63 7A: SerialVersionUID, 序列化ID.
- 0x02: 标记号. 该值声明该对象支持序列化.

- 00 01: 类中域的个数.
- 0x49: 域类型. 49 代表"I", 也就是Int.
- 00 0D: 域名字长度.
- 70 61 72 65 6E 74 56 65 72 73 69 6F 6E: parentVersion，域名字描述。
- 0x78: TC_ENDBLOCKDATA,对象块结束的标志。
- 0x70: TC_NULL, 说明没有其他超类的标志。.
- 00 00 00 0A: 10, parentVersion域的值.
- 00 00 00 42: 66, version域的值.
- 0x73: TC_OBJECT, 声明这是一个新的对象.
- 0x72: TC_CLASSDESC声明这里开始一个新Class.
- 00 07: 类名的长度.
- 63 6F 6E 74 61 69 6E: contain,类名描述.
- FC BB E6 0E FB CB 60 C7: SerialVersionUID, 序列化ID.
- 0x02: Various flags. 标记号. 该值声明该对象支持序列化
- 00 01: 类内的域个数。
- 0x49: 域类型. 49 代表"I", 也就是Int..
- 00 0E: 域名字长度.
- 63 6F 6E 74 61 69 6E 56 65 72 73 69 6F 6E: containVersion, 域名字描述.
- 0x78: TC_ENDBLOCKDATA对象块结束的标志.
- 0x70:TC_NULL，没有超类了。
- 00 00 00 0B: 11, containVersion的值.

转自 http://www.java3z.com/cwbwebhome/article/article8/862.html

SerialVersionUID值的重要作用

根据上面的分析,可以发现如果一个类可序列化,serialVersionUID建议给一个确定的值,不要由系统自动生成,否则在增减字段(不能修改字段类型及长度)时,如果两边的类的版本不同会导致反序列化失败.

注意问题

如果序列化时代码这样写:

```
SerialTest st = new SerialTest();

oos.writeObject((parent)st);
```

会发现序列化的对象依然是SerialTest,如果在分布式环境中用Parent反序列化(调用段不存在SerialTest),会造成ClassNotFoundException.

serialVersionUID适用于Java的序列化机制。简单来说，Java的序列化机制是通过判断类的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，即是InvalidCastException。

serialVersionUID有两种显示的生成方式：
一是默认的1L，比如：private static final long serialVersionUID = 1L;
二是根据类名、接口名、成员方法及属性等来生成一个64位的哈希字段，比如：
private static final long serialVersionUID = xxxxL;

当一个类实现了Serializable接口，如果没有显示的定义serialVersionUID，Eclipse会提供相应的提醒。面对这种情况，我们只需要在Eclipse中点击类中warning图标一下，Eclipse就会　　自动给定两种生成的方式。如果不想定义，在Eclipse的设置中也可以把它关掉的，设置如下：
Window ==> Preferences ==> Java ==> Compiler ==> Error/Warnings ==> Potential programming problems
将Serializable class without serialVersionUID的warning改成ignore即可。

当实现java.io.Serializable接口的类没有显式地定义一个serialVersionUID变量时候，Java序列化机制会根据编译的Class自动生成一个serialVersionUID作序列化版本比较用，这种情况下，如果Class文件(类名，方法明等)没有发生变化(增加空格，换行，增加注释等等)，就算再编译多次，serialVersionUID也不会变化的。

如果我们不希望通过编译来强制划分软件版本，即实现序列化接口的实体能够兼容先前版本，就需要显式地定义一个名为serialVersionUID，类型为long的变量，不修改这个变量值的序列化实体都可以相互进行串行化和反串行化。

下面用代码说明一下serialVersionUID在应用中常见的几种情况。

（1）序列化实体类

```
1  import java.io.Serializable;
2  public class Person implements Serializable
3  {
4      private static final long serialVersionUID = 1234567890L;
5      public int id;
6      public String name;
7
8      public Person(int id, String name)
9      {
10         this.id = id;
11         this.name = name;
12     }
13
14     public String toString()
15     {
16         return "Person: " + id + " " + name;
17     }
18 }
```

（2）序列化功能：

```
1  import java.io.FileOutputStream;
2  import java.io.IOException;
```

```
3  import java.io.ObjectOutputStream;
4
5  public class SerialTest
6  {
7
8      public static void main(String[] args) throws IOException
9      {
10         Person person = new Person(1234, "wang");
11         System.out.println("Person Serial" + person);
12         FileOutputStream fos = new FileOutputStream("Person.txt");
13         ObjectOutputStream oos = new ObjectOutputStream(fos);
14         oos.writeObject(person);
15         oos.flush();
16         oos.close();
17     }
18 }
```

（3）反序列化功能：

```
1  import java.io.FileInputStream;
2  import java.io.IOException;
3  import java.io.ObjectInputStream;
4  public class DeserialTest
5  {
6      public static void main(String[] args) throws IOException, ClassNotFo
7      {
8          Person person;
9
10         FileInputStream fis = new FileInputStream("Person.txt");
11         ObjectInputStream ois = new ObjectInputStream(fis);
12         person = (Person) ois.readObject();
13         ois.close();
14         System.out.println("Person Deserial" + person);
15     }
16
17 }
```

情况一：假设Person类序列化之后，从A端传输到B端，然后在B端进行反序列化。在序列化Person和反序列化Person的时候，A端和B端都需要存在一个相同的类。如果两处的serialVersionUID不一致，会产生什么错误呢?

【答案】可以利用上面的代码做个试验来验证：

先执行测试类SerialTest，生成序列化文件，代表A端序列化后的文件，然后修改serialVersion值，再执行测试类DeserialTest，代表B端使用不同serialVersion的类去反序列化，结果报错:

Exception in thread "main" java.io.InvalidClassException: test.Person; local class incompatible: stream classdesc serialVersionUID = 1234567890, local class serialVersionUID = 123456789

at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:560)

at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1580)

at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1493)

at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1729)

at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1326)

```
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:348)
    at test.DeserialTest.main(DeserialTest.java:15)
```

情况二：假设两处serialVersionUID一致，如果A端增加一个字段，然后序列化，而B端不变，然后反序列化，会是什么情况呢?
【答案】新增 public int age; 执行SerialTest，生成序列化文件，代表A端。删除 public int age，反序列化，代表B端，最后的结果为：执行序列化，反序列化正常，但是A端增加的字段丢失(被B端忽略)。

情况三：假设两处serialVersionUID一致，如果B端减少一个字段，A端不变，会是什么情况呢?
【答案】序列化，反序列化正常，B端字段少于A端，A端多的字段值丢失(被B端忽略)。

情况四：假设两处serialVersionUID一致，如果B端增加一个字段，A端不变，会是什么情况呢?
验证过程如下：
先执行SerialTest，然后在实体类Person增加一个字段age，如下所示，再执行测试类DeserialTest.

```java
import java.io.Serializable;
public class Person implements Serializable
{
    private static final long serialVersionUID = 123456789L;
    public int id;
    public String name;
    public int age;

    public Person(int id, String name)
    {
        this.id = id;
        this.name = name;
    }

    public String toString()
    {
        return "Person: " + id + " " + name;
    }
}
```

相应的修改测试类DeserialTest，打印出age的值。

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

public class SerialTest
{

    public static void main(String[] args) throws IOException
    {
        Person person = new Person(1234, "wang");
        System.out.println("Person Serial" + person + " age:" + person.ag
        FileOutputStream fos = new FileOutputStream("Person.txt");
```

```
13          ObjectOutputStream oos = new ObjectOutputStream(fos);
14          oos.writeObject(person);
15          oos.flush();
16          oos.close();
17      }
18  }
```

结果为：
Person Deserial Person: 1234 wang age: 0
说明序列化，反序列化正常，B端新增加的int字段被赋予了默认值0。
最后通过下面的图片，总结一下上面的几种情况。