Java API 中 java.io.Serializable 接口源码：

```
1 public abstract interface Serializable {
2 }
```

类通过实现 java.io.Serializable 接口可以启用其序列化功能。未实现次接口的类无法使其任何状态序列化或反序列化。可序列化类的所有子类型本身都是可序列化的。序列化接口没有方法或字段，仅用于标识可序列化的语义。

Java 的"对象序列化"能让你将一个实现了 Serializable 接口的对象转换成 byte 流，这样日后要用这个对象时候，你就能把这些 byte 数据恢复出来，并据此重新构建那个对象了。

要想序列化对象，你必须先创建一个 OutputStream，然后把它嵌进 ObjectOutputStream。这时，你就能用 writeObject()方法把对象写入 OutputStream 了。

writeObject()方法负责写入特定类的对象的状态，以便相应的 readObject()方法可以还原它。通过调用 out.defaultWriteObject 可以调用保存 Object 的字段的默认机制。该方法本身不需要涉及属于其超类或子类的状态。状态是通过使用 writeObject 方法或使用 DataOutput 支持的用于基本数据类型的方法将各个字段写入 ObjectOutputStream 来保存的。

读的时候，你得把 InputStream 嵌到 ObjectInputStream 里面，然后再调用 readObject()方法。不过这样读出来的，只是一个 Object 的 reference，因此在用之前，还得先下传。readObject() 方法负责从流中读取并还原类字段。它可以调用 in.defaultReadObject 来调用默认机制，以还原对象的非静态和非瞬态字段。defaultReadObject()方法使用流中的信息来分配流中通过当前对象中相应命名字段保存的对象的字段。这用于处理类发展后需要添加新字段的情形。该方法本身不需要涉及属于其超类或子类的状态。状态是通过使用 writeObject 方法或使用 DataOutput 支持的用于基本数据类型的方法将各个字段写入 ObjectOutputStream 来保存的。

在序列化时，有几点要注意的：

1：当一个对象被序列化时，只保存对象的非静态成员变量（包括声明为 private 的变量），不能保存任何的成员方法和静态的成员变量。

2：如果一个对象的成员变量是一个对象，那么这个对象的数据成员也会被序列化。

3：如果一个可序列化的对象包含对某个不可序列化的对象的引用，那么整个序列化操作将会失败，并且会抛出一个 NotSerializableException。我们可以将这个引用标记为 transient，那么对象仍然可以序列化。

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，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：

```
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。

```
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

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

b）Make a ObjectOutputStream

java 代码

```
ObjectOutputStream os = new ObjectOutputStream(fs);
```

c）write the object

java 代码

```
os.writeObject(myObject1);
os.writeObject(myObject2);
os.writeObject(myObject3);
```

d) close the ObjectOutputStream

java 代码

```
os.close();
```

5、举例说明

```
1 public class Box implements Serializable {
2     private static final long serialVersionUID =
-3450064362986273896L;
3
4     private int width;
5     private int height;
6
7     public static void main(String[] args) {
8         Box myBox=new Box();
9         myBox.setWidth(50);
10        myBox.setHeight(30);
11        try {
12            FileOutputStream fs=new
FileOutputStream("F:\\foo.ser");
13            ObjectOutputStream os=new ObjectOutputStream(fs);
14            os.writeObject(myBox);
15            os.close();
16            FileInputStream fi=new FileInputStream("F:\\foo.ser");
17            ObjectInputStream oi=new ObjectInputStream(fi);
18            Box box=(Box)oi.readObject();
19            oi.close();
20            System.out.println(box.height+","+box.width);
21        } catch (Exception e) {
22            e.printStackTrace();
23        }
```

```
24      }
25
26      public int getWidth() {
27          return width;
28      }
29      public void setWidth(int width) {
30          this.width = width;
31      }
32      public int getHeight() {
33          return height;
34      }
35      public void setHeight(int height) {
36          this.height = height;
37      }
38  }
```

6、相关注意事项

a）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；

b）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；

c）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。

serialVersionUID

序列化运行时使用一个称为 serialVersionUID 的版本号与每个可序列化类相关联，该序列号在反序列化过程中用于验证序列化对象的发送者和接收者是否为该对象加载了与序列化兼容的类。如果接收者加载的该对象的类的 serialVersionUID 与对应的发送者的类的版本号不同，则反序列化将会导致 InvalidClassException。可序列化类可以通过声明名为 "serialVersionUID" 的字段（该字段必须是静态 (static)、最终 (final) 的 long 型字段）显式声明其自己的 serialVersionUID：

```
ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;
```

如果可序列化类未显式声明 serialVersionUID，则序列化运行时将基于该类的各个方面计算该类的默认 serialVersionUID 值，如“Java(TM) 对象序列化规范”中所述。不过，强烈建议 所有可序列化类都显式声明 serialVersionUID 值，原因是计算默认的 serialVersionUID 对类的详细信息具有较高的敏感性，根据编译器实现的不同可能千差万别，这样在反序列化过程中可能会导致意外的 InvalidClassException。因此，为保证 serialVersionUID 值跨不同 java 编译器实现的一致性，序列化类必须声明一个明确的 serialVersionUID 值。还强烈建议使用 private 修饰符显示声明 serialVersionUID（如果可能），原因是这种声明仅应用于直接声明类 -- serialVersionUID 字段作为继承成员没有用处。数组类不能声明一个明确的 serialVersionUID，因此它们总是具有默认的计算值，但是数组类没有匹配 serialVersionUID 值的要求。