# Chapter 10 序列化

当你创建对象时,只要你需要,它就会一直存在,但是在程序终止时,无论如何它都不会继续存在.尽管这样做肯定是有意义的,但是仍然存在某种情况,如果对象能够在程序不运行的情况下仍能存在并保存信息,那将非常有用.这样,在下次运行程序时,该对象将被重建并且拥有的信息与在程序上次运行时它所拥有的信息相同。当然，我们可以通过将信息写入文件或数据库来达到相同的效果，但是在使万物都成为对象的精神中，如果能够将一个对象声明为是“持久的”，并为我们处理掉所有的细节，那将会显得十分方便，并且也符合面向对象的设计原则，即封装实现的细节。

如果想要将对象进行持久化或网络传输需要先将对象转换为二进制流。这种将对象转变为二进制流的过程称为对象的序列化（Serialization）。反之，将二进制流恢复为数据对象的过程称之为反序列化（Deserialization）。序列化需要保留充分的信息以恢复数据对象，但是为了节省存储空间和网络带宽，序列化后的二进制流又要尽可能小。序列化常见的使用场景是RPC（Remote Procedure Call—[远程过程调用](https://baike.baidu.com/item/%E8%BF%9C%E7%A8%8B%E8%BF%87%E7%A8%8B%E8%B0%83%E7%94%A8/7854346" \t "https://baike.baidu.com/item/_blank)，它是一种通过[网络](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C/143243" \t "https://baike.baidu.com/item/_blank)从远程计算机程序上请求服务，而不需要了解底层网络技术的协议）框架的数据传输。常见的序列化方式有三种,Java原生序列化、Hession序列化以及JSON序列化,下面我们就来一一了解下,并分析三者的优劣点,使得在以后的开发中有的放矢,最出最好的选择。

## 10.1 Java原生序列化

Java的对象序列化将那些实现了Serializable接口的对象转换成一个字节序列，并能够在以后将这个字节序列完全恢复为原来的对象。这一过程甚至不局限于地域，可以通过网络进行；这意味着序列化机制能自动弥补不同操作系统之间的差异。也就是说，可以在运行Unix系统的计算机上创建一个对象，将其序列化，通过网络将它发给一台运行Windows系统的计算机，然后在那里准确的重新组装，而却不必担心数据在不同机器上的表示会不同，也不必关心字节的顺序或者其他任何细节。

就其本身来说，对象的序列化是非常有趣的，因为利用它实现轻量级持久性。“持久性”意味着一个对象的生存周期并不取决于程序是否正在执行；它可以生存于程序的调用之间。通过将一个序列化对象写入磁盘，然后在重新调用程序时恢复该对象，就能够实现持久性的效果。之所以称其为“轻量级”，是因为不能用某种“persistence”关键字来简单定义一个对象，并让系统自动维护其他实现细节问题（但并不意味未来没有这种可能性）。相反，对象必须在程序中显式的序列化和反序列化还原。如果需要一个更严格的持久性机制，可以考虑其它如Hibernate之类的工具。

对象序列化的概念加入到语言中是为了支持两种主要特性。一是Java的远程方法调用（Remote Method Invocation，RMI），它使存活在其它计算机上的对象使用起来就像是存活在本机上一样。当向远程对象发送信息时，需要通过对象序列化来传输参数和返回值。关于RMI，将会在后面详细探讨；

再者，对Java Beans来说，对象的序列化也是必须的。使用一个Bean时，一般情况下是在设计阶段对它的状态信息进行配置。这种状态信息必须保存下来，并在程序启动时进行后期恢复；这种具体工作就是由对象序列化来完成的。

只要对象实现了Serializable接口(这仅仅只是一个标记接口,同Cloneable相同,不包括任何方法,只是标识它可以具有某个功能),对象的序列化处理将会非常简单.当序列化的概念被加入到语言中时,许多标准库类都发生了改变,以便具备序列化特性----其中包括所有基本数据类型的封装器、所有容器类以及许多其他的东西，甚至Class对象也可以被序列化。

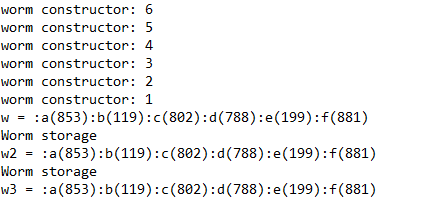
Java序列化保留了对象类的元数据（如类、成员变量、继承类信息等），以及对象数据等，兼容性最好，但却并不支持跨语言，而且性能十分一般。

### 10.1.1 Java序列化基础与细节

要序列化一个对象，首先要创建某些OutputStream对象，然后将其封装在一个ObjectOutputStream对象内。这时，只需调用writeObject（）即可将对象序列化，并将其发送给OutputStream（对象序列化是基于字节的，因要使用InputStream和OutputStream继承层次结构）。要反向进行该过程（即将一个序列还原为一个对象），需要将一个InputStream封装在ObjectInputStream内,然后调用readObject()。和往常一样，我们最后获得的是一个引用(新的引用,不同于序列化前的引用,是一个全新地址的引用)，它指向一个向上转型的Object，所以必须向下转型才能直接设置它们。

对象序列化特别聪明的一个的地方是它不仅保存了对象的“全景图”，而且能追踪对象内所包含的所有引用，并保存那些对象；接着又能对对象内包含的每个这样的引用进行追踪；依次类推。这种情况有时候被称为“对象网”，单个对象可与之建立连接，而且它还包含了对象的引用数组以及成员对象。如果必须保持一套自己的对象序列化机制，那么维护那些可追踪到所有链接的代码可能显得非常麻烦。然而，由于Java的对象序列化似乎找不出什么缺点，所以请尽量不要自己动手，让它用优化的算法自动维护整个对象网。下面这个例子通过对链接的对象生成一个worm对序列化机制进行了测试。每个对象都与worm中的下一段链接，同时又与属于不同类（Data）的对象引用数组链接:

|  |
| --- |
| **package** cn.yorick.serialization.practice;  **import** java.io.Serializable;  **public** **class** Data **implements** Serializable {  **private** **int** n;  **public** Data(**int** n) {  **super**();  **this**.n = n;  }  @Override  **public** String toString() {  // **TODO** Auto-generated method stub  **return** Integer.*toString*(n);  }  } |
| package cn.yorick.serialization.practice;  import java.util.Random;  public class Worm implements Serializable {  private static Random random = new Random(47);  private Data[] d = {new Data(random.nextInt(10)),new Data(random.nextInt(10)),new Data(random.nextInt(10))};  private Worm next;  private char c;  public Worm(int i, char x) {  super();  System.out.println("worm constructor: " + i);  c = x;  if(--i > 0) {  next = new Worm(i,(char) (x + 1));  }  }  public Worm() {  System.out.println("default constructor!");  }  @Override  public String toString() {  StringBuilder result = new StringBuilder(":");//构造一个初始化为指定字符串内容的字符串构建器。  result.append(c);  result.append("(");  for (Data data : d) {  result.append(data);  }  result.append(")");  if(next != null) {  result.append(next);  }  return result.toString();  }  public static void main(String[] args) throws FileNotFoundException, IOException, ClassNotFoundException {  Worm w = new Worm(6, 'a'); //实际创建了6个对象,所以构造器中进行了6次输出打印  System.out.println("w = " + w);  ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("worm.out"));  oos.writeObject("Worm storage\n"); //写入一个字符串对象到数据输出流  oos.writeObject(w);//写入一个对象到数据输出流  oos.close();  ObjectInputStream ois = new ObjectInputStream(new FileInputStream("worm.out"));//对象读入流  String s = (String) ois.readObject();  Worm w2 = (Worm) ois.readObject();  ***//执行该打印时,会调用toString方法,首先构造一个包含字符串:的构造器,然后添加字符等等,然后循环添加Date,此时结果是:a(853)最后判断next,此时next为new worm(5,(char)(a+1)),循环打印next,知道next为null,所以会出现如此打印的效果***  ***6 next = new worm(5,b)--->next = new worm(4,c)--->next = new worm(3,d)----> next = new worm(2,e)---> next = new worm(1,f)***  System.out.println(s + "w2 = " + w2);  ByteArrayOutputStream bout = new ByteArrayOutputStream();  ObjectOutputStream out2 = new ObjectOutputStream(bout);  out2.writeObject("Worm storage\n");  out2.writeObject(w);  out2.flush();  ObjectInputStream ois2 = new ObjectInputStream(new ByteArrayInputStream(bout.toByteArray()));  s = (String) ois2.readObject();  Worm w3 = (Worm) ois2.readObject();  System.out.println(s + "w3 = " + w3);  }  } |



Worm内的Date对象数组是用随机数初始化的(阴影1,这样就可以不用怀疑编译器保留了某种原始信息)。每个Worm段都用一个char加以标记。该char是在递归生成链接的Worm列表时自动生成的（阴影2）。要创建一个Worm，必须告诉构造器你所希望的它的长度。在产生下一个引用时，要调用Worm构造器，并将长度减1，以此类推。最后一个next引用则为null（空），表示已到达Worm的尾部。【注意这个测试用例是极易出现栈溢出异常的，当我们初始化i的值过大时就会出现Stack OverflowException,需要注意】

序列化使得应用程序以便携式方式将原始Java数据类型写入数据输出流,然后应用程序可以从数据输入流中来读取数据;

上面这段代码看起来很复杂,这只是代码设计的问题,真正的序列化是十分简单的,如同我们看到的那样,实现接口,然后将对象数据写入流,再从流中读取。可以利用序列化将对象读写到任何实现DataInputStream或者DataOutputStream的ObjectInputStream和ObjectOutputStream，甚至包括网络。

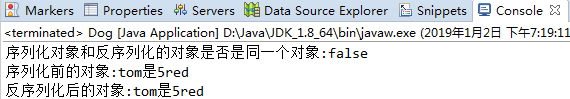
在这里我们要注意一点，对一个Serializable对象进行还原的过程中，没有调用任何构造器，包括默认的构造器，整个对象都是通过从InputStream中取得数据恢复而来的。

当对象被序列化时，该对象引用的对象都被序列化，这是由Java的序列化机制自动完成的（观察发现上面Data数据就被序列化了）。

#### 10.1.1.1 对象网序列化注意点

前面我们知道当对象被序列化时,该对象引用的对象都被序列化,且是由Java的序列化机制自动完成的,但这并不意味着是绝对的。前提是对象版图上的所有对象都应该实现序列化接口,我们测试如果其中有一个类没有实现序列化接口,那么会抛出运行时异常java.io.NotSerializableException,我们假设Collar类没有序列化,那么测试结果如下

|  |
| --- |
| **public** **class** Collar **implements** Serializable{  **private** **int** size;  **private** String color;  **public** Collar(**int** size, String color) {  **super**();  **this**.size = size;  **this**.color = color;  }  **public** **int** getSize() {  **return** size;  }  **public** **void** setSize(**int** size) {  **this**.size = size;  }  **public** String getColor() {  **return** color;  }  **public** **void** setColor(String color) {  **this**.color = color;  }  @Override  **public** String toString() {  **return** getSize() + getColor();  }  } |
| **public** **class** Dog **implements** Serializable{  String name ; //狗的名字  Collar collar; //狗的类型  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** Collar getCollar() {  **return** collar;  }  **public** **void** setCollar(Collar collar) {  **this**.collar = collar;  }  @Override  **public** String toString() {  **return** getName() + "是" + getCollar();  }  **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException {  Dog dog = **new** Dog();  dog.setCollar(**new** Collar(5, "red"));  dog.setName("tom");  //序列化dog对象  FileOutputStream fos = **new** FileOutputStream("dog.ser");  ObjectOutputStream oos = **new** ObjectOutputStream(fos);  oos.writeObject(dog);  oos.close();  //反序列化  FileInputStream fis = **new** FileInputStream("dog.ser");  ObjectInputStream ois = **new** ObjectInputStream(fis);  Object object = ois.readObject();  //需要注意读取后都是Object,需要进行强制类型转换,同时也可能出现classNotFoundException  Dog dog1 = (Dog) object;  System.***out***.println("序列化对象和反序列化的对象是否是同一个对象:" + (dog == dog1));  System.***out***.println("序列化前的对象:" + dog.toString());  System.***out***.println("反序列化后的对象:" + dog1.toString());  }  } |



|  |
| --- |
| Exception in thread "main" java.io.NotSerializableException: cn.yorick.serialization.Collar  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184)  at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1548)  at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1509)  at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1432)  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1178)  at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348)  at cn.yorick.serialization.Dog.main(Dog.java:41) |

我们发现根目录下仍然有这个dog.ser文件,这是由于FileOutPutStream创建的,并不是真的成功序列化了。我们试着将代码中序列化代码注释然后反序列化这个文件,发现控制台报出异常,反序列化无法读取这个对象,如下

|  |
| --- |
| Exception in thread "main" java.io.WriteAbortedException: writing aborted; java.io.NotSerializableException: cn.yorick.serialization.Collar  at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1572)  at java.io.ObjectInputStream.defaultReadFields(ObjectInputStream.java:2282)  at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:2206)  at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:2064)  at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1568)  at java.io.ObjectInputStream.readObject(ObjectInputStream.java:428)  at cn.yorick.serialization.Dog.main(Dog.java:52)  Caused by: java.io.NotSerializableException: cn.yorick.serialization.Collar  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184)  at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1548)  at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1509)  at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1432)  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1178)  at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348)  at cn.yorick.serialization.Dog.main(Dog.java:46) |

由此我们可以知道任何要序列化的对象，那么在他的对象版图所有的对象对应该实现序列化接口，否则执行时一定会出问题，序列化和反序列化也不会成功。

|  |
| --- |
| 总结：序列化和反序列化是底层机制自动进行的，而且很强的的是，它会将对象网序列化，但是需要注意的是对象网中每个对象都要实现Serializable接口，否则序列化和反序列化一定不会成功。同时序列化是针对对象的所以类成员（static）是不会被默认序列化机制序列化的。 |

#### 10.1.1.2 寻找类

对序列化有了初步认识后，我们想一下将一个对象从它的序列化状态中恢复出来，有哪些工作是必须做的呢？举个例子来说，假如我们将一个对象序列化，并通过网络将其作为文件传递给另一台计算机；那么，另一台计算机上的程序可以只利用该文件内容来还原这个对象吗？要想知道答案最好的办法就是亲身测试一下，我们下面来试验一下：

|  |
| --- |
| **public** **class** Alien **implements** Serializable {} ///:~ |
| **public** **class** FreezeAlice {  **public** **static** **void** main(String[] args) **throws** FileNotFoundException, IOException {  ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream("X.file"));  cn.yorick.serialization.Alien alien = **new** cn.yorick.serialization.Alien();  oos.writeObject(alien);  oos.close();  }  } |

上面这个方法一旦经过编译和运行那么是会在根目录serialization下生成一个x.file文件的,此时我们将cn.yorick.serialization.Alien删除或者移到更高层级的目录,然后我们通过下面的代码来反序列化x.file

|  |
| --- |
| **public** **class** ThawAlice {  **public** **static** **void** main(String[] args) **throws** FileNotFoundException, IOException, ClassNotFoundException {  ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream("X.file"));  Object alice = ois.readObject();  System.***out***.println(alice.getClass());  }  } |

运行结果是控制台抛出了异常如下:

|  |
| --- |
| Exception in thread "main" java.lang.ClassNotFoundException: cn.yorick.serialization.Alien  at java.net.URLClassLoader.findClass(URLClassLoader.java:381)  at java.lang.ClassLoader.loadClass(ClassLoader.java:424)  at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:335)  at java.lang.ClassLoader.loadClass(ClassLoader.java:357)  at java.lang.Class.forName0(Native Method)  at java.lang.Class.forName(Class.java:348)  at java.io.ObjectInputStream.resolveClass(ObjectInputStream.java:683)  at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1863)  at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1746)  at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:2037)  at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1568)  at java.io.ObjectInputStream.readObject(ObjectInputStream.java:428)  at cn.yorick.serialization.practice.ThawAlice.main(ThawAlice.java:15) |

打开文件和读取alice对象中的内容都是需要Alien的Class对象;而Java虚拟机找不到Alien.class(除非它正好在类路径classpath内，而本例却不在类路径内)，这样就得到了一个ClassNotFoundException的异常（同样，除非能够保证Alien存在，否则它等于消失）。反序列化成功的关键就是必须保证Java虚拟机能找到相关的.class文件。

上述结论子啊对象网中同样适用，对象网中只要有一个类文件找不到，那么整个反序列也都会失败；

|  |
| --- |
| 总结：反序列化不会调用构造器创建对象，它会从文件中读取数据构造对象，但是反序列化成功的关键是该对象对应的类要存在，否则会出现ClassNotFoundException |

#### 10.1.1.3 序列化号

序列化时每个对象都是用一个序列号(serial number)保存的，这就是这种机制之所以称为对象序列化的原因。下面是其算法：

* 对你遇到的每一个对象引用都关联一个序列号。
* 对于每个对象，当第一次遇到时，保存其对象数据到输出流中。
* 如果某个对象之前已经保存过，那么只写出“与之前保存过的序列号为x的对象相同”。

在反序列化读回对象时，整个过程是反过来的。

* 对于对象输入流中的对象，在第一次遇到其序列化号时，构建它，并使用流中数据来初始化它，然后记录这个顺序号和新对象之间的关联。
* 当遇到“与之前保存过的序列号为x的对象相同”标记时，获取与这个顺序号相关联的对象引用。

我们使用序列化将对象集合保存在磁盘文件中，并按照它们被保存的样子获取它们。序列化的另一种非常重要的应用是通过网络将对象集合传送到另一台计算机上。正如在文件中保存原生的内存地址毫无意义一样，这些地址对于在不同的处理器之间的通信也是毫无意义的。因为序列化用序列化号代替了内存地址，所以它允许将对象集合从一台机器传送到另一台机器。

下面我们针对序列化与反序列化来详细看看。下面是一个雇员类和一个经理类，其中

|  |
| --- |
| **public** **class** Employee **implements** Serializable{  **private** String name;  **private** **double** salary;  **private** LocalDate hireDay;  **public** Employee() {}  **public** Employee(String n, **double** s, **int** year, **int** month, **int** day) {  name = n;  salary = s;  hireDay = LocalDate.*of*(year, month, day);  }  **public** String getName() {  **return** name;  }  **public** **double** getSalary() {  **return** salary;  }  **public** LocalDate getHireDay() {  **return** hireDay;  }  **public** **void** raiseSalary(**double** byPercent) {  **double** raise = salary \* byPercent / 100;  salary += raise;  }  **public** String toString() {  **return** getClass().getName()  + "[name=" + name  + ",salary=" + salary  + ",hireDay=" + hireDay  + "]";  }  } |
| **public** **class** Manager **extends** Employee{  **private** Employee secretary;//秘书  **public** Manager(String n, **double** s, **int** year, **int** month, **int** day){  **super**(n, s, year, month, day);  secretary = **null**;  }  **public** **void** setSecretary(Employee s) {  secretary = s;  }  **public** String toString() {  **return** **super**.toString() + "[secretary=" + secretary + "]";  }  } |
| **class** ObjectStreamTest{  **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException {  Employee harry = **new** Employee("Harry Hacker", 50000, 1989, 10, 1);  Manager carl = **new** Manager("Carl Cracker", 80000, 1987, 12, 15);  carl.setSecretary(harry);  Manager tony = **new** Manager("Tony Tester", 40000, 1990, 3, 15);  tony.setSecretary(harry);  Employee[] staff = **new** Employee[3];  staff[0] = carl;  staff[1] = harry;  staff[2] = tony;  **try** (ObjectOutputStream out = **new** ObjectOutputStream(**new** FileOutputStream("employee.dat"))) {  out.writeObject(staff);  }  **try** (ObjectInputStream in = **new** ObjectInputStream(**new** FileInputStream("employee.dat"))) {  Employee[] newStaff = (Employee[]) in.readObject();  newStaff[1].raiseSalary(10);  **for** (Employee e : newStaff)  System.***out***.println(e);  }  }  } |
| cn.yorick.serialization.Manager[name=Carl Cracker,salary=80000.0,hireDay=1987-12-15][secretary=cn.yorick.serialization.Employee[name=Harry Hacker,salary=55000.0,hireDay=1989-10-01]]  cn.yorick.serialization.Employee[name=Harry Hacker,salary=55000.0,hireDay=1989-10-01]  cn.yorick.serialization.Manager[name=Tony Tester,salary=40000.0,hireDay=1990-03-15][secretary=cn.yorick.serialization.Employee[name=Harry Hacker,salary=55000.0,hireDay=1989-10-01]] |

我们发现我们只是设置了 newStaff[1].raiseSalary(10);,但是所有harry出现的地方都改变了,我们可以推断出,的确如上所述当对象序列化时对于每个对象，当第一次遇到时，保存其对象数据到输出流中。如果某个对象之前已经保存过，那么只写出“与之前保存过的序列号为x的对象相同”。在反序列化读回对象时，整个过程是反过来的。当遇到“与之前保存过的序列号为x的对象相同”标记时，获取与这个顺序号相关联的对象引用，而不是重新构建一个新的对象，我们可以通过另外一个例子来说明：

|  |
| --- |
| **public** **class** House **implements** Serializable {} |
| **public** **class** Animal **implements** Serializable {  **private** String name;  **private** House preferredHouse;  **public** Animal(String nm, House h) {  **super**();  **this**.name = nm;  **this**.preferredHouse = h;  }  @Override  **public** String toString() {  // **TODO** Auto-generated method stub  **return** name + "[" + **super**.toString() + "], " + preferredHouse + "\n";  }  } |
| **public** **class** MyWorld {  **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException {  House house = **new** House();  List<Animal> animals = **new** ArrayList<>();  animals.add(**new** Animal("Bosco the dog", house));  animals.add(**new** Animal("Ralph the hamster", house));  animals.add(**new** Animal("Molly the cat", house));  System.***out***.println("animals: " + animals);  ByteArrayOutputStream bos1 = **new** ByteArrayOutputStream();  ObjectOutputStream oos1 = **new** ObjectOutputStream(bos1);  oos1.writeObject(animals);  oos1.writeObject(animals);  System.***out***.println(bos1.size());  oos1.writeObject(animals);  System.***out***.println(bos1.size());  animals.add(**new** Animal("Molly the cats", house));  // System.out.println(animals);这里animals已经发生了改变  oos1.writeObject(animals);  System.***out***.println(bos1.size());  ByteArrayOutputStream bos2 = **new** ByteArrayOutputStream();  ObjectOutputStream oos2 = **new** ObjectOutputStream(bos2);  oos2.writeObject(animals);  ObjectInputStream ois1 = **new** ObjectInputStream(**new** ByteArrayInputStream(bos1.toByteArray()));  //我们在这里改变了对象,并且重新序列化,反序列化后发现对象是变化前的对象,这是由于序列化机制,序列化时会  //判断序列化的对象是否为同一个对象,判断依据是内存地址,如果相同那么只会序列化一次,  //这也就是为什么我们重复序列化同一个对象,重复读取后是一样的,因为序列化后的文件中只有一份  ObjectInputStream ois2 = **new** ObjectInputStream(**new** ByteArrayInputStream(bos2.toByteArray()));  List<Animal> animals1 = (List<Animal>) ois1.readObject();  List<Animal> animals2 = (List<Animal>) ois1.readObject();  List<Animal> animals4 = (List<Animal>) ois1.readObject();  List<Animal> animals3 = (List<Animal>) ois2.readObject();  List<Animal> animals5 = (List<Animal>) ois1.readObject();  System.***out***.println("animals1: " + animals1);  System.***out***.println("animals2: " + animals2);  System.***out***.println("animals3: " + animals3);  System.***out***.println("animals4: " + animals4);  System.***out***.println("animals5: " + animals5);  }  } |
| animals: [Bosco the dog[cn.yorick.serialization.Animal@7852e922], cn.yorick.serialization.House@4e25154f  , Ralph the hamster[cn.yorick.serialization.Animal@70dea4e], cn.yorick.serialization.House@4e25154f  , Molly the cat[cn.yorick.serialization.Animal@5c647e05], cn.yorick.serialization.House@4e25154f  ]  309  314  319  animals1: [Bosco the dog[cn.yorick.serialization.Animal@2d98a335], cn.yorick.serialization.House@16b98e56  , Ralph the hamster[cn.yorick.serialization.Animal@7ef20235], cn.yorick.serialization.House@16b98e56  , Molly the cat[cn.yorick.serialization.Animal@27d6c5e0], cn.yorick.serialization.House@16b98e56  ]  animals2: [Bosco the dog[cn.yorick.serialization.Animal@2d98a335], cn.yorick.serialization.House@16b98e56  , Ralph the hamster[cn.yorick.serialization.Animal@7ef20235], cn.yorick.serialization.House@16b98e56  , Molly the cat[cn.yorick.serialization.Animal@27d6c5e0], cn.yorick.serialization.House@16b98e56  ]  animals3: [Bosco the dog[cn.yorick.serialization.Animal@4f3f5b24], cn.yorick.serialization.House@15aeb7ab  , Ralph the hamster[cn.yorick.serialization.Animal@7b23ec81], cn.yorick.serialization.House@15aeb7ab  , Molly the cat[cn.yorick.serialization.Animal@6acbcfc0], cn.yorick.serialization.House@15aeb7ab  , Molly the cats[cn.yorick.serialization.Animal@5f184fc6], cn.yorick.serialization.House@15aeb7ab  ]  animals4: [Bosco the dog[cn.yorick.serialization.Animal@2d98a335], cn.yorick.serialization.House@16b98e56  , Ralph the hamster[cn.yorick.serialization.Animal@7ef20235], cn.yorick.serialization.House@16b98e56  , Molly the cat[cn.yorick.serialization.Animal@27d6c5e0], cn.yorick.serialization.House@16b98e56  ]  animals5: [Bosco the dog[cn.yorick.serialization.Animal@2d98a335], cn.yorick.serialization.House@16b98e56  , Ralph the hamster[cn.yorick.serialization.Animal@7ef20235], cn.yorick.serialization.House@16b98e56  , Molly the cat[cn.yorick.serialization.Animal@27d6c5e0], cn.yorick.serialization.House@16b98e56  ] |

运行这个例子，我们发现在注释的语句处，我们改变了animals，然后再次在oos1中保存,最后取出的却是改变前的和第一次存储时内容相同的,此时我们就更能说明问题了,此时前后的对象状态已经发生了改变,但是没有变的是序列号,因为当对象创建时序列号就已经根据这个对象的状态生成了,并且在后期在这个对象基础上所做的任何改变都不会影响到这个序列化号,所以序列化时仍会将其作为一个对象存储,但是oos2存储时,由于该序列号是第一次出现,所以存放的是改变过的状态,那么问自己一个问题,如果我们new两个不同状态的对象,存储和取出会如何呢?下面我们来看看:

|  |
| --- |
| **public** **class** House **implements** Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** **int** i;  **public** **static** **void** main(String[] args) **throws** FileNotFoundException, IOException, ClassNotFoundException {  House h1 = **new** House();  House h2 = **new** House();  h1.i = 11;  h2.i = 12;  ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream("house.out"));  oos.writeObject(h1);  oos.writeObject(h2);  h1.i = 13;  h2.i = 14;  oos.writeObject(h1);  oos.writeObject(h2);  ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream("house.out"));  House h3 = (House) ois.readObject();  House h4 = (House) ois.readObject();  House h5 = (House) ois.readObject();  House h6 = (House) ois.readObject();  System.***out***.println((h3 == h4) + "-" + (h3 == h5) + "-" + (h4 == h6));  System.***out***.println(h3.i + "--" + h4.i);  }  } |

我们看到这是实际上序列化后是存储了两个对象,这和上面的同理,new的两个对象生成的序列号是不同的,所以存储两次,但是后来在对象基础上改变状态,并没有改变序列化号,所以并没有再重复序列化。在这里我们看到类中出现了一个serialVersionUID字段，但是这并没有对结果有什么影响，所以我们可以推断序列化号和serialVersionUID是并不相容的概念，那么serialVersionUID是做什么的呢？下节我们来探讨一下；

|  |
| --- |
| 总结：当对象具有相同序列号时它只会存入一次，反序列化后相同序列号只会创建一个对象，其它相同序列号反序列化读取对象会将第一次创建的指针返回给它们。 |

#### 10.1.1.4 序列化ID与版本控制

上面我们阐述了序列化号的重要性，并且引入了***serialVersionUID***，我们经常在工具类中特别是JDK提供的工具类中见到serialVersionUID。那么它是做什么的呢？

其实它主要应用于反序列化时，由于我们序列化保存的对象可能会存放很长时间，你在这期间无法保证序列化对象对应的类在这期间是否发生了什么变化，所以就需要一个serialVersionUID来进行一些版本一致性验证检查。

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。在上面House代码中，我们将serialVersionUID字段去掉，然后进行序列化，序列化后，我们改变原有的类结构，添加字段，我们来看看会发生什么吧？

|  |
| --- |
| **public** **class** House **implements** Serializable {  // private static final long serialVersionUID = 1L;  **private** **int** i;  // private String field;  **public** **static** **void** main(String[] args) **throws** FileNotFoundException, IOException, ClassNotFoundException {  House h1 = **new** House();  h1.i = 11;  ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream("house.out"));//需注释  oos.writeObject(h1);//需注释  ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream("house.out"));  House h3 = (House) ois.readObject();  System.***out***.println(h3.i);//11  }  } |

我们现在将注释的String字段去掉注释,将序列化代码注释掉,然后进行反序列测试,结果如下:

|  |
| --- |
| Exception in thread "main" java.io.InvalidClassException: cn.yorick.serialization.House; local class incompatible: stream classdesc serialVersionUID = 3109553457046805864, local class serialVersionUID = -1901998781927510664  at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:687)  at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1880)  at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1746)  at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:2037)  at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1568)  at java.io.ObjectInputStream.readObject(ObjectInputStream.java:428)  at cn.yorick.serialization.House.main(House.java:38) |

这是由于此时类发生了变化,所以编译器根据类的状态等因素生成的UID也发生了变化,在反序列化时,根据前后两个UID判断,返现版本发生了改变,所以抛出了无效的类异常。

实现Serialization接口的类建议设置serialVersionUID字段值，如果不设置，那么每次运行时编译器会根据类的内部实现，包括类名、接口名、方法和属性等来自动生成serialVersionUID。如果类的源代码有修改，那么重新编译后serialVersionUID的取值可能会发生变化。因此实现Serialization接口的类一定要显式地定义serialVersionUID属性值。修改类时需要根据兼容性决定是否修改serialVersionUID值：

* 如果是兼容升级，请不要修改serialVersionUID字段，避免反序列化失败。
* 如果是不兼容升级，需要修改serialVersionUID值，避免反序列化混乱。

这里的兼容指的是升级后原有的对象反序列化是否能够正确工作。

有时我们可能想要改变可序列化类的版本（比如源类的对象可能保存在数据库中）。虽然Java支持这种做法，但是那可能只在特殊的情况下才这样做，此外，还需要对它有相当深程度的了解，这里我们就不再做过多赘述。

我们在JDK文档中发现有许多注解是从下面的文字开始的：

警告：该类的序列化对象和未来的Swing版本不兼容。当前对序列化的支持只适用于短期存储或应用之间的RMI。

这是因为Java的版本控制过于简单，因而不能在任何场合都可靠运转，尤其是对JavaBeans更是如此。有关人员正在设法修改这一设计，也就是警告中的相关部分。

#### 10.1.1.5 序列化的控制

前面对序列化的认识都是建立在默认的序列化机制，正如所看到的一样，只要我们按照规范进行，那么默认的序列化机制实际上并不难操控。然而如果我们在序列化的机制中有了自己的特别需要该怎么办呢？如为了安全考虑我们不希望对象的某一部分被序列化，如密码这些敏感数据，或者一个对象被还原后，某子对象需要重新创建，从而不必将子对象序列化。Java提供了很多方式来满足我们这样的需求,下面我们来认识一下,并且能够在以后实际应用中选择出最适合自己的方式。

##### 10.1.1.5.1 Externalizable接口

可通过实现Externalizable接口代替实现Serializable接口来对序列化过程进行控制.这个Externalizable接口继承了Serializable接口,不同于Serializable接口仅仅起到标识作用Externalizable接口增添了两个方法:writeExternal和readExternal。这两个方法会在序列化和反序列化还原的过程中被自动调用，以便执行一些特殊操作。

下面我们来看一下Externalizable接口方法的简单实现。注意Blip1和Blip2除了细微差别以外，几乎完全一致，仔细观察找到区别：

|  |
| --- |
| **public** **class** Blip1 **implements** Externalizable {  **public** Blip1() {  System.***out***.println("Blip1 Constructor");  }  @Override  **public** **void** writeExternal(ObjectOutput out) **throws** IOException {  System.***out***.println("Blip1 writeExternal!");  }  @Override  **public** **void** readExternal(ObjectInput in) **throws** IOException, ClassNotFoundException {  System.***out***.println("Blip1 readExternal!");  }  } |
| **public** **class** Blip2 **implements** Externalizable {  Blip2() {  System.***out***.println("Blip2 Constructor");  }  @Override  **public** **void** writeExternal(ObjectOutput out) **throws** IOException {  System.***out***.println("Blip2 writeExternal!");  }  /\*public Blip2() {  System.out.println("Blip2 Constructor");  }  \*/ @Override  **public** **void** readExternal(ObjectInput in) **throws** IOException, ClassNotFoundException {  System.***out***.println("Blip2 readExternal!");  }  } |
| **public** **class** Blips{  **public** **static** **void** main(String[] args) **throws** FileNotFoundException, IOException, ClassNotFoundException {  System.***out***.println("Constructing objects:");  Blip1 blip1 = **new** Blip1();  Blip2 blip2 = **new** Blip2();  ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream("Blips.out"));  System.***out***.println("saving objects:");  oos.writeObject(blip1);  oos.writeObject(blip2);  oos.close();  ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream("Blips.out"));  System.***out***.println("Recovering b1:");  // blip2 = (Blip2) ois.readObject();//注意序列化和反序列化时存放对象和取出对象书序要一致,否则很容易发生ClassCastException  blip1 = (Blip1) ois.readObject();  /\*\*  \* 此时会出现异常 java.io.InvalidClassException: cn.yorick.externalization.Blip2; no valid constructor  \* 这是因为在序列化是调用构造函数,但是Blip2的构造函数不是public的,而可控制序列化需要首先调用公共无参的构造函数,Blip2没有  \* 解决办法是添加一个:公共的无参构造函数  \*/  blip2 = (Blip2) ois.readObject();  }  } |
| Constructing objects:  Blip1 Constructor  Blip2 Constructor  saving objects:  Blip1 writeExternal!  Blip2 writeExternal!  Recovering b1:  Blip1 Constructor  Blip1 readExternal!  Exception in thread "main" java.io.InvalidClassException: cn.yorick.externalization.Blip2; no valid constructor  at java.io.ObjectStreamClass$ExceptionInfo.newInvalidClassException(ObjectStreamClass.java:157)  at java.io.ObjectStreamClass.checkDeserialize(ObjectStreamClass.java:862)  at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:2038)  at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1568)  at java.io.ObjectInputStream.readObject(ObjectInputStream.java:428)  at cn.yorick.externalization.Blips.main(Blips.java:33) |

运行我们发现并没有恢复Blip2对象，并且抛出了无效的类异常。观察发现Blip1和Blip2中不同的地方就在于Blip1的构造函数是公共无参的，而Blip2中的构造是默认权限的，当我们将默认权限升级为公共的时候，在测试会发现是下面的正确结果：

|  |
| --- |
| Constructing objects:  Blip1 Constructor  Blip2 Constructor  saving objects:  Blip1 writeExternal!  Blip2 writeExternal!  Recovering b1:  Blip1 Constructor  Blip1 readExternal!  Blip2 Constructor  Blip2 readExternal! |

实现Externalizable接口恢复序列化的对象会调用对象对应类的默认构造器(无参公共构造器，且只能是该构造函数)。这与恢复一个Serializable对象不同。对于Serializable对象，对象完全以它存储的二进制位为基础来构造，而不调用构造器。而对于一个Externalizable对象，所有普通的默认构造器都会被调用（包括在字段定义时的初始化），然后调用readExternal方法，必须注意这一点--所有默认的构造器都会被调用，才能使Externalizable对象产生正确的行为，所以在通过实现Externalizable接口序列化的时候，如果没有默认的构造器，需要显式的构造它，防止出现上面出现的异常。

下面我们来看一下如何完整保存和恢复一个Externalizable对象：

|  |
| --- |
| **public** **class** Blip3 **implements** Externalizable {  **private** **transient** **int** i;  **private** String s;  **public** Blip3() {  System.***out***.println("Blip3 Constructor");  }  **public** Blip3(**int** i, String s) {  **super**();  System.***out***.println("Blip3(int i, String s)");  **this**.i = i;  **this**.s = s;  }  @Override  **public** **void** writeExternal(ObjectOutput out) **throws** IOException {  System.***out***.println("序列化是会调用该方法,但要注意序列化的数据操作也是在这里控制的");  out.writeObject(s);  out.writeInt(i);  }  @Override  **public** String toString() {  **return** s + i;  }  @Override  **public** **void** readExternal(ObjectInput in) **throws** IOException, ClassNotFoundException {  System.***out***.println("反序列化时调用该方法,需要注意的有两点,第一是反序列化的操作,要在这里实现,第二是反序列化和序列化的顺序要对应!");  s = (String) in.readObject();//如果注释掉这两句,那么反序列化后就是一个字段是初始化值的对象,即s=null i=0  i = in.readInt();  }  **public** **static** **void** main(String[] args) **throws** FileNotFoundException, IOException, ClassNotFoundException {  System.***out***.println("Constructing objects:");  Blip3 blip3 = **new** Blip3(47, "A String");  System.***out***.println(blip3);  ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream("Blip3.out"));  oos.writeObject(blip3);  oos.close();  ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream("Blip3.out"));  System.***out***.println("Recovering blip3");  blip3 = (Blip3) ois.readObject();  System.***out***.println(blip3);  }  } |
| Constructing objects:  Blip3(int i, String s)  A String47  序列化是会调用该方法,但要注意序列化的数据操作也是在这里控制的  Recovering blip3  Blip3 Constructor  反序列化时调用该方法,需要注意的有两点,第一是反序列化的操作,要在这里实现,第二是反序列化和序列化的顺序要对应!  A String47 |

代码中s和i只在第二个在构造器中初始化,而不是在默认的构造其中初始化。这意味着假如不在readExternal中初始化s和i，s就会为nul，li就会为0（因为在创建对象的第一步中就会将对象的存储空间清理为0），如果我们注释掉readExternal中的两句对s和i的操作语句,那么救过也会是null和0。

我们如果从一个Externalizable对象继承，通常需要调用基类版本的writeExternal和readExternal来为基类组件完成恰当的存储和恢复功能。

因此，为了正常运行，我们不仅需要在writeExternal方法(没有任何默认行为来为Externalizable对象写入任何成员对象)中将来自对象的重要信息写入,还必须在readExternal方法中恢复数据。起先，可能会有一点迷惑，因为Externalizable对象的默认构造行为使其看起来似乎像某种自动发生的存储与恢复操作，实际上并非如此。

注意：可以同时实现Externalizable接口和Serializable接口，但是在实际序列化和反序列化时，仍然是通过调用实现Externalizable的方式来解决的，这个我们要注意。

总结;Externalizable对象中的共有writeExternal和readExternal方法对包括超类数据在内的整个对象的存储和恢复负全责。在写出对象的时候，序列化机制在输出流中仅仅只是记录该对象所属的类。在读入可外部化的类时，对象输入流将用无参构造器创建一个对象，然后调用readExternal方法。

##### 10.1.1.5.2 transient（瞬时关键字）

当我们对序列化进行控制时,可能某个特定子对象不想让Java的序列化机制自动保存和恢复.如果子对象表示的是我们不希望将其序列化的敏感信息（如密码），通常就会面临这种情况。即使对象中的这些信息是private属性，已经序列化处理，人们就可以通过读取文件或者拦截网络传输的方式来访问到它，这本身也是对封装的巨大破坏，是安全的巨大漏洞。

上面我们的Externalizable接口已经可以满足我们这个需求，但是Externalizable对象，没有任何东西可以自动序列化，并且可以在writeExternal内部只对所需部分进行显式的初始化。然而，如果我们正在操作的是一个Serializable对象，那么所有的序列化操作都是自动进行的。为了能够予以控制，可以使用transient（瞬时关键字）逐个字段的关闭序列化，它的意思是“不用麻烦你保存或者恢复数据-我自己会处理的”，如下面这个例子，假设某个Logon对象保存某个特定的登录会话信息，登录的合法性经过校验后，我们把数据保存下来，但是不包括敏感型的密码，为做到这一点最简单的方法就是实现Serializable，让其自动完成，且将password字段设置为transient，下面是具体代码：

|  |
| --- |
| **public** **class** Logon **implements** Serializable {  **private** Date date = **new** Date();  **private** String username;  **private** **transient** String password;  **public** Logon(String username, String password) {  **super**();  **this**.username = username;  **this**.password = password;  }  @Override  **public** String toString() {  **return** "Logon info: \n username: " + username +  "\n date: " + date + "\n password: " + password;  }  **public** **static** **void** main(String[] args) **throws** FileNotFoundException, IOException, ClassNotFoundException {  Logon logon = **new** Logon("Yorick", "guess");  System.***out***.println("logon logon = " + logon);  ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream("Logon.out"));  oos.writeObject(logon);  oos.close();  ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream("Logon.out"));  System.***out***.println("recocering object at " + **new** Date());  logon = (Logon) ois.readObject();  System.***out***.println("logon logon = " + logon);  }  } |
| logon logon = Logon info:  username: Yorick  date: Sun Jan 06 13:09:06 CST 2019  password: guess  recocering object at Sun Jan 06 13:09:06 CST 2019  logon logon = Logon info:  username: Yorick  date: Sun Jan 06 13:09:06 CST 2019  password: null |

可以看到Date和Username域是一般的,没有transient修饰的,所以它们会被自动序列化,而password是transient的,所以不会被自动保存到硬盘；另外，自动序列化机制也不会尝试去恢复它。当对象被恢复时，password域就会变成null。这里需要注意的是，虽然toString是用重载后的+操作符来连接String对象的，但是null引用会被自动装换成字符串null，这是StringBuilder的append方法决定的，如下源码：

我们还可以发现Date字段被存储到了磁盘并从磁盘上被恢复了出来，并且没有再重新生成。

由于Externalizable对象在默认情况下不保存它们的任何字段，所以transient关键字只能和Serializable对象一起使用。但是Externalizable对象中使用transient字段并不会出现编译和运行报错，但是这样没有丝毫意义，所以也不建议这么做。

当然这里将Serializable对象实现Externalizable也是可以满足需求的，但是对比就会发现，实现Externalizable的话，我们需要自己重写方法，增加代码量，而通过transient关键字，Serializable能更好的帮助我们自动完成序列化和反序列化，所以这里transient关键字更加适合。

##### 10.1.1.5.3 Externalizable的替代方案

如果，我们并不是特别坚持通过实现Externalizable接口的方式来实现序列化控制，那么Java还提供了一种方法，我们仍然可以通过实现Serializable接口来完成序列化控制，并添加writeObject和readObject方法，注意这里说的是添加而非覆盖或者实现（因为Serializable接口仅仅只是标识接口，并没有抽象方法来给我们实现或覆盖），这样一旦对象被序列化或者反序列化还原，就会自动地分别调用这两个方法。也就是说。只要我们提供了这两个方法，就会使用它们而不是默认的序列化机制。但是Java也要求这些方法具有特别的方法签名：

|  |
| --- |
| **private** **void** writeObject(ObjectOutputStream out) **throws** IOException {}  **private** **void** readObject(ObjectInputStream in) **throws** IOException {} |

从设计的观点来看，现在事情变得有些不可思议。首先，我们可能会认为由于这些方法不是基类或者Serializable接口的一部分，所以应该在它们自己的接口中进行定义。但是注意它们被定义成了private，这意味着它们仅能被这个类的其他成员调用。然而，实际上我们并没有从这个类的其它方法中调用它们。而是ObjectOutputStream和ObjectInputStream对象的writeObject和readObject方法调用你的对象的writeObject和readObject方法（注意这里用到的相同的方法名，我们不要去过分关注它，这种设计显得混乱，并不十分好）。我们想知道的可能是ObjectOutputStream和ObjectInputStream对象是怎么访问你的类中的private方法的。我们只能假设这正是序列化的神奇之处了。

在接口中定义的所有东西自动都是public的，因此如果writeObject和readObject必须是private的，那么它们不会是接口的一部分。因为我们必须要完全遵循其方法特征签名，所以其效果就和实现了接口一样。

在调用ObjectOutputStream.writeObject时，会检查所传递的Serializable对象，看看是否实现了它自己的writeObject。如果是这样，就跳过正常的序列化过程并调用它的writeObject（），readObject的情形与此相同。

还有另外一个技巧是，在你的writeObject内部，可以调用defaultWriteObject（）来选择执行默认的writeObject。类似的，在readObject内部，我们可以调用defaultReadObject（），下面这个例子就演示了如何对一个Serializable对象的存储和恢复就行控制。

|  |
| --- |
| **public** **class** SerialCtl **implements** Serializable {  **private** String a;  **private** **transient** String b;//我们知道transient修饰的成员是不能被实例化的,但是我们可以通过重写序列化方法,来使得该成员也可以被序列化  **public** SerialCtl(String a, String b) {  **super**();  **this**.a = "Not Transient : " + a;  **this**.b = "Transient : " + b;  }  @Override  **public** String toString() {  // **TODO** Auto-generated method stub  **return** a + "\n" + b;  }  //注意重写序列化和反序列化方法的发签名特征是固定的  **private** **void** writeObject(ObjectOutputStream out) **throws** IOException {  //若将下面序列化,则将会将transient修饰的成员序列化,注释后不序列化  out.writeObject(b);  out.defaultWriteObject();  }  **private** **void** readObject(ObjectInputStream in) **throws** ClassNotFoundException, IOException {    //若下面依据注释,那么不会读出  // b = (String) in.readObject();  in.defaultReadObject();  }  **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException {  SerialCtl ctl = **new** SerialCtl("Test1", "Test2");  System.***out***.println("Before:\n" + ctl);  ByteArrayOutputStream bos = **new** ByteArrayOutputStream();  ObjectOutputStream oos = **new** ObjectOutputStream(bos);  oos.writeObject(ctl);  ObjectInputStream ois = **new** ObjectInputStream(**new** ByteArrayInputStream(bos.toByteArray()));  SerialCtl ctl2 = (SerialCtl) ois.readObject();  System.***out***.println("After:\n" + ctl2);  }  } |
| **将注释去掉运行结果如下：**  Before:  Not Transient : Test1  Transient : Test2  After:  Not Transient : Test1  Transient : Test2 |

上面的例子中，有一个String字段是普通字段，而另一个是transient字段，用来证明非transient字段是由defaultWriteObject方法保存,而transient字段必须在程序中明确保存和恢复。字段是在构造器内部而不是在定义处进行初始化的，以此可以证实它们在反序列化还原期间没有被一些自动化机制初始化。即反序列化并非是由构造函数构造的。实际上同普通Serializable对象一样，反序列化也是通过文件直接构造对象，而不是通过调用构造器，不同的是具有可控序列化的Serializable对象是通过自己类中的readObject方法来进行反序列化的，

如果我们打算使用默认机制写入对象的非transient部分，那么必须调用defaultWriteObject（）作为writeObject中的第一个操作，并让defaultReadObject（）作为readObject中的第一个操作（实际上测试发现，并非一定要让default方法在第一行，但一定要注意序列化和反序列化的对应顺序，但我们仍建议将其放在第一行），这些在我们看来都是奇怪的方法调用。例如，如果我们正在为ObjectOutputStream调用defaultWriteObject（）且没有传递任何参数，然而不知何故它却能运行，并且知道对象的引用以及如何写入非transient部分。这看起来奇怪之极。

对transient对象的存储和恢复使用了我们比较熟悉的代码。请再考虑一下在这里所发生的事情。在main中，创建SerialCtl 对象，并且将其序列化到ObjectOutputStream（注意这里使用的是字节缓冲区，而不是文件，不过对于输出流来说它们只是输出的一个目的地，并没有太大的区别）。真正的序列化发生在下面这行代码当中：

|  |
| --- |
| oos.writeObject(ctl); |

writeObject方法必须检查ctl，判断它是否拥有自己的writeObject方法（不是检查接口，这里根本就没有接口，也不是检查类的类型，而是利用反射来真正的搜索方法）。如果有，那么就会使用它。对readObject也采用类似的方法。或许这是解决这个问题的唯一切实可行的方法，但它确实有点古怪。

|  |
| --- |
| 注意：如果在类中写自己的writeObject和readObject的时候，方法签名一定要是规定的方法签名，如果不同，那么并不会被序列化和反序列化认为是规范中可以被调用的，我们举例如下：   * 当writeObject是public的时候，而readObject不变时，此时测试如下：   Before:  Not Transient : Test1  Transient : Test2  Exception in thread "main" java.io.OptionalDataException  at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1529)  at java.io.ObjectInputStream.readObject(ObjectInputStream.java:428)  at cn.yorick.serialization.override.SerialCtl.readObject(SerialCtl.java:42)  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)  at java.lang.reflect.Method.invoke(Method.java:498)  at java.io.ObjectStreamClass.invokeReadObject(ObjectStreamClass.java:1158)  at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:2173)  at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:2064)  at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1568)  at java.io.ObjectInputStream.readObject(ObjectInputStream.java:428)  at cn.yorick.serialization.override.SerialCtl.main(SerialCtl.java:53)   * 当writeObject是private，而readObject是public的时候，那么此时发现：   Before:  Not Transient : Test1  Transient : Test2  After:  Not Transient : Test1  Null   * 当两个方法都为public时，那么此时发现   Before:  Not Transient : Test1  Transient : Test2  After:  Not Transient : Test1  null |

我们来分析上面三种情况，第一种情况，当序列化时，ObjectOutputStream从当前类查看是否有符合规范的writeObject方法，由于此时writeObject方法签名不符合，所以调用Serializable的默认序列化机制，此时生成的持久化对象文件中只存放了对象中非transient字段的成员，所以在readObject时由于b = (String) in.readObject();并没有该字段，所以报错，我们如果将该条语句注释，那么结果会是下面这样

|  |
| --- |
| Before:  Not Transient : Test1  Transient : Test2  After:  Not Transient : Test1  null |

第二种情况，我么在序列化时，由于本类中有符合条件的writeObject方法，所以使用该方法进行序列化，但是反序列化时使用的是Serializable的默认反序列化机制，所以默认transient字段会被赋值为null，并且当我们尝试从持久化对象文件中读取写入的String字段时，也会出现如下错误：

|  |
| --- |
| ObjectInputStream ois = **new** ObjectInputStream(**new** ByteArrayInputStream(bos.toByteArray()));  SerialCtl ctl2 = (SerialCtl) ois.readObject();  String b = (String) ois.readObject();  System.***out***.println("After:\n" + ctl2);  System.***out***.println(b); |
| Before:  Not Transient : Test1  Transient : Test2  Exception in thread "main" java.io.EOFException  at java.io.ObjectInputStream$BlockDataInputStream.peekByte(ObjectInputStream.java:2954)  at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1535)  at java.io.ObjectInputStream.readObject(ObjectInputStream.java:428)  at cn.yorick.serialization.override.SerialCtl.main(SerialCtl.java:54) |

在第三种情况中，实际序列化和反序列化都是采用Serializable的默认形式；

由此可见，如果我们想通过Serializable提供的可控制的序列化机制，那么遵循其独有的方法签名是关键。

#### 10.1.1.6 使用持久性

一个比较诱人的使用序列化技术的想法是:存储程序的一些状态,以便我们随后可以很容易地将程序恢复到当前状态。但是在我们能够这样做之前，必须回答几个问题。如果我们将两个对象--它们都具有指向第三个对象的引用---进行序列化，会发生什么情况呢？当我们从它们的序列化状态恢复这两个对象时，第三个对象会只出现一次吗？如果将这两个对象序列化成独立的文件，然后在代码的不同部分对它们进行反序列化还原，又会怎么样呢？

下面这个例子就是为了说明上述问题：

|  |
| --- |
| **public** **class** House **implements** Serializable {} |
| **public** **class** Animal **implements** Serializable {  **private** String name;  **private** House preferredHouse;  **public** Animal(String nm, House h) {  **super**();  **this**.name = nm;  **this**.preferredHouse = h;  }  @Override  **public** String toString() {  // **TODO** Auto-generated method stub  **return** name + "[" + **super**.toString() + "], " + preferredHouse + "\n";  }  } |
| **public** **class** MyWorld {  **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException {  House house = **new** House();  List<Animal> animals = **new** ArrayList<>();  animals.add(**new** Animal("Bosco the dog", house));  animals.add(**new** Animal("Ralph the hamster", house));  animals.add(**new** Animal("Molly the cat", house));  System.***out***.println("animals: " + animals);  ByteArrayOutputStream bos1 = **new** ByteArrayOutputStream();  ObjectOutputStream oos1 = **new** ObjectOutputStream(bos1);  oos1.writeObject(animals);  oos1.writeObject(animals);  // System.***out***.println(bos1.size());  // oos1.writeObject(animals);  // System.***out***.println(bos1.size());  // animals.add(**new** Animal("Molly the cats", house));  // System.out.println(animals);  // oos1.writeObject(animals);  // System.***out***.println(bos1.size());  ByteArrayOutputStream bos2 = **new** ByteArrayOutputStream();  ObjectOutputStream oos2 = **new** ObjectOutputStream(bos2);  oos2.writeObject(animals);  ObjectInputStream ois1 = **new** ObjectInputStream(**new** ByteArrayInputStream(bos1.toByteArray()));  //我们在这里改变了对象,并且重新序列化,反序列化后发现对象是变化前的对象,这是由于序列化机制,序列化时会  //判断序列化的对象是否为同一个对象,判断依据是内存地址,如果相同那么只会序列化一次,  //这也就是为什么我们重复序列化同一个对象,重复读取后是一样的,因为序列化后的文件中只有一份  ObjectInputStream ois2 = **new** ObjectInputStream(**new** ByteArrayInputStream(bos2.toByteArray()));  List<Animal> animals1 = (List<Animal>) ois1.readObject();  List<Animal> animals2 = (List<Animal>) ois1.readObject();  // List<Animal> animals4 = (List<Animal>) ois1.readObject();  List<Animal> animals3 = (List<Animal>) ois2.readObject();  // List<Animal> animals5 = (List<Animal>) ois1.readObject();  System.***out***.println("animals1: " + animals1);  System.***out***.println("animals2: " + animals2);  System.***out***.println("animals3: " + animals3);  // System.***out***.println("animals4: " + animals4);  // System.***out***.println("animals5: " + animals5);  }  } |
| animals: [Bosco the dog[cn.yorick.serialization.Animal@7852e922], cn.yorick.serialization.House@4e25154f  , Ralph the hamster[cn.yorick.serialization.Animal@70dea4e], cn.yorick.serialization.House@4e25154f  , Molly the cat[cn.yorick.serialization.Animal@5c647e05], cn.yorick.serialization.House@4e25154f  ]  animals1: [Bosco the dog[cn.yorick.serialization.Animal@378bf509], cn.yorick.serialization.House@5fd0d5ae  , Ralph the hamster[cn.yorick.serialization.Animal@2d98a335], cn.yorick.serialization.House@5fd0d5ae  , Molly the cat[cn.yorick.serialization.Animal@16b98e56], cn.yorick.serialization.House@5fd0d5ae  ]  animals2: [Bosco the dog[cn.yorick.serialization.Animal@378bf509], cn.yorick.serialization.House@5fd0d5ae  , Ralph the hamster[cn.yorick.serialization.Animal@2d98a335], cn.yorick.serialization.House@5fd0d5ae  , Molly the cat[cn.yorick.serialization.Animal@16b98e56], cn.yorick.serialization.House@5fd0d5ae  ]  animals3: [Bosco the dog[cn.yorick.serialization.Animal@7ef20235], cn.yorick.serialization.House@27d6c5e0  , Ralph the hamster[cn.yorick.serialization.Animal@4f3f5b24], cn.yorick.serialization.House@27d6c5e0  , Molly the cat[cn.yorick.serialization.Animal@15aeb7ab], cn.yorick.serialization.House@27d6c5e0  ] |

这里有一件有趣的事:我们可以通过一个字节数组来使用对象序列化,从而实现对任何可Serializable对象的深度复制--深度复制意味着我们复制的是整个对象网,而不仅仅是基本对象及其引用。

在这个例子中，Animal对象包含有House类型的字段。在main方法中，创建一个Animal列表并将其两次序列化，分别送至不同的流。当其被反序列化还原并被打印时，我们可以看到所示的执行某次运行后的结果（每次运行时，对象将会处在不同的内存地址，但同一个对象的在同一个流的两次序列化后，反序列化后内存中是一个内存地址，两次反序列化得到的对象都指向这个内存地址）。

当然，我们期望这些反序列化还原后的对象地址与原来的地址不同。但请注意，在animals1和animals2中出现了相同的地址，包括二者共享的那个原来指向House对象的引用。另一方面，当恢复animals3时，系统无法知道另一个流内的对象是第一个流内对象的别名，因此它会产生完全不同的对象网。

只要将任何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，并且没有任何意外重复复制出的对象。当然，我们可以在写出第一个对象和写出最后一个对象期间改变这些对象的状态，但是这是我们自己的事情；无论对象在被序列化时出于什么状态（无论它们和其它对象有什么样的连接关系），它们都可以被写出。

如果我们想保存系统状态，最安全的做法是将其作为“原子”操作进行序列化。如果我们序列化了某些东西，再去做其他一些工作，再来序列化更多的东西，如此等等，那么将无法安全的保存系统状态。取而代之的是，将构成系统状态的所有对象都置入单一容器内，并在一个操作中将该容器直接写出。然后同样只需一次方法调用，既可以将其恢复。

下面这个例子是一个想象中的计算机辅助设计（CAD）系统，该例演示了这一方法。此外还引入了static字段的问题;如果我们查看JDK文档,就会发现Class是Serializable的,因此只需直接对Class对象序列化,就可以很容易保存static字段。在任何情况下这都是一种明智的做法。

|  |
| --- |
| **public** **abstract** **class** Shape **implements** Serializable{  **public** **static** **final** **int** ***RED*** = 1, ***BLUE*** = 2, ***GREEN*** = 3;  **private** **int** xPos, yPos, dimension;  **private** **static** Random *random* = **new** Random(47);  **private** **static** **int** *counter* = 0;  **public** **abstract** **void** setColor(**int** newColor);  **public** **abstract** **int** getColor();  **public** Shape(**int** xPos, **int** yPos, **int** dimension) {  **super**();  **this**.xPos = xPos;  **this**.yPos = yPos;  **this**.dimension = dimension;  }  @Override  **public** String toString() {  **return** getClass() + "color[" + getColor() + "] xPos[" +  xPos + "] yPos[" + yPos + "] dim[" + dimension + "]\n";  }  **public** **static** Shape randomFactory() {  **int** xVal = *random*.nextInt(100);  **int** yVal = *random*.nextInt(100);  **int** dimension = *random*.nextInt(100);  **switch** (*counter*++ % 3) {  **case** 0: **return** **new** Circle(xVal,yVal,dimension);  **case** 1: **return** **new** Square(xVal,yVal,dimension);  **case** 2: **return** **new** Line(xVal,yVal,dimension);  }  **return** **null**;  }  } |
| **public** **class** Circle **extends** Shape {  **private** **static** **int** *color* = ***RED***;  **public** Circle(**int** xPos, **int** yPos, **int** dimension) {  **super**(xPos, yPos, dimension);  }  @Override  **public** **void** setColor(**int** newColor) {  **this**.*color* = newColor;  }  @Override  **public** **int** getColor() {  **return** *color*;  }  } |
| **public** **class** Square **extends** Shape {  **private** **static** **int** *color*;  **public** Square(**int** xPos, **int** yPos, **int** dimension) {  **super**(xPos, yPos, dimension);  *color* = ***RED***;  }  @Override  **public** **void** setColor(**int** newColor) {  *color* = newColor;  }  @Override  **public** **int** getColor() {  **return** *color*;  }  } |
| **public** **class** Line **extends** Shape {  **private** **static** **int** *color* = ***RED***;  **public** **static** **void** serializeStaticState(ObjectOutputStream out) **throws** IOException {  out.writeInt(*color*);  }  //怎么序列化静态成员  **public** **static** **void** deserializeStaticState(ObjectInputStream os) **throws** IOException {  *color* = os.readInt();  }  **public** Line(**int** xPos, **int** yPos, **int** dimension) {  **super**(xPos, yPos, dimension);  }  @Override  **public** **void** setColor(**int** newColor) {  *color* = newColor;  }  @Override  **public** **int** getColor() {  // **TODO** Auto-generated method stub  **return** *color*;  }  } |
| **public** **class** StoreCADState {  **public** **static** **void** main(String[] args) **throws** IOException {  List<Class<? **extends** Shape>> shapeTypes = **new** ArrayList<>();  shapeTypes.add(Circle.**class**);  shapeTypes.add(Square.**class**);  shapeTypes.add(Line.**class**);  List<Shape> shapes = **new** ArrayList<>();  **for** (**int** i = 0; i < 10; i++) {  shapes.add(Shape.*randomFactory*());  }  **for** (**int** i = 0; i < 10; i++) {  shapes.get(i).setColor(Shape.***GREEN***);  }  ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream("CADState.out"));  oos.writeObject(shapeTypes);  Line.*serializeStaticState*(oos);  oos.writeObject(shapes);  System.***out***.println(shapes);  }  } |
| [class cn.yorick.serialization.Circlecolor[3] xPos[58] yPos[55] dim[93]  , class cn.yorick.serialization.Squarecolor[3] xPos[61] yPos[61] dim[29]  , class cn.yorick.serialization.Linecolor[3] xPos[68] yPos[0] dim[22]  , class cn.yorick.serialization.Circlecolor[3] xPos[7] yPos[88] dim[28]  , class cn.yorick.serialization.Squarecolor[3] xPos[51] yPos[89] dim[9]  , class cn.yorick.serialization.Linecolor[3] xPos[78] yPos[98] dim[61]  , class cn.yorick.serialization.Circlecolor[3] xPos[20] yPos[58] dim[16]  , class cn.yorick.serialization.Squarecolor[3] xPos[40] yPos[11] dim[22]  , class cn.yorick.serialization.Linecolor[3] xPos[4] yPos[83] dim[6]  , class cn.yorick.serialization.Circlecolor[3] xPos[75] yPos[10] dim[42]  ] |

**Shape类实现了Serializable，所以任何自Shape继承的类也都会自动是Serializable的**。每个Shape都含有数据，而且每个派生自Shape的类都包含一个static字段，用来确定各种Shape类型的颜色（如果将static字段置入基类，只会产生一个static字段，因为static字段不能在派生类中复制，static字段相对是全局共享的，而非实例私有的）。可对基类中的方法进行重载，以便为不同的类型设置颜色（static方法不会动态绑定，所以这些都是普通的方法）。每次调用randomFactory（）方法时，它都会使用不同的随机数作为Shape的数据，从而重建不同的Shape。

在main中，一个ArrayList用于保存Class对象，而另一个用于保存几何形状。

|  |
| --- |
| **public** **class** RecoverCADState {  @SuppressWarnings({ "unused", "unchecked" })  **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException {  ObjectInputStream in = **new** ObjectInputStream(**new** FileInputStream("CADState.out"));  List<Class<? **extends** Shape>> shaperTypes = (List<Class<? **extends** Shape>>) in.readObject();//此时内存中已经有了类对象了,类一经加载  Line.*deserializeStaticState*(in);  List<Shape> shapes = (List<Shape>) in.readObject();  System.***out***.println(shapes);  }  } |
| [class cn.yorick.serialization.Circlecolor[1] xPos[58] yPos[55] dim[93]  , class cn.yorick.serialization.Squarecolor[0] xPos[61] yPos[61] dim[29]  , class cn.yorick.serialization.Linecolor[3] xPos[68] yPos[0] dim[22]  , class cn.yorick.serialization.Circlecolor[1] xPos[7] yPos[88] dim[28]  , class cn.yorick.serialization.Squarecolor[0] xPos[51] yPos[89] dim[9]  , class cn.yorick.serialization.Linecolor[3] xPos[78] yPos[98] dim[61]  , class cn.yorick.serialization.Circlecolor[1] xPos[20] yPos[58] dim[16]  , class cn.yorick.serialization.Squarecolor[0] xPos[40] yPos[11] dim[22]  , class cn.yorick.serialization.Linecolor[3] xPos[4] yPos[83] dim[6]  , class cn.yorick.serialization.Circlecolor[1] xPos[75] yPos[10] dim[42]  ] |

可以看出，xPos、yPos以及dim的值都被成功的保存和恢复了，但是对static信息的读取却出现了问题。所有读回的颜色应该都是“3”，但是真实情况却并非如此。Circle的值是1（定义为RED），而Square的值是0（记住，它们是在构造器中被初始化的）。看上去似乎static数据根本没有被序列化！确实如此--**尽管Class类是Serializable的，但它却不能按我们所期望的方式运行。所以加入想序列化static值，必须自己动手去实现**。

这正是Line中的serializeStaticState和deserializeStaticState两个static方法的用途。可以看到，它们是作为存储和读取过程的一部分被显式的调用。（注意必须维护写入序列化文件和从该文件中读回的顺序。）因此，为了使CADState.java正确运转起来,我们必须:

* 为几何形状添加serializeStaticState和deserializeStaticState两个方法;
* 移除ArrayList shapeTypes以及与之相关的所有代码
* 在几何形状内添加对新的序列化和反序列化还原静态方法的调用。

另一个要注意的问题是安全，因为序列化也会将private数据保存下来。如果你关心安全问题，那么应将其标记成transient。但是这之后，还必须设计一种安全的保存信息的方法，以便在执行时可以复位那些private变量。

#### 10.1.1.7 序列化与clone

序列化机制提供了一种很有趣的用法:既提供了一种克隆对象的简便途径,只要对应的类是可序列化的即可。其做法很简单：直接将对象序列化到输出流中，然后将其读回。这样产生的新对象是对现有对象的一个深拷贝（deep copy）。在此过程中，我们不必将对象写到文件中，因为可以用ByteArrayOutputStream将数据保存在字节数组中。

正如下面代码所示，要想得到clone方法，只需扩展SerialClone类，这样就完事了：

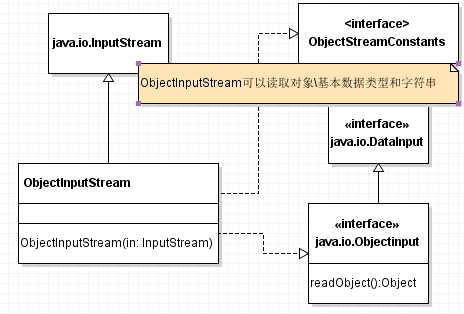
|  |
| --- |
| **public** **class** SerialCloneTest **implements** Cloneable, Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **public** **static** **void** main(String[] args) **throws** CloneNotSupportedException {  Employee2 harry = **new** Employee2("Harry Hacker", 35000, 1989, 10, 1);  Employee2 harry2 = (Employee2)harry.clone();  harry.raiseSalary(10);  System.***out***.println(harry);  System.***out***.println(harry2);  }  }  **class** SerialClonable **implements** Cloneable,Serializable{  @Override  **protected** Object clone() **throws** CloneNotSupportedException {  ByteArrayOutputStream bos = **new** ByteArrayOutputStream();  ObjectOutputStream oos;  **try** {  oos = **new** ObjectOutputStream(bos);  oos.writeObject(**this**);  } **catch** (IOException e1) {  // **TODO** Auto-generated catch block  e1.printStackTrace();  }  **try** {  ObjectInputStream ois = **new** ObjectInputStream(**new** ByteArrayInputStream(bos.toByteArray()));  **return** ois.readObject();  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } **catch** (ClassNotFoundException e) {  CloneNotSupportedException e2 = **new** CloneNotSupportedException();  e2.initCause(e);//将此throwable的原因初始化为指定值  **throw** e2;  }  **return** **null**;  }  } |
| **public** **class** Employee2 **extends** SerialClonable **implements** Serializable{  **private** String name;  **private** **double** salary;  **private** LocalDate hireDay;  **public** Employee2() {}  **public** Employee2(String n, **double** s, **int** year, **int** month, **int** day){  name = n;  salary = s;  hireDay = LocalDate.*of*(year, month, day);  }  **public** String getName(){  **return** name;  }  **public** **double** getSalary() {  **return** salary;  }  **public** LocalDate getHireDay(){  **return** hireDay;  }  **public** **void** raiseSalary(**double** byPercent){  **double** raise = salary \* byPercent / 100;  salary += raise;  }  **public** String toString() {  **return** getClass().getName()  + "[name=" + name  + ",salary=" + salary  + ",hireDay=" + hireDay  + "]";  }  } |
| cn.yorick.serialization.Employee2[name=Harry Hacker,salary=38500.0,hireDay=1989-10-01]  cn.yorick.serialization.Employee2[name=Harry Hacker,salary=35000.0,hireDay=1989-10-01] |

我们应该当心这个方法，尽管它很灵活，但是他通常会比显式的构建新对象并复制或克隆数据域的克隆方法慢得多。

#### 10.1.1.8 ObjectOutputStream与ObjectInputStream

前面我们系统的学习了Java原生的序列化，这节我们简单介绍一下用于对象序列化的两个流。ObjectOutputStream和ObjectInputStream是用于写读序列化的对象。Java中提供了另外两个流DataOutputStream和DataInputStream，这两个类可以实现基本数据类型与字符串的输出和输入。而ObjectInputStream和ObjectOutputStream除了读写对象外，也支持对基本数据类型和字符串的读写。由于ObjectOutputStream和ObjectInputStream类包含DataOutputStream和DataInputStream的所有功能，所以**完全可以用ObjectInputStream和ObjectOutputStream类来代替DataOutputStream和DataInputStream。**

ObjectInputStream扩展InputStream类，并实现接口ObjectInput和ObjectStreamConstants，ObjectInput是DataInput的子接口。ObjectStreamConstants包含支持ObjectInputStream和ObjectOutputStream类所用的常量。下面是以ObjectInputStream为例的类关系图。



我们可以向数据输出流中写多个数据，但是要求读数据的时候要按照写入的顺序来读取。读取返回的都是Object，为了得到所需的类型，必须使用java安全的类型转换。

readObject方法可能会抛出一个ClassNotFoundException。这是因为Java虚拟机恢复一个对象时，如果没有架子啊该对象所在的类，就应该先加载这个类。而且由于ClassNotFoundException是一个必检异常，所以，在方法中应该明处处理或者抛出。

当存储一个可序列化对象时，会对该对象的类进行编码。编码包括类名、类的签名、对象实例变量的值以及该对象引用的任何其他对象的闭包，但是不存储对象静态变量的值，如需保存，需要在自己的静态方法中进行处理。

需要注意的一个小问题是：如果序列化两个同样类型的对象，它们并不一定会占用相同的空间，如String，变长字符串类型相同，但占用空间是不同的。

#### 10.1.1.9 序列化单例和类型安全的枚举

在序列化和反序列化时，如果目标对象是唯一的，那么你必须加倍当心了，这通常会在实现单例和类型安全的枚举时发生。

如果你使用Java的enum结构，那么你就不必担心序列化，他能正常工作。但是，假如你在维护遗留代码，其中包含下面这样的枚举类型：

|  |
| --- |
| **public** **class** Orientation {  **public** **static** **final** Orientation ***HORIZONTAL*** = **new** Orientation(1);  **public** **static** **final** Orientation ***VERTICAL*** = **new** Orientation(2);  **private** **int** value;  **private** Orientation(**int** v) {  value = v;  }  } |

这种风格在枚举被添加到Java语言中之前是很普遍的。注意：其构造器是私有的。因此，不可能创建出超出Orientation.HORIZONTAL和Origination.VERTICAL之外的对象。特别是，你可以使用==操作符来测试对象的等同性：

|  |
| --- |
| If(orientation == Orientation.HORIZONTAL)....... |

当类型安全的枚举实现接口Serializable时,你必须牢记存在着一种重要的变化,此时,默认的序列化机制时不适用的.假如我们写出一个Orientation类型的值,并再次将其读回:

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException {  Orientation original = Orientation.***HORIZONTAL***;  ByteArrayOutputStream bos = **new** ByteArrayOutputStream();  ObjectOutputStream oos = **new** ObjectOutputStream(bos);  oos.writeObject(original);  oos.close();  ObjectInputStream ois = **new** ObjectInputStream(**new** ByteArrayInputStream(bos.toByteArray()));  Orientation orientation = (Orientation) ois.readObject();  **if**(orientation != original) {  System.***out***.println("枚举类型受到破坏");  }  System.***out***.println("end");  } |
| 枚举类型受到破坏  end |

实际上orientation 是Orientation类的一个全新的对象,它和任何预定义的常量都不等同。既使构造器是私有的，序列化机制也可以创建新的对象！

为了解决这个问题，你需要定义另一种称为readResolve的特殊序列化方法。如果定义了readResolve方法，在对象被序列化之后就会调用它。它必须返回一个对象，而该对象之后会成为readResolve的返回值。我们下面来改写上面的代码：

|  |
| --- |
| **protected** Object readResolve() **throws** ObjectStreamException{  **if**(value == 1) **return** Orientation.***HORIZONTAL***;  **if** (value == 2) **return** Orientation.***VERTICAL***;  **return** **null**;  } |

此时进行测试发现测试结果是end，证明的确起到了保护枚举的作用，单例同枚举的操作是相同的。

注意：记住向遗留代码中所有类型安全的枚举以及向所有支持单例设计模式的类中添加readResolve方法。

#### 10.1.1.10 序列化与继承

在前面，实际上我们已经多次提到过子类继承父类，如果父类时Serializable的，那么子类也会继承Serializable，这节我们详细说一下，并探讨一些应该注意的问题。

正常的子父类都实现Serializable我们就不在讨论，也都知道直接，我们要说的是在子父类中，如果父类没有实现Serializable，而子类实现序列化，那么此时会出现什么情况呢？

|  |
| --- |
| **public** **class** Collar **implements** Serializable{  **private** **int** size;  **private** String color;  **public** Collar(**int** size, String color) {  **super**();  **this**.size = size;  **this**.color = color;  }  **public** **int** getSize() {  **return** size;  }  **public** **void** setSize(**int** size) {  **this**.size = size;  }  **public** String getColor() {  **return** color;  }  **public** **void** setColor(String color) {  **this**.color = color;  }  @Override  **public** String toString() {  // **TODO** Auto-generated method stub  **return** getSize() + getColor();  }  } |
| **public** **class** Dog{  String name ; //狗的名字  Collar collar; //狗的类型  **public** Dog() {  **super**();  System.***out***.println("父类的无参构造!");  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** Collar getCollar() {  **return** collar;  }  **public** **void** setCollar(Collar collar) {  **this**.collar = collar;  }  @Override  **public** String toString() {  **return** getName() + "是" + getCollar();  }  } |
| **public** **class** Cibotium **extends** Dog **implements** Serializable{  **public** Cibotium() {  System.***out***.println("运行!");  }  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **public** **static** **void** main(String[] args){  Cibotium cibotium = **new** Cibotium();  cibotium.setCollar(**new** Collar(8, "golden"));  cibotium.setName("暖男");  **try** {  FileOutputStream fileOutputStream = **new** FileOutputStream("cibotium");  ObjectOutputStream os = **new** ObjectOutputStream(fileOutputStream);  os.writeObject(cibotium);  os.close();  } **catch** (Exception e) {  e.printStackTrace();  }  **try** {  FileInputStream fis = **new** FileInputStream("cibotium");  ObjectInputStream ois = **new** ObjectInputStream(fis);  Object object = ois.readObject();  Cibotium cibotium2 = (Cibotium) object;  ois.close();  System.***out***.println("序列化对象和反序列化的对象是否是同一个对象:" + (cibotium == cibotium2));  System.***out***.println("序列化前的对象:" + cibotium.toString());  System.***out***.println("反序列化后的对象:" + cibotium2.toString() + cibotium2.hashCode());  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |
| 父类的无参构造!  运行!  序列化对象和反序列化的对象是否是同一个对象:false  序列化前的对象:暖男是8golden  反序列化后的对象:null是null2003749087 |

为什么会出现这样的结果呢？这是由于父类没有实现Serializable，所以序列化时父类和祖父类都没有序列化，而反序列化时，由于父类没有序列化，所以此时默认的序列化机制会调用父类的无参构造函数，然后一层层的调用上层构造函数，调用完后，会调用子类的构造函数进行初始化，所以我们看到，虽然我们正常进行默认序列化机制，但是结果却是调用构造器初始化值。那么怎么避免这种情况呢？实际上了我么你通过序列化控制就可以很好的解决这个问题，我们这里仅作一种展示：

|  |
| --- |
| **private** **void** writeObject(ObjectOutputStream out) **throws** IOException {  out.defaultWriteObject();  out.writeObject(name);  out.writeObject(collar);  }  **private** **void** readObject(ObjectInputStream in) **throws** ClassNotFoundException, IOException {  in.defaultReadObject();  name = (String) in.readObject();  collar = (Collar) in.readObject();  } |

此时我们再次测试结果就和我们想的一样了如下：

|  |
| --- |
| 父类的无参构造!  运行!  父类的无参构造!  序列化对象和反序列化的对象是否是同一个对象:false  序列化前的对象:暖男是8golden  反序列化后的对象:暖男是8golden558638686 |

#### 补充阅读

|  |  |
| --- | --- |
| Question | Answer |
| 如果序列化这么重要，为什么不默认成每个类都有？为什么不让Object实现Serializable以让所有类都自动地成为可序列化的？ | 就算大部分的类都应该实现Serializable，你还是有选择性。且你必须有意识地对各个类做决定。如果它会变成默认的，那要如何关掉呢？毕竟界面是用来指示功能性而不是所缺乏的功能。 |
| 为什么我写过不需要序列化的类？ | 也许是因为安全性的理由让你不想把密码对象存储在文件上。或者某些对象的存储是没有意义的，所以你不会把它实现成可序列化的。这个问题关键还是在于自己对该类的判断，判断是否可以或有必要序列化。 |
| 如果我使用了一个不能序列化的类，我是否能把它给子类出一个标记为可序列化的类？ | 这其实是可以的，如果该类是可以继承的，你就可以制作出可以被序列化的类。当子类对象被还原且它的父类不可序列化时，父类的构造器会跟创建新的对象一起地执行。如果类没有什么好理由不能被序列化，制作可序列化的子类会是个好选择。 |
| 如果你将某个变量标记为transient的，那就代表说在序列化的过程中该变量会被略过。然后会发生什么事情呢？我们用transient标记变量来解决不能序列化的实例变量问题，但我们不是在恢复对象的时候需要该变量吗？序列化的重点不就是保存对象的状态吗？ | 没错，问题就在这里，但是幸好有解决办法，如果你把某个对象序列化，transient的引用实例变量会以null返回，而不管当时存储它的值是什么。这代表整个对象版图中连接到该特定实例变量的部分不会被存储。  但这样可能会有问题，所以我们有两个解决方案：  （1）当对象被带回来的时候，重新初始化实例变量回到默认的状态。如在对象中有一个成员（对象），但是该对象无关紧要，所以重建一个新的也不会有什么影响。  （2）如果transient变量的值很重要，那么你就得需要同时把它的值也保存下来，然后恢复时，再将该值恢复为原来的状态。 |
| 如果两个对象都有引用实例变量指向相同的对象会怎么样？ | 序列化聪明的足以分辨两个对象是否相同。在此情况下只有一个对象会被存储，其它引用会复原指向该对象。这一点在序列化号中有所呈现了。 |
| 序列化中都会经常遇到哪些异常？ | （1）反序列化是类不在ClassNotFoundException  （2）反序列化时读取对象次数超出存入的次数java.io.EOFException  （3）序列化的类中又不支持序列化的成员[NotSerializableException](mk:@MSITStore:E:\\常用API\\JDK%20API%201.8.CHM::/java/io/../../java/io/NotSerializableException.html" \o "class in java.io)  （4）InvalidClassException当序列化运行时检测到类中的以下问题之一时抛出。   * 类的串行版本与从流中读取的类描述符的类型不匹配 * 该类包含未知的数据类型 * 该类没有可访问的无参数构造函数   （5）[OptionalDataException](mk:@MSITStore:E:\\常用API\\JDK%20API%201.8.CHM::/java/io/../../java/io/OptionalDataException.html" \o "class in java.io) - 在流中找到原始数据，而不是对象。 |
| Serializable对象解序列化的时候都发生了什么事？ | * 对象从Stream中读出来； * Java虚拟机通过存储的信息判断出对象的Class类型. * Java虚拟机尝试寻找和加载对象的类。如果Java虚拟机找不到或无法加载该类,则Java虚拟机会抛出ClassNotFoundException * 新的对象会被配置在堆上,但构造函数不会执行!很明显的,这样会把对象的状态抹去又变成全新的,而这不是我们想要的结果。我们需要的是对象回到存储时的状态。 * 如果对象在继承树上有个不可序列化的祖先类，则该不可序列化的类以及在它之上的类的构造函数（就算是可序列化也一样）就会执行。一旦构造函数连锁启动之后将无法停止。也就是说，从第一个不可序列化的父类开始，全部都会重新初始状态。 * 对象的实例变量会被还原成序列化时点的状态值。Transient变量会被赋值null的对象引用或primitive主数据类型的默认为0、false等值。 |
| 静态变量会被序列化吗？ | 并不会，要知道static代表“每个类一个”而不是“每个对象一个”。当对象被还原时，静态变量会维持类中原本的样子，而不是存储时的样子。所以如果需要对static状态保存，那么需要一些额外的操作。 |
| 为什么不将类存储称为对象的一部分呢？这样ClassNotFoundException不就不会出现了吗？ | 这在原则上是没有问题的，可以这样设计。但这样是非常的浪费，并且会有额外的工作。虽然把对象的序列化写在本机的磁盘上并不是什么很困难的工作，但序列化也有将对象送到网络联机上的用途。如果每个序列化对象都带有类，带宽的消耗可能就是大问题。  对于通过网络传送序列化来说，事实上是有一种机制可以让类使用URL来指定位置。该机制用在Java的RMI。让你可以将序列化的对象当做参数的一部分来传递，若接收此调用的Java虚拟机没有这个类的话，它可以自动地使用URL来取回并加载类。 |

## 10.2 Hessian序列化

前面大篇幅的介绍了Java原生的序列化机制,我们需要注意的是Java原生序列化机制中默认处理机制在反序列化时是不会调用类的无参构造方法,而是调用native方法将成员变量赋值为对应类型的初始值.基于性能及兼容性考虑（Java不能跨语言）,不推荐使用Java序列化。这节我们来认识一下另一种序列化方法Hessian。

### 10.2.1 Hessian概述

Hessian是一个轻量级的remoting onhttp工具，使用简单的方法提供了RMI的功能。 相比WebService，Hessian更简单、快捷。采用的是二进制RPC协议，因为采用的是二进制协议，所以它很适合于发送二进制数据。

Hessian序列化是一种支持动态类型，跨语言、基于对象传输的网络协议。Java对象序列化的二进制流可以被其它语言(如C++、Python)反序列化。Hessian协议具有如下特性：

* 自描述序列化类型。不依赖外部描述文件或接口定义，用一个字节表示常用基础类型，极大缩短二进制流。
* 语言无关，支持脚本语言。
* 协议简单，比Java原生序列化高效。

相比Hessian1.0，Hessian2.0中增加了压缩编码，其序列化二进制流大小是Java序列化的50%，序列化耗时是Java序列化的30%，反序列化耗时是Java反序列化的20%。

Hessian会把复杂对象所有属性存储在一个Map中进行序列化。所以在父类、子类存在同名成员变量的情况下，Hessian序列化时，先序列化子类，然后序列化父类，因此反序列化结果会导致子类同名成员变量被父类的值覆盖。

### 10.2.2 Hessian序列化的代码实现

在开始代码前，我们必须说明的一点是，Hessian是一种性能优于Java原生序列化的序列化方式,但是它仍建立在Java序列化规范上,即一个实例如果想要序列化,那么该实例对应的类就必须实现Serializable接口,这是一种标识,标识许可该类实例进行序列化。Transient关键字仍旧起到标识忽略字段的序列化。下面简单看一个使用Hessian进行的序列化：

|  |
| --- |
| **public** **class** Employee **implements** Serializable{  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** **int** employeeId;  **private** String employeeName;  /\*\*  \* 使用transient关键字，表示该字段不序列化，所以最终反序列化后这个字段为null  \*/  **private** **transient** String department;    **public** **int** getEmployeeId() {  **return** employeeId;  }  **public** **void** setEmployeeId(**int** employeeId) {  **this**.employeeId = employeeId;  }  **public** String getEmployeeName() {  **return** employeeName;  }  **public** **void** setEmployeeName(String employeeName) {  **this**.employeeName = employeeName;  }  **public** String getDepartment() {  **return** department;  }  **public** **void** setDepartment(String department) {  **this**.department = department;  }  @Override  **public** String toString() {  **return** "Employee{" +  "employeeId=" + employeeId +  ", employeeName='" + employeeName + '\'' +  ", department='" + department + '\'' +  '}';  }  } |
| **public** **class** HessianTest {  **public** **static** **void** main(String[] args) **throws** IOException {  Employee employee = **new** Employee();  employee.setEmployeeId(90);  employee.setEmployeeName("Yorick");  employee.setDepartment("序列化");  **byte**[] bytes = *serialize*(employee);  Employee employee2 = (Employee) *deSerialize*(bytes);  **if**(employee != employee2) {  System.***out***.println(employee);  System.***out***.println(employee2);  }  }  //Hessian实现序列化  **public** **static** **byte**[] serialize(Object object) **throws** IOException {  ByteArrayOutputStream bos = **new** ByteArrayOutputStream();  HessianOutput hop = **new** HessianOutput(bos);  hop.writeObject(object);  **return** bos.toByteArray();  }  //Hessian实现反序列化  **public** **static** Object deSerialize(**byte**[] objectBytes) **throws** IOException {  ByteArrayInputStream bis = **new** ByteArrayInputStream(objectBytes);  HessianInput his = **new** HessianInput(bis);  **return** his.readObject();  }  } |
| Employee{employeeId=90, employeeName='Yorick', department='序列化'}  Employee{employeeId=90, employeeName='Yorick', department='null'} |

从上面Hessian的一个小例子我们发现，它和Serializable的操作是极其类似的，我们对于学习是毫无障碍的，但我们也不能大意，就像Serializable中一不小心会抛出异常一样，Hessian也有一些需要我们注意的内容，下面我们就来认识一下；

### 10.2.3 Hessian误区

Hessian会把复杂对象所有属性存储在一个Map中进行序列化。所以在父类、子类存在同名成员变量的情况下，Hessian序列化时，先序列化子类，然后序列化父类，因此反序列化结果会导致子类同名成员变量被父类的值覆盖。

|  |
| --- |
| **public** **class** User **implements** Serializable {  **private** String username ;  **private** String password;  **private** Integer age;  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  **public** Integer getAge() {  **return** age;  }  **public** **void** setAge(Integer age) {  **this**.age = age;  }  @Override  **public** String toString() {  **return** getClass().getName() + "\t\_" + username + "\_" + password +  "\_" + age;  }  } |
| **public** **class** UserInfo **extends** User {  **private** String username;  **public** **static** **void** main(String[] args) **throws** IOException {  UserInfo user = **new** UserInfo();  user.setUsername("hello world");  user.setPassword("buzhidao");  user.setAge(21);  **byte**[] bytes = HessianTest.*serialize*(user);  UserInfo info = (UserInfo) HessianTest.*deSerialize*(bytes);  System.***out***.println(user);  System.***out***.println(info);  }  } |
| cn.yorick.hessian.UserInfo \_hello world\_buzhidao\_21  cn.yorick.hessian.UserInfo \_null\_buzhidao\_21 |

我们下面了解一下为什么会出现这种结果：

* hessian序列化的时候会取出对象的所有自定义属性，相同类型的属性是子类在前父类在后的顺序。
* hessian在反序列化的时候，是将对象所有属性取出来，存放在一个map中   key = 属性名  value是反序列类，相同名字的会以子类为准进行反序列化。
* 相同名字的属性 在反序列化的是时候，由于子类在父类前面，子类的属性总是会被父类的覆盖，由于java多态属性，在上述例子中父类 User.username = null

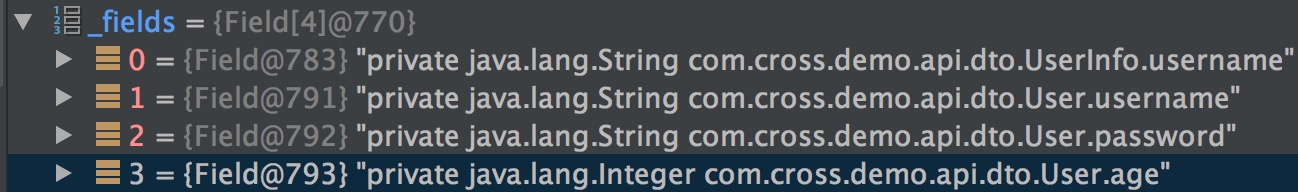
下面是关键源码分析 ，hessian版本是4.0.7

     当序列化对象是一个java自定对象时，默认的序列化类是 UnsafeSerializer，调用writeObject

|  |
| --- |
| public void writeObject(Object obj, AbstractHessianOutput out)  throws IOException{  if (out.addRef(obj)) {  return;  }  Class<?> cl = obj.getClass()；  int ref = out.writeObjectBegin(cl.getName());  if (ref >= 0) {  writeInstance(obj, out);  }  else if (ref == -1) {  writeDefinition20(out);  out.writeObjectBegin(cl.getName());  writeInstance(obj, out);  }  else {  writeObject10(obj, out);  }  } |

以上代码会调用 writeObject10(obj, out);

|  |
| --- |
| protected void writeObject10(Object obj, AbstractHessianOutput out)  throws IOException{  for (int i = 0; i < \_fields.length; i++) {  Field field = \_fields[i];  out.writeString(field.getName());  \_fieldSerializers[i].serialize(out, obj);  }  out.writeMapEnd();  } |



反序列化的时候，当反序列化时，默认的反序列化类是 UnsafeSerializer，会首先根据反序列化类型，创建一个map

|  |
| --- |
| protected HashMap<String,FieldDeserializer> getFieldMap(Class<?> cl){  HashMap<String,FieldDeserializer> fieldMap  = new HashMap<String,FieldDeserializer>();  for (; cl != null; cl = cl.getSuperclass()) {  Field []fields = cl.getDeclaredFields();  for (int i = 0; i < fields.length; i++) {  Field field = fields[i];  if (Modifier.isTransient(field.getModifiers())  || Modifier.isStatic(field.getModifiers()))  continue;  else if (fieldMap.get(field.getName()) != null) //相同名字的会以子类为准进行序列化  continue;  // XXX: could parameterize the handler to only deal with public  try {  field.setAccessible(true);  } catch (Throwable e) {  e.printStackTrace();  }  Class<?> type = field.getType();  FieldDeserializer deser;  if (String.class.equals(type)) {  deser = new StringFieldDeserializer(field);  }  else if (byte.class.equals(type)) {  deser = new ByteFieldDeserializer(field);  }  。。。。。。  fieldMap.put(field.getName(), deser);  }  }  return fieldMap;  } |

 如果是String类型的属性，使用的是StringFieldDeserializer

|  |
| --- |
| StringFieldDeserializer(Field field) {  \_field = field;  \_offset = \_unsafe.objectFieldOffset(\_field); //这个会把属性对象对象的偏移量设置好  } |

接下来会对每个属性用map对应序列化方式进行反序列化和赋值

|  |
| --- |
| public Object readMap(AbstractHessianInput in, Object obj)  throws IOException{  try {  int ref = in.addRef(obj);  while (! in.isEnd()) {  Object key = in.readObject();  FieldDeserializer deser = (FieldDeserializer) \_fieldMap.get(key);  if (deser != null)  deser.deserialize(in, obj);  else  in.readObject();  }  in.readMapEnd();  Object resolve = resolve(in, obj);  if (obj != resolve)  in.setRef(ref, resolve);  return resolve;  } catch (IOException e) {  throw e;  } catch (Exception e) {  throw new IOExceptionWrapper(e);  }  } |

这个是StringFieldDeserializer 反序列化，由于名字相同的属性，反序列化是第一个子类，往后父类的发现map中有就会忽略，所以在属性序列化的时候，先序列化子类的，接着是父类的，但是他们在对象中的偏移量是一样的（用的是同一个反序列化类），所以相同名字的属相，子类总是会被父类覆盖掉。

|  |
| --- |
| @SuppressWarnings("restriction")  void deserialize(AbstractHessianInput in, Object obj)  throws IOException{  String value = null;  try {  value = in.readString();  \_unsafe.putObject(obj, \_offset, value);  } catch (Exception e) {  logDeserializeError(\_field, obj, value, e);  }  } |

总结：使用hessian序列化时，一定要注意子类和父类不能有同名字段

## 10.3 JSON序列化

JSON（javaScript Object Notation ）是一种基于JavaScript语言的轻量级的数据交换格式。主要采用键值对({"name": "json"})的方式来保存和表示数据。JSON是JS对象的字符串表示法，它使用文本表示一个JS对象的信息，本质上是一个字符串。

JSON序列化就是将数据对象转换为JSON字符串。在序列化过程中抛弃了类型信息，所以反序列化时只是提供类型信息才能准确地反序列化。相比原生序列化和Hessian，JSON可读性好，方便调试。

JSON用于序列化和反序列化的方法有很多，我们下面来认识几个常见的；

### 10.3.1 fastjson

FastJson是阿里的开源框架,被不少企业使用,是一个极其优秀的Json框架。FastJson具有如下特点：

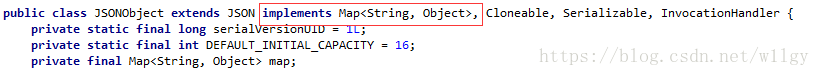
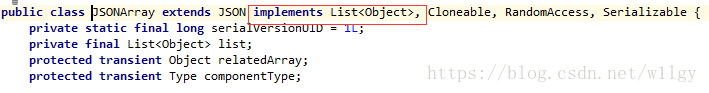
* FastJson速度快,无论序列化和反序列化,都是当之无愧的fast ,测试表明，fastjson具有极快的性能，超越任其他的java json parser。
* 功能强大(支持普通JDK类包括任意Java Bean Class、Collection、Map、Date或enum)
* 零依赖(能够直接运行在Java SE 5.0以上版本)

下面我们就针对Fastjson来做一下简单的介绍和使用说明:

#### 10.3.1.1 FastJson的简单说明:

FastJson对于json格式字符串的解析主要用到了下面三个类：

* JSON：fastJson的解析器，用于JSON格式字符串与JSON对象及javaBean之间的转换
* JSONObject：fastJson提供的json对象
* JSONArray：fastJson提供json数组对象

我们可以把JSONObject当成一个Map<String,Object>来看，只是JSONObject提供了更为丰富便捷的方法，方便我们对于对象属性的操作。我们看一下源码。   
  
同样我们可以把JSONArray当做一个List，可以把JSONArray看成JSONObject对象的一个集合。   
  此外，由于JSONObject和JSONArray继承了JSON，所以说也可以直接使用两者对JSON格式字符串与JSON对象及javaBean之间做转换，不过为了避免混淆我们还是使用JSON。 下面是一些方法的介绍。

|  |
| --- |
| public static final Object parse(String text); // 把JSON文本parse为JSONObject或者JSONArray  public static final JSONObject parseObject(String text)； // 把JSON文本parse成JSONObject  public static final <T> T parseObject(String text, Class<T> clazz); // 把JSON文本parse为JavaBean  public static final JSONArray parseArray(String text); // 把JSON文本parse成JSONArray  public static final <T> List<T> parseArray(String text, Class<T> clazz); //把JSON文本parse成JavaBean集合  public static final String toJSONString(Object object); // 将JavaBean序列化为JSON文本  public static final String toJSONString(Object object, boolean prettyFormat); // 将JavaBean序列化为带格式的JSON文本  public static final Object toJSON(Object javaObject); //将JavaBean转换为JSONObject或者JSONArray。 |

#### 10.3.1.2 FastJson的用法

首先定义三个Json格式的字符串:JSONTest

|  |
| --- |
| //JSON字符串简单对象  **private** **static** **final** String ***JSON\_OBJ\_STR*** = "{\"studentName\":\"lily\",\"studentAge\""  + ":12}";  //JSON字符串数组类型  **private** **static** **final** String ***JSON\_ARRAY\_STR*** = "[{\"studentName\":\"lily\",\"studengAge\":12},"  + "{\"studentName\":\"licy\",\"studentAge\":15}]";  //复杂格式JSON字符串  **private** **static** **final** String ***COMPLEX\_JSON\_STR*** = "{\"teacherName\":\"crystall\",\"teacherAge\":27,"  + "\"course\":{\"courseName\":\"English\",\"code\":1270},\"students\":"  + "[{\"studentName\":\"lily\",\"studentAge\":12},{\"studentName\":\"lucy\",\"studentAge\":15}]}"; |

#### 10.3.1.3 JSON格式字符串与JSON对象之间的转换

如果想将一个对象持久化传输,那么首先将对象转变为JSON字符串是一个必不可缺的步骤,下面我们以上面三种JSON格式字符串为例来看一下相互之间的转变:

|  |
| --- |
| @Test  **public** **void** jsonObjectAndJsonStringTest() {  //JSON字符串转变为JSON对象  JSONObject jsonObject = JSONObject.*parseObject*(***JSON\_OBJ\_STR***);  System.***out***.println("studentName:" + jsonObject.getString("studentName"));  //从JSON对象到JSON字符串  String jsonString = jsonObject.toJSONString();  String jsonStringWithParameter = jsonObject.*toJSONString*(jsonObject);  System.***out***.println("从JSONObject到JSON字符串:" + jsonString);  System.***out***.println(jsonStringWithParameter);  } |
| studentName:lily  从JSONObject到JSON字符串:{"studentAge":12,"studentName":"lily"}  {"studentAge":12,"studentName":"lily"} |
| @Test  **public** **void** jsonObjectAndJsonArrayTest() {  //JSON字符串转变为JSON数组  JSONArray jsonArray = JSONObject.*parseArray*(***JSON\_ARRAY\_STR***);  **int** size = jsonArray.size();  JSONObject jsonObject;  **for** (**int** i = 0; i < size; i++) {  jsonObject = jsonArray.getJSONObject(i);  System.***out***.println(jsonObject.toString());  }  //另一种遍历方式  **for** (Object object : jsonArray) {  jsonObject = (JSONObject) object;  System.***out***.println(jsonObject.getInteger("studentAge"));  }  //将JSONArray转变为JSON字符串数组  String jsonArrayString = JSONArray.*toJSONString*(jsonArray);  System.***out***.println(jsonArrayString);  } |
| {"studentAge":12,"studentName":"lily"}  {"studentAge":15,"studentName":"licy"}  12  15  [{"studentAge":12,"studentName":"lily"},{"studentAge":15,"studentName":"licy"}] |
| @Test  **public** **void** complexJSONObjectAndJSONStringTest() {  //复杂JSON字符串转变为JSON对象  JSONObject jsonObject = JSONObject.*parseObject*(***COMPLEX\_JSON\_STR***);  System.***out***.println(jsonObject.toJSONString());  //下面不会成功,会报出异常java.lang.ClassCastException: com.alibaba.fastjson.JSONArray cannot be cast to com.alibaba.fastjson.JSONObject  // JSONObject array = jsonObject.getJSONObject("course");  /\*JSONArray array = jsonObject.getJSONArray("students");  System.out.println(array.toString());空指针异常\*/  JSONArray jsonArray = jsonObject.getJSONArray("course");  String course1 = jsonArray.getString(0);  System.***out***.println(course1);  // jsonArray = jsonArray.getJSONArray(1); IndexOutOfBoundsException  jsonObject = jsonArray.getJSONObject(0);  jsonArray = jsonObject.getJSONArray("students");  //JSON对象不管是普通Object对象还是数组,通过toString都可以得到一个JSOnString对象  System.***out***.println(jsonArray.toJSONString());  } |
| {"teacherAge":27,"teacherName":"crystall","course":[{"courseName":"English","code":1270,"students":[{"studentAge":12,"studentName":"lily"},{"studentAge":15,"studentName":"lucy"}]}]}  {"courseName":"English","code":1270,"students":[{"studentAge":12,"studentName":"lily"},{"studentAge":15,"studentName":"lucy"}]}  [{"studentAge":12,"studentName":"lily"},{"studentAge":15,"studentName":"lucy"}] |

#### 10.3.1.4 JSON格式字符串与javaBean之间的转换

JSON格式字符串和JSON对象之间的转变是很简单的,但是我们在实际开发中,往往会使用一个JavaBean来映射数据,那么JavaBean若想序列化传输是不是也应该先转换为一个JSON字符串呢?下面我们来认识一下,先要说明的是,我们并不能真正实现一个从无到有的过程,即我们要先有一个JavaBean类。

|  |
| --- |
| **public** **class** Student {  **private** String studentName;  **private** String studentAge;  **public** Student(String studentName, String studentAge) {  **super**();  **this**.studentName = studentName;  **this**.studentAge = studentAge;  }  **public** String getStudentName() {  **return** studentName;  }  **public** **void** setStudentName(String studentName) {  **this**.studentName = studentName;  }  **public** String getStudentAge() {  **return** studentAge;  }  **public** **void** setStudentAge(String studentAge) {  **this**.studentAge = studentAge;  }  @Override  **public** String toString() {  **return** studentName + ":" + studentAge;  }  } |
| //json字符串到JavaBean对象的互相转换--前提是我们要有一个JavaBean类  @Test  **public** **void** jsonStringToJavaBeanTest() {  JSONObject jsonObject = JSONObject.*parseObject*(***JSON\_OBJ\_STR***);  //第一种方式  String studentName = jsonObject.getString("studentName");  String studentAge = jsonObject.getString("studentAge");  Student student = **new** Student(studentName, studentAge);  System.***out***.println(student);  //第二种方式:使用TypeReference<T>类,由于其构造方法使用protected进行修饰,故创建其子类,这里要注意fastjson版本  Student student2 = JSON.*parseObject*(***JSON\_OBJ\_STR***, **new** TypeReference<Student>() {});  // jsonObject.parseObject(JSON\_OBJ\_STR, new TypeReference<Student>() {}); 等同上句  System.***out***.println(student2);  //第三种方式,使用Gson思想  Student student3 = JSON.*parseObject*(***JSON\_OBJ\_STR***, Student.**class**);  System.***out***.println(student3);  }  @Test  **public** **void** JavaBeanToJsonStringTest() {  Student student = **new** Student("Yorick", "20");  String jsonString = JSONObject.*toJSONString*(student);  System.***out***.println(jsonString);  JSONObject jsonObject = (JSONObject) JSON.*toJSON*(student);//JavaBean转变为JSONObject或JSONArray  System.***out***.println(jsonObject.toString());  } |
| lily:12  lily:12  lily:12 JSONString到JavaBean的转变  {"studentAge":"20","studentName":"Yorick"}  {"studentAge":"20","studentName":"Yorick"} |
| 对于TypeReference，由于其构造方法使用 protected 进行修饰，所以在其他包下创建其对象的时候，要用其实现类的子类：new TypeReference() {}  **public** **abstract** **class** TypeReference<T> {  **private** **final** Type type;  **private** **volatile** Constructor<?> constructor;  **protected** TypeReference() {  Type superclass = getClass().getGenericSuperclass();  **if** (superclass **instanceof** Class) {  **throw** **new** RuntimeException("Missing type parameter.");  }  **this**.type = ((ParameterizedType) superclass).getActualTypeArguments()[0];  } |
| //JSONArrayString到JavaBean的转变  @Test  **public** **void** jsonArrayStringToJavaBeanTest() {  //第一种方式  JSONArray jsonArray = JSONArray.*parseArray*(***JSON\_ARRAY\_STR***);  List<Student> list = **new** ArrayList<>();  Student student = **null**;  **for** (Object object : jsonArray) {  // student = (Student) object; ClassCastException JSONObject cannot be cast to student  JSONObject jsonObject = (JSONObject) object;  String studentName = jsonObject.getString("studentName");  **int** studentAge = jsonObject.getInteger("studentAge");  student = **new** Student(studentName, String.*valueOf*(studentAge));  list.add(student);  }  System.***out***.println("students:" + list);  //第二种方式使用TypeReference<T>类  List<Student> list2 = JSONArray.*parseObject*(***JSON\_ARRAY\_STR***, **new** TypeReference<ArrayList<Student>>() {});  System.***out***.println("students: " + list2);  //第三种使用Gson的思想  List<Student> list3 = JSONArray.*parseArray*(***JSON\_ARRAY\_STR***, Student.**class**);  System.***out***.println("students: " + list3);  } |
| students:[lily:12, licy:15]  students: [lily:12, licy:15]  students: [lily:12, licy:15] |
| //JavaBeanList到json数组的转换  @Test  **public** **void** javaBeanListToJsonArrayTest() {  Student student1 = **new** Student("lily", "15");  Student student2 = **new** Student("Lucy", "16");  List<Student> list = **new** ArrayList<>();  list.add(student1);  list.add(student2);  String jsonArrayString = JSONArray.*toJSONString*(list);//若list中没有元素,那么返回[]  System.***out***.println(jsonArrayString);  } |
| [{"studentAge":"15","studentName":"lily"},{"studentAge":"16","studentName":"Lucy"}] |
| **public** **class** Course {  **private** String courseName;  **private** Integer code;  **public** String getCourseName() {  **return** courseName;  }  **public** **void** setCourseName(String courseName) {  **this**.courseName = courseName;  }  **public** Integer getCode() {  **return** code;  }  **public** **void** setCode(Integer code) {  **this**.code = code;  }  @Override  **public** String toString() {  **return** courseName + ":" + code;  }  } |
| **public** **class** Teacher {  **private** String teacherName;  **private** Integer teacherAge;  **private** Course course;  **private** List<Student> students;  **public** String getTeacherName() {  **return** teacherName;  }  **public** **void** setTeacherName(String teacherName) {  **this**.teacherName = teacherName;  }  **public** Integer getTeacherAge() {  **return** teacherAge;  }  **public** **void** setTeacherAge(Integer teacherAge) {  **this**.teacherAge = teacherAge;  }  **public** Course getCourse() {  **return** course;  }  **public** **void** setCourse(Course course) {  **this**.course = course;  }  **public** List<Student> getStudents() {  **return** students;  }  **public** **void** setStudents(List<Student> students) {  **this**.students = students;  }  @Override  **public** String toString() {  **return** teacherName + ":" + teacherAge + "," + course + students;  }  } |
| //复杂的JSON字符串到JavaBean的转变  @Test  **public** **void** ComplexJsonStringToJavaBeanTest() {  //第一种方式,使用TypeReference<T>类,由于其构造方法使用protected进行修饰,故创建其子类  Teacher teacher = JSONObject.*parseObject*(***COMPLEX\_JSON\_STR***, **new** TypeReference<Teacher>() {});  String teacherName = teacher.getTeacherName();  String teacherAge = teacher.getTeacherAge() + "";  Course course = teacher.getCourse();  List<Student> students = teacher.getStudents();  System.***out***.println(***COMPLEX\_JSON\_STR***);  System.***out***.println(teacherName + ":" + teacherAge + course + students);  System.***out***.println(teacher);  //第二种方式使用Gson思想  Teacher teacher2 = JSON.*parseObject*(***COMPLEX\_JSON\_STR***, Teacher.**class**);  System.***out***.println(teacher);  } |
| {"teacherName":"crystall","teacherAge":27,"course":{"courseName":"English","code":1270},"students":[{"studentName":"lily","studentAge":12},{"studentName":"lucy","studentAge":15}]}  crystall:27English:1270[lily:12, lucy:15]  crystall:27,English:1270[lily:12, lucy:15]  crystall:27,English:1270[lily:12, lucy:15] |
| //复杂的JavaBean到JSON字符串的转变  @Test  **public** **void** ComplexToJsonStringTest() {  Teacher teacher = JSON.*parseObject*(***COMPLEX\_JSON\_STR***, Teacher.**class**);  String jsonTeacherString = JSON.*toJSONString*(teacher);  System.***out***.println(teacher);  System.***out***.println(jsonTeacherString);  } |
| crystall:27,English:1270[lily:12, lucy:15]  {"course":{"code":1270,"courseName":"English"},"students":[{"studentAge":"12","studentName":"lily"},{"studentAge":"15","studentName":"lucy"}],"teacherAge":27,"teacherName":"crystall"} |
| //进行JavaBean和JSONObject之间的转变  @Test  **public** **void** javaBeanAndJSONObjectTest() {  Student student = JSON.*parseObject*(***JSON\_OBJ\_STR***, Student.**class**);  List<Student> students = JSON.*parseArray*(***JSON\_ARRAY\_STR***, Student.**class**);  Teacher teacher = JSON.*parseObject*(***COMPLEX\_JSON\_STR***, Teacher.**class**);  //方式一就是先将JavaBean转变为对应的字符串,然后再转变为JSONObject  String studentJSON = JSON.*toJSONString*(student);  String studentsJSON = JSON.*toJSONString*(students);  String teacherJSON = JSON.*toJSONString*(teacher);  JSONObject studentJSONObject = JSONObject.*parseObject*(studentJSON);  // JSONObject studentsJSONObject = JSONObject.parseObject(studentsJSON);注意数组只能转变为JSONArray  JSONArray studentsJOSNArray = JSONArray.*parseArray*(studentsJSON);  JSONObject teacherJSONObject = JSONObject.*parseObject*(teacherJSON);  System.***out***.println(studentJSONObject);  System.***out***.println(studentsJOSNArray);  System.***out***.println(teacherJSONObject);  //方式二直接向JOSN转变,然后强转  studentJSONObject = (JSONObject) JSONObject.*toJSON*(student);  studentsJOSNArray = (JSONArray) JSONArray.*toJSON*(students);  teacherJSONObject = (JSONObject) JSONObject.*toJSON*(teacher);  System.***out***.println(studentJSONObject);  System.***out***.println(studentsJOSNArray);  System.***out***.println(teacherJSONObject);  } |
| {"studentAge":"12","studentName":"lily"}  [{"studentAge":"12","studentName":"lily"},{"studentAge":"15","studentName":"licy"}]  {"teacherAge":27,"teacherName":"crystall","course":{"courseName":"English","code":1270},"students":[{"studentAge":"12","studentName":"lily"},{"studentAge":"15","studentName":"lucy"}]}  {"studentAge":"12","studentName":"lily"}  [{"studentAge":"12","studentName":"lily"},{"studentAge":"15","studentName":"licy"}]  {"teacherAge":27,"teacherName":"crystall","course":{"courseName":"English","code":1270},"students":[{"studentAge":"12","studentName":"lily"},{"studentAge":"15","studentName":"lucy"}]} |
| //将JSONObject转变为JavaBean  @Test  **public** **void** jsonObjectToJavaBeanTest() {  JSONObject studentJSONObject = JSONObject.*parseObject*(***JSON\_OBJ\_STR***);  JSONArray studentsJOSNArray = JSONArray.*parseArray*(***JSON\_ARRAY\_STR***);  JSONObject teacherJSONObject = JSONObject.*parseObject*(***COMPLEX\_JSON\_STR***);  //将JSONObject抓变为JAvaBean和将json字符串转变为JavaBean相同,都有三种方式,我们下面来介绍一下  Student student = JSONObject.*parseObject*(studentJSONObject.toJSONString(), **new** TypeReference<Student>() {});  Student student2 = JSONObject.*parseObject*(studentJSONObject.toJSONString(), Student.**class**);  List<Student> list = JSONArray.*parseObject*(studentsJOSNArray.toJSONString(), **new** TypeReference<ArrayList<Student>>() {});  List<Student> list2 = JSONArray.*parseArray*(studentsJOSNArray.toJSONString(),Student.**class**);  System.***out***.println(student);  System.***out***.println(student2);  System.***out***.println(list);  System.***out***.println(list2);  } |
| lily:12  lily:12  [lily:12, licy:15]  [lily:12, licy:15] |

#### 10.3.1.5 fastjson总结

上面针对各种对象之间的转变进行了探讨,至于一些更深入的了解,以后在工作中会逐渐认识添加。总的来说：

* 对于JSON对象与JSON格式字符串的转换可以直接用 toJSONString()这个方法。
* javaBean与JSON格式字符串之间的转换要用到：JSON.toJSONString(obj);
* javaBean与json对象间的转换使用：JSON.toJSON(obj)，然后使用强制类型转换，JSONObject或者JSONArray。

## 10.4 序列化总结

序列化通常会通过网络传输对象，而对象中往往有敏感数据，所以序列化常常成为黑客的攻击点，攻击者巧妙地利用反序列化过程构造恶意代码，使得程序在反序列化的过程中执行任意代码。Java工程中广泛使用的Apache Commons Collections、Jackson、fastjson等都出现过反序列化漏洞。如何防范这种黑客攻击呢？有些对象不敏感属性不需要进行序列化传输，可以加transient关键字，避免把此属性信息转化为序列化的二进制流。如果一定要传递对象的敏感属性，可以使用对称和非对称加密方式独立传输，再使用某个方法把属性还原到对象中。应用开发者对序列化要有一定的安全防范意识，对传入数据的内容进行校验和权限控制，及时更新安全漏洞，避免受到攻击。