数年前，当和一个软件团队一起用 Java 语言编写一个应用程序时，我体会到比一般程序员多知道一点关于 Java 对象序列化的知识所带来的好处。

## 关于本系列

您觉得自己懂 Java 编程？事实上，大多数程序员对于 Java 平台都是浅尝则止，只学习了足以完成手头上任务的知识而已。在本 [系列](http://www.ibm.com/developerworks/cn/views/java/libraryview.jsp?search_by=%E6%82%A8%E4%B8%8D%E7%9F%A5%E9%81%93%E7%9A%84+5+%E4%BB%B6%E4%BA%8B" \t "http://www.importnew.com/_blank) 中，Ted Neward 深入挖掘 Java 平台的核心功能，揭示一些鲜为人知的事实，帮助您解决最棘手的编程挑战。

大约一年前，一个负责管理应用程序所有用户设置的开发人员，决定将用户设置存储在一个 Hashtable中，然后将这个 Hashtable 序列化到磁盘，以便持久化。当用户更改设置时，便重新将 Hashtable 写到磁盘。

这是一个优雅的、开放式的设置系统，但是，当团队决定从 Hashtable 迁移到 Java Collections 库中的HashMap 时，这个系统便面临崩溃。

Hashtable 和 HashMap 在磁盘上的格式是不相同、不兼容的。除非对每个持久化的用户设置运行某种类型的数据转换实用程序（极其庞大的任务），否则以后似乎只能一直用Hashtable 作为应用程序的存储格式。

团队感到陷入僵局，但这只是因为他们不知道关于 Java 序列化的一个重要事实：Java 序列化允许随着时间的推移而改变类型。当我向他们展示如何自动进行序列化替换后，他们终于按计划完成了向 HashMap 的转变。

本文是本系列的第一篇文章，这个系列专门揭示关于 Java 平台的一些有用的小知识 — 这些小知识不易理解，但对于解决 Java 编程挑战迟早有用。

将 Java 对象序列化 API 作为开端是一个不错的选择，因为它从一开始就存在于 JDK 1.1 中。本文介绍的关于序列化的 5 件事情将说服您重新审视那些标准 Java API。

## Java 序列化简介

Java 对象序列化是 JDK 1.1 中引入的一组开创性特性之一，用于作为一种将 Java 对象的状态转换为字节数组，以便存储或传输的机制，以后，仍可以将字节数组转换回 Java 对象原有的状态。

实际上，序列化的思想是 “冻结” 对象状态，传输对象状态（写到磁盘、通过网络传输等等），然后 “解冻” 状态，重新获得可用的 Java 对象。所有这些事情的发生有点像是魔术，这要归功于 ObjectInputStream/ObjectOutputStream 类、完全保真的元数据以及程序员愿意用Serializable 标识接口标记他们的类，从而 “参与” 这个过程。

清单 1 显示一个实现 Serializable 的 Person 类。

##### 清单 1. Serializable Person

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | **package** com.tedneward;    **public** **class** Person  **implements** java.io.Serializable  {  **public** Person(String fn, String ln, **int** a)      {  **this**.firstName = fn; **this**.lastName = ln; **this**.age = a;      }    **public** String getFirstName() { **return** firstName; }  **public** String getLastName() { **return** lastName; }  **public** **int** getAge() { **return** age; }  **public** Person getSpouse() { **return** spouse; }    **public** **void** setFirstName(String value) { firstName = value; }  **public** **void** setLastName(String value) { lastName = value; }  **public** **void** setAge(**int** value) { age = value; }  **public** **void** setSpouse(Person value) { spouse = value; }    **public** String toString()      {  **return** "[Person: firstName=" + firstName +              " lastName=" + lastName +              " age=" + age +              " spouse=" + spouse.getFirstName() +              "]";      }    **private** String firstName;  **private** String lastName;  **private** **int** age;  **private** Person spouse;    } |

将 Person 序列化后，很容易将对象状态写到磁盘，然后重新读出它，下面的 JUnit 4 单元测试对此做了演示。

##### 清单 2. 对 Person 进行反序列化

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | **public** **class** SerTest  {      @Test **public** **void** serializeToDisk()      {  **try**          {              com.tedneward.Person ted = **new** com.tedneward.Person("Ted", "Neward", 39);              com.tedneward.Person charl = **new** com.tedneward.Person("Charlotte",                  "Neward", 38);                ted.setSpouse(charl); charl.setSpouse(ted);                FileOutputStream fos = **new** FileOutputStream("tempdata.ser");              ObjectOutputStream oos = **new** ObjectOutputStream(fos);              oos.writeObject(ted);              oos.close();          }  **catch** (Exception ex)          {              fail("Exception thrown during test: " + ex.toString());          }    **try**          {              FileInputStream fis = **new** FileInputStream("tempdata.ser");              ObjectInputStream ois = **new** ObjectInputStream(fis);              com.tedneward.Person ted = (com.tedneward.Person) ois.readObject();              ois.close();                assertEquals(ted.getFirstName(）， "Ted");              assertEquals(ted.getSpouse().getFirstName(）， "Charlotte");                // Clean up the file  **new** File("tempdata.ser").delete();          }  **catch** (Exception ex)          {              fail("Exception thrown during test: " + ex.toString());          }      }  } |

到现在为止，还没有看到什么新鲜的或令人兴奋的事情，但是这是一个很好的出发点。我们将使用 Person 来发现您可能不 知道的关于 Java 对象序列化 的 5 件事。

## 1. 序列化允许[重构](http://www.amazon.cn/gp/product/B003BY6PLK/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&tag=importnew-23&linkCode=as2&camp=536&creative=3200&creativeASIN=B003BY6PLK" \o "重构" \t "http://www.importnew.com/_blank)

序列化允许一定数量的类变种，甚至重构之后也是如此，ObjectInputStream 仍可以很好地将其读出来。

Java Object Serialization 规范可以自动管理的关键任务是：

* 将新字段添加到类中
* 将字段从 static 改为非 static
* 将字段从 transient 改为非 transient

取决于所需的向后兼容程度，转换字段形式（从非 static 转换为 static 或从非 transient 转换为 transient）或者删除字段需要额外的消息传递。

### 重构序列化类

既然已经知道序列化允许重构，我们来看看当把新字段添加到 Person 类中时，会发生什么事情。

如清单 3 所示，PersonV2 在原先 Person 类的基础上引入一个表示性别的新字段。

##### 清单 3. 将新字段添加到序列化的 Person 中

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | **enum** Gender  {      MALE, FEMALE  }    **public** **class** Person  **implements** java.io.Serializable  {  **public** Person(String fn, String ln, **int** a, Gender g)      {  **this**.firstName = fn; **this**.lastName = ln; **this**.age = a; **this**.gender = g;      }    **public** String getFirstName() { **return** firstName; }  **public** String getLastName() { **return** lastName; }  **public** Gender getGender() { **return** gender; }  **public** **int** getAge() { **return** age; }  **public** Person getSpouse() { **return** spouse; }    **public** **void** setFirstName(String value) { firstName = value; }  **public** **void** setLastName(String value) { lastName = value; }  **public** **void** setGender(Gender value) { gender = value; }  **public** **void** setAge(**int** value) { age = value; }  **public** **void** setSpouse(Person value) { spouse = value; }    **public** String toString()      {  **return** "[Person: firstName=" + firstName +              " lastName=" + lastName +              " gender=" + gender +              " age=" + age +              " spouse=" + spouse.getFirstName() +              "]";      }    **private** String firstName;  **private** String lastName;  **private** **int** age;  **private** Person spouse;  **private** Gender gender;  } |

序列化使用一个 hash，该 hash 是根据给定源文件中几乎所有东西 — 方法名称、字段名称、字段类型、访问修改方法等 — 计算出来的，序列化将该 hash 值与序列化流中的 hash 值相比较。

为了使 Java 运行时相信两种类型实际上是一样的，第二版和随后版本的 Person 必须与第一版有相同的序列化版本 hash（存储为 private static final serialVersionUID 字段）。因此，我们需要 serialVersionUID 字段，它是通过对原始（或 V1）版本的 Person 类运行 JDK serialver命令计算出的。

一旦有了 Person 的 serialVersionUID，不仅可以从原始对象 Person 的序列化数据创建 PersonV2 对象（当出现新字段时，新字段被设为缺省值，最常见的是“null”），还可以反过来做：即从 PersonV2 的数据通过反序列化得到 Person，这毫不奇怪。

## 2. 序列化并不安全

让 Java 开发人员诧异并感到不快的是，序列化二进制格式完全编写在文档中，并且完全可逆。实际上，只需将二进制序列化流的内容转储到控制台，就足以看清类是什么样子，以及它包含什么内容。

这对于安全性有着不良影响。例如，当通过 RMI 进行远程方法调用时，通过连接发送的对象中的任何 private 字段几乎都是以明文的方式出现在套接字流中，这显然容易招致哪怕最简单的安全问题。

幸运的是，序列化允许 “hook” 序列化过程，并在序列化之前和反序列化之后保护（或模糊化）字段数据。可以通过在 Serializable 对象上提供一个 writeObject 方法来做到这一点。

### 模糊化序列化数据

假设 Person 类中的敏感数据是 age 字段。毕竟，女士忌谈年龄。 我们可以在序列化之前模糊化该数据，将数位循环左移一位，然后在反序列化之后复位。（您可以开发更安全的算法，当前这个算法只是作为一个例子。）

为了 “hook” 序列化过程，我们将在 Person 上实现一个 writeObject 方法；为了 “hook” 反序列化过程，我们将在同一个类上实现一个readObject 方法。重要的是这两个方法的细节要正确 — 如果访问修改方法、参数或名称不同于清单 4 中的内容，那么代码将不被察觉地失败，Person 的 age 将暴露。

##### 清单 4. 模糊化序列化数据

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | **public** **class** Person  **implements** java.io.Serializable  {  **public** Person(String fn, String ln, **int** a)      {  **this**.firstName = fn; **this**.lastName = ln; **this**.age = a;      }    **public** String getFirstName() { **return** firstName; }  **public** String getLastName() { **return** lastName; }  **public** **int** getAge() { **return** age; }  **public** Person getSpouse() { **return** spouse; }    **public** **void** setFirstName(String value) { firstName = value; }  **public** **void** setLastName(String value) { lastName = value; }  **public** **void** setAge(**int** value) { age = value; }  **public** **void** setSpouse(Person value) { spouse = value; }    **private** **void** writeObject(java.io.ObjectOutputStream stream)  **throws** java.io.IOException      {          // "Encrypt"/obscure the sensitive data          age = age << 2;          stream.defaultWriteObject();      }    **private** **void** readObject(java.io.ObjectInputStream stream)  **throws** java.io.IOException, ClassNotFoundException      {          stream.defaultReadObject();            // "Decrypt"/de-obscure the sensitive data          age = age << 2;      }    **public** String toString()      {  **return** "[Person: firstName=" + firstName +              " lastName=" + lastName +              " age=" + age +              " spouse=" + (spouse!=**null** ? spouse.getFirstName() : "[null]") +              "]";      }    **private** String firstName;  **private** String lastName;  **private** **int** age;  **private** Person spouse;  } |

如果需要查看被模糊化的数据，总是可以查看序列化数据流/文件。而且，由于该格式被完全文档化，即使不能访问类本身，也仍可以读取序列化流中的内容。

## 3. 序列化的数据可以被签名和密封

上一个技巧假设您想模糊化序列化数据，而不是对其加密或者确保它不被修改。当然，通过使用 writeObject 和 readObject 可以实现密码加密和签名管理，但其实还有更好的方式。

如果需要对整个对象进行加密和签名，最简单的是将它放在一个 javax.crypto.SealedObject 和/或 java.security.SignedObject 包装器中。两者都是可序列化的，所以将对象包装在 SealedObject 中可以围绕原对象创建一种 “包装盒”。必须有对称密钥才能解密，而且密钥必须单独管理。同样，也可以将 SignedObject 用于数据验证，并且对称密钥也必须单独管理。

结合使用这两种对象，便可以轻松地对序列化数据进行密封和签名，而不必强调关于数字签名验证或加密的细节。很简洁，是吧？

## 4. 序列化允许将代理放在流中

很多情况下，类中包含一个核心数据元素，通过它可以派生或找到类中的其他字段。在此情况下，没有必要序列化整个对象。可以将字段标记为 transient，但是每当有方法访问一个字段时，类仍然必须显式地产生代码来检查它是否被初始化。

如果首要问题是序列化，那么最好指定一个 flyweight 或代理放在流中。为原始 Person 提供一个 writeReplace 方法，可以序列化不同类型的对象来代替它。类似地，如果反序列化期间发现一个 readResolve 方法，那么将调用该方法，将替代对象提供给调用者。

### 打包和解包代理

writeReplace 和 readResolve 方法使 Person 类可以将它的所有数据（或其中的核心数据）打包到一个 PersonProxy 中，将它放入到一个流中，然后在反序列化时再进行解包。

##### 清单 5. 你完整了我，我代替了你

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68 | **class** PersonProxy  **implements** java.io.Serializable  {  **public** PersonProxy(Person orig)      {          data = orig.getFirstName() + "," + orig.getLastName() + "," + orig.getAge();  **if** (orig.getSpouse() != **null**)          {              Person spouse = orig.getSpouse();              data = data + "," + spouse.getFirstName() + "," + spouse.getLastName() + ","                + spouse.getAge();          }      }    **public** String data;  **private** Object readResolve()  **throws** java.io.ObjectStreamException      {          String[] pieces = data.split(",");          Person result = **new** Person(pieces[0], pieces[1], Integer.parseInt(pieces[2]));  **if** (pieces.length > 3)          {              result.setSpouse(**new** Person(pieces[3], pieces[4], Integer.parseInt                (pieces[5])));              result.getSpouse().setSpouse(result);          }  **return** result;      }  }    **public** **class** Person  **implements** java.io.Serializable  {  **public** Person(String fn, String ln, **int** a)      {  **this**.firstName = fn; **this**.lastName = ln; **this**.age = a;      }    **public** String getFirstName() { **return** firstName; }  **public** String getLastName() { **return** lastName; }  **public** **int** getAge() { **return** age; }  **public** Person getSpouse() { **return** spouse; }    **private** Object writeReplace()  **throws** java.io.ObjectStreamException      {  **return** **new** PersonProxy(**this**);      }    **public** **void** setFirstName(String value) { firstName = value; }  **public** **void** setLastName(String value) { lastName = value; }  **public** **void** setAge(**int** value) { age = value; }  **public** **void** setSpouse(Person value) { spouse = value; }    **public** String toString()      {  **return** "[Person: firstName=" + firstName +              " lastName=" + lastName +              " age=" + age +              " spouse=" + spouse.getFirstName() +              "]";      }    **private** String firstName;  **private** String lastName;  **private** **int** age;  **private** Person spouse;  } |

注意，PersonProxy 必须跟踪 Person 的所有数据。这通常意味着代理需要是 Person 的一个内部类，以便能访问 private 字段。有时候，代理还需要追踪其他对象引用并手动序列化它们，例如 Person 的 spouse。

这种技巧是少数几种不需要读/写平衡的技巧之一。例如，一个类被重构成另一种类型后的版本可以提供一个 readResolve 方法，以便静默地将被序列化的对象转换成新类型。类似地，它可以采用 writeReplace 方法将旧类序列化成新版本。

## 5. 信任，但要验证

认为序列化流中的数据总是与最初写到流中的数据一致，这没有问题。但是，正如一位美国前总统所说的，“信任，但要验证”。

对于序列化的对象，这意味着验证字段，以确保在反序列化之后它们仍具有正确的值，“以防万一”。为此，可以实现 ObjectInputValidation接口，并覆盖 validateObject() 方法。如果调用该方法时发现某处有错误，则抛出一个 InvalidObjectException。

## 结束语

Java 对象序列化比大多数 Java 开发人员想象的更灵活，这使我们有更多的机会解决棘手的情况。

幸运的是，像这样的编程妙招在 JVM 中随处可见。关键是要知道它们，在遇到难题的时候能用上它们。

5 件事 系列下期预告：Java Collections。在此之前，好好享受按自己的想法调整序列化吧！