Java 2 如何传送非 Serializable 的 final 对象

Java 2 how to deliver not the object of final that series turn

曹大有　(湖北丹江口 郧阳师范高等专科学校 计算机科学系　442700)

摘要:在 Java 中有很多非 Serializable 的 final 的对象,在默认条件下,它们是不能通过流的方式来完成持续性的。本文则探讨了如何通过对象流的子类来完成这些类对象的 Serializable。

关键词:持续性 系列化 对象

Java 2 中有很多非 Serializable 的 final 的对象,如:DatagramPacket 类。这些类对象是不能通过正常的系列化(是将数据分解成能够在信道中传输的过程,如整数分解成 4 个字节,并写入一个 OutputStream 中,它是用于对当前对象进行编组的术语以使对象在所有的域甚至其他参考类型中能够被不加改动地传送和分解)方式进行存储和保存的,这些类对象通常是随生成该对象的程序终止而消失,当不存在到该对象的引用时,Java 虚拟机(JVM)中的自动垃圾收集器(garbage collector)就会回收分配给该对象的内存空间。我们如何保存这些非 Serializable 的 final 的对象呢?

1　非 Serializable 的 final 类对象完成持续性的可行性分析

通常情况下,没有实现 Serializable 接口的类对象中是不能被系列化的,对于非 Serializable 的类对象解决系列化的通用方式是该用类对象实现 Serializable 接口的子类来增加系列化的,但由于这些类又是 final 的,是 final 就说明不能通过子类化来增加系列化功能的。我们通过分析 Java 对象流的 API 文档发现,ObjectOutputStream 流的 writeObject(Object obj)是通过调用它 private 的方法 writeObject0(obj, false)来写对象 obj 的,而在方法 writeObject0(obj, false)的实现中有以下一段程序:

```
if (enableReplace) {
  Object rep = replaceObject(obj);
  if (rep != obj && rep != null) {
    cl = rep.getClass();
    desc = ObjectStreamClass.lookup(cl, true);
  }
  obj = rep;
}
```

同样 ObjectInputStream 流的 readObject()是通过调用它 private 的方法 readObject0(false)来读对象 obj 的,而方法 readObject0()要调用 checkResolve()方法,在方法 checkResolve()的实现中也有以下一段程序:

```
if (!enableResolve || handles.lookupException
(passHandle) != null) {
  return obj;
}
Object rep = resolveObject(obj);
if (rep != obj) {
  handles.setObject(passHandle, rep);
}
return rep;
```

这说明对这些非 Serializable 的 final 类对象 Java 可以通过间接的方式来系列化它们,即通过对象流的子类,在对象流子类中加入对象替换和对象恢复机制,就是说在系列化之前将该对象替换另一个可以系列化的对象进行传送,而在反系列化(指数据接收后的解包过程,如收到整数的 4 个字节,将这 4 个字节还原成一个整数)时再将保存的对象还原成原始的对象,这样就可以达到我们的目的。在具体过程中,除了对 Java 对象流进行子类化处理之外,我们还要在对象输出流子类及输入流子类中重载以下方法。

1.1 对象输出流子类重载

protected Object replaceObject(Object obj) throws

IOException;

protected boolean enableReplaceObject(boolean enable)

throws SecurityException;

其中:replaceObject()**方法的原型为**:

protected Object replaceObject(Object obj) throws IOException {

return obj;

}

它的功能是在传送前用另一个可以系列化的对象来替换对象这里的 Object obj,**对它我们应该在对象输出流子类中重载,以完成对象的替换工作。**enableReplaceObject()**方法的原型为**:

protected boolean enableReplaceObject(boolean enable)

throws SecurityException

{

if(enable == enableReplace){

return enable;

}

if(enable){

SecurityManager sm = System.getSecurityManager();

if(sm != null){

sm.checkPermission(SUBSTITUTION_PERMISSION);

}

}

enableReplace = enable;

return ! enableReplace;

}

在这里应重载它让它返回 true,**因为该方法引入了某种安全问题,它在初始时是不能用的。为了使对象能替换,应将它返回** true,**所需要的工作就是让参数** enable = true。

1.2 对象输入流子类中重载

protected Object resolveObject(Object obj) throws IOException;

protected boolean enableResolveObject(boolean enable)

throws SecurityException;

同样:resolveObject()**方法的原型为**:

protected Object resolveObject(Object obj) throws IOException {

return obj;

}

它的功能是用来在接收某一对象后用原始类对象替换这里的对象 Object obj,**它常常是与一个** ObjectOutputStream **流的** replaceObject()**方法一起使用的,从而达到传送那些常常不能系列化的对象,这个方法必须总是返回一个与原始类相兼容的** Object,**它应是原始类或原始类的一个子类,这里也应该重载它,以完成对象的替换工作。**enableResolveObject()**方法的原型为**:

protected boolean enableResolveObject(boolean enable)

throws SecurityException

{

if(enable == enableResolve){

return enable;

}

if(enable){

SecurityManager sm = System.getSecurityManager();

if(sm != null){

sm.checkPermission(SUBSTITUTION_PERMISSION);

}

}

enableResolve = enable;

return ! enableResolve;

}

它和 ObjectOutputStream **流的** enableReplaceObject()**方法一样,也应让它返回** true,**同样只要让参数** enable = true **就可以了。有了这些分析之后,我们便可以系列化非** Serializable **的** final **对象了,下面通过一个具体实例来做进一步的详细说明。**

2 非 Serializable 的 final 类对象系列化和反系列化的实例

下面通过系列化 DatagramPacket **类的具体实例来说明上述操作。由于它是一个非** Serializable **的** final

类,在具体系列化时由于要进行对象的替换操作,所以我们要在该类的基础之上定义一个转换类。

2.1 转换类的具体定义

```
class MyDatagramPacket implements Serializable
{
  private byte[] data;
  private int length,port;
  private InetAddress address;
  public MyDatagramPacket(DatagramPacket packet)
  {
    data = packet.getData();
    length = packet.getLength();
    address = packet.getAddress();
    port = packet.getPort();
  }
  public DatagramPacket toDatagramPacket()
  {
    return new DatagramPacket(data, length, address,port);
  }
}
```

该类的作用是:在系列化 DatagramPacket 类对象时,我们利用该类将 DatagramPacket 转换成 MyDatagramPacket 类对象,在 MyDatagramPacket 类对象中我们保存了 DatagramPacket 类对象的信息,而 MyDatagramPacket 类对象是可系列化的。而在反系列化时,我们利用 MyDatagramPacket 类对象方法 toDatagramPacket() 将 MyDatagramPacket 类对象还原成 DatagramPacket 类对象。

上面只是定义了一个辅助类,具体系列化和反系列化工作要在对象流的子类中进行。

2.2 对象输出流子类的定义

```
class MyObjectOutputStream extends ObjectOutputStream
{
  public MyObjectOutputStream(OutputStream out) throws IOException
  {
    super(out); //调用超类构造方法
    enableReplaceObject(true);//返回 true
  }
  protected Object replaceObject(Object object) throws IOException
  {
    if(object.getClass() == = DatagramPacket.class)
      return new MyDatagramPacket((DatagramPacket) object);//进行替换
    else
      return object;
  }
}
```

2.3 对象输入流子类的定义

程序代码与 2.2 所述类似不重复描述

有了以上子类定义之后,当我们调用标准对象输出流 ObjectOutputStream 的 writeObject(Object obj) 方法时,如果写入的对象是 DatagramPacket 类对象时,就会将它转换成 MyDatagramPacket 类对象,实际写入的是 MyDatagramPacket 类对象。当我们调用标准对象输入流 ObjectInputStream 的 readObject() 方法时,如果返回的对象类型是 MyDatagramPacket 类,就会将对象转换成 DatagramPacket 类对象返回。这样就完成了 DatagramPacket 类对象的系列化的反系列化工作。主程序见所附实例(MyObjectStream.java),所附实例中的 DatagramPacket 类对象的 IP 地址是"time-A.timefreq.bldrdoc.gov",它是美国科罗拉多洲博尔德市的国家标准与技术研究所的时间服务器的地址,而端口号是 port=13。

3 总结

上面我们通过具体实例介绍了如何通过自定义对象流的方法来系列化非 Serializable 的 final 类对象,从上述过程可以看出,主要的工作是:

(1) 要实现一个辅助类,该类将非 Serializable 的 final 类对象转换成一个可 Serializable 的类对象;

(2) 写出对象流的子类,在对象流子类中重点重载方法 replaceObject()、enableReplaceObject()、resolveObject() 和 enableResolveObject()。对于其他非 Serializable 的 final 类对象的系列化工作完全可仿照完成。

(上述方法原型来自 Java 1.4 的 API 文档)