# 1. 其他IO流

## 1.1 打印流

打印流只负责数据的输出，打印流有字节打印流PrintStream和字符打印流PrintWriter。

在API中查看两个打印流的构造方法，发现打印流非常好用，既能根据OutputStream创建打印流，也能通过File对象创建打印流，还能直接通过文件名字符串创建打印流。并且在构造中还能指明流的编码。

在构造函数中，还能设置启用启动刷新（即设置autoFlush参数为true），这样流将自动调用flush()方法刷新流。

打印流提供了带有很多重载的print()方法（不换行打印），可以向流中打印各种类型的数据，同时还提供带有重载println()方法（换行打印），直接打印带有换行符的数据。print()和write()方法的区别在于print()提供自动刷新。普通的write()方法需要调用flush()或者close()方法才会真正写入数据。

打印流还提供了printf()格式化输出方法，这个和C语言中printf()用法类似。

|  |  |  |
| --- | --- | --- |
| No. | 字符 | 描述 |
| 1 | %s | 表示内容为字符串 |
| 2 | %d | 表示内容为整数 |
| 3 | %c | 表示内容为字符 |
| 4 | %f | 表示内容为小数 |

打印流使用例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.IOException; **import** java.io.PrintStream;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  PrintStream ps = **new** PrintStream(**"d:\\1.txt"**, **"utf-8"**);  ps.printf(**"%d %c \n"**, 10, **'a'**);  ps.print(**"hello"**);  ps.println();  ps.close();  } } |

这样，我们就能再理解一下以前使用的System类中的标准输入输出流。System类定义了两个静态成员变量：

public static final PrintStream out; 这就是标准输出流

public static final InputStream in; 这就是标准输入流

也就是说，Sysem.out和System.in就是普通的流对象，因此我们可以调用流中的方法，比如System.out.print()等。并且可用System.in来构造Scanner对象方便控制台输入数据。既然我们学习了流的使用，那么现在也可以不用Scanner，直接用流的方式来获取标准输入流的数据，例如：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.BufferedReader; **import** java.io.IOException; **import** java.io.InputStreamReader;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 为了方便处理，用高效字符流包装System.in，因为System.in是InputStream类型，可以传递* BufferedReader br = **new** BufferedReader(**new** InputStreamReader(System.***in***));  *// 读取一行* System.***out***.println(**"请输入数据，并按回车："**);  String data = br.readLine(); *// 此方法会等待输入，阻塞* System.***out***.println(**"你输入的数据是："** + data);  *// 关闭* br.close();  } } |

## 1.2 合并流SequenceInputStream

合并流SequenceInputStream，又叫序列流，可以将多个输入流串流在一起，合并为一个输入流。合并流其实就是从第一个输入流开始读取，直到到达文件末尾，接着从第二个输入流读取，依次类推，直到读取完最后一个输入流。

合并流的构造方法：

（1）SequenceInputStream(InputStream s1, InputStream s2)：用于合并两个输入流

（2）SequenceInputStream(Enumeration<? extends InputStream> e)：用于合并多个文件流。Enumeration中可存储多个输入流对象。

创建好合并流后，可以用字节流的读取方法来读取完合并流的文件。

例1：合并2个文件。

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.\*;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 创建三个流，目的是将1.txt和2.txt内容合并输出到3.txt* FileInputStream fis1 = **new** FileInputStream(**new** File(**"d:\\1.txt"**));  FileInputStream fis2 = **new** FileInputStream(**new** File(**"d:\\2.txt"**));  FileOutputStream fos = **new** FileOutputStream(**new** File(**"d:\\3.txt"**));  *// 创建合并流* SequenceInputStream sis = **new** SequenceInputStream(fis1, fis2);  *// 读取合并流并输出到fos中* **byte**[] buffer = **new byte**[1024];  **int** length = 0;  **while**((length = sis.read(buffer)) != -1) {  fos.write(buffer, 0, length);  }   fos.close();  sis.close(); *// fis1和fis2不需要关闭。因为sis.close()自动关闭它所有的输入流。查看sis.close()源码即可知道* } } |

例2：合并多个输入流，这样使用构造2，要Enumeration。前面学过，Vector集合的elements()方法就返回Enumeration。因此可先将多个输入流添加在Vector集合中。

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.\*; **import** java.util.Vector;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 创建四个流，目的是将1.txt、2.txt和3.txt内容合并输出到4.txt* FileInputStream fis1 = **new** FileInputStream(**new** File(**"d:\\1.txt"**));  FileInputStream fis2 = **new** FileInputStream(**new** File(**"d:\\2.txt"**));  FileInputStream fis3 = **new** FileInputStream(**new** File(**"d:\\3.txt"**));  FileOutputStream fos = **new** FileOutputStream(**new** File(**"d:\\4.txt"**));  *// 把输入流添加到Vector集合中* Vector<InputStream> streams = **new** Vector<>();  streams.add(fis1);  streams.add(fis2);  streams.add(fis3);  *// 创建合并流* SequenceInputStream sis = **new** SequenceInputStream(streams.elements());  *// 读取合并流并输出到fos中* **byte**[] buffer = **new byte**[1024];  **int** length = 0;  **while**((length = sis.read(buffer)) != -1) {  fos.write(buffer, 0, length);  }   fos.close();  sis.close();  } } |

如果不想用Vector，也能用自己想用的集合，然后自己实现Enumeration接口。需要先获得集合的迭代器，以便实现Enumeration。例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.\*; **import** java.util.ArrayList; **import** java.util.Enumeration; **import** java.util.Iterator;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 创建四个流，目的是将1.txt、2.txt和3.txt内容合并输出到4.txt* FileInputStream fis1 = **new** FileInputStream(**new** File(**"d:\\1.txt"**));  FileInputStream fis2 = **new** FileInputStream(**new** File(**"d:\\2.txt"**));  FileInputStream fis3 = **new** FileInputStream(**new** File(**"d:\\3.txt"**));  FileOutputStream fos = **new** FileOutputStream(**new** File(**"d:\\4.txt"**));  *// 把输入流添加到ArrayList集合中* ArrayList<InputStream> streams = **new** ArrayList<>();  streams.add(fis1);  streams.add(fis2);  streams.add(fis3);  *// 得到迭代器* Iterator<InputStream> streamIterator = streams.iterator();  *// 创建合并流* SequenceInputStream sis = **new** SequenceInputStream(**new** Enumeration<InputStream>() {  @Override  **public boolean** hasMoreElements() {  **return** streamIterator.hasNext();  }   @Override  **public** InputStream nextElement() {  **return** streamIterator.next();  }  });  *// 读取合并流并输出到fos中* **byte**[] buffer = **new byte**[1024];  **int** length = 0;  **while**((length = sis.read(buffer)) != -1) {  fos.write(buffer, 0, length);  }   fos.close();  sis.close();  } } |

即实现Enumeration接口需要重写hasMoreElements()和nextElement()方法，因此先得到集合的迭代器，使用迭代器的方法很方便。

## 1.3 对象的序列化和反序列化流

对象的序列化就是将对象转化成二进制流，将对象保存为二进制文件存储在磁盘上。将对象序列化便于对象数据的保存和在网络中的数据传递。而对象的反序列化就是将该二进制文件转换成对象在程序中使用，两者是相反的过程。

要序列化的对象所在的类必须实现可序列化接口Serializable，此接口像Cloneable接口一样，也是一个标记接口，即标记此类的对象是可序列化的。如果不想让类的某些成员序列化，那么使用transient关键字声明此变量，比如：private transient int age。

使用ObjectOutputStream将对象序列化并输出到文件，此类的一个构造方法就是接收一个OutputStream对象。调用writeObject(对象)方法来把对象输出到文件。

序列化例子：

（1）要序列化的Student对象：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.Serializable;  *// 实现序列化接口* **public class** Student **implements** Serializable {  **private** String **name**;  *// 让age不可序列化* **private transient int age**;   *// get set方法* **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public int** getAge() {  **return age**;  }   **public void** setAge(**int** age) {  **this**.**age** = age;  } } |

（2）创建Student对象并序列化。序列化关注的只是对象的数据。

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.FileOutputStream; **import** java.io.IOException; **import** java.io.ObjectOutputStream;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 创建对象* Student stu = **new** Student();  stu.setName(**"张三"**);  stu.setAge(12);  *// 创建对象序列化流* ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream(**"d:\\data.obj"**));  oos.writeObject(stu);  oos.close();  } } |

然后对象的数据就写入到了文件中，以文本编辑器查看文件，是乱码，因为这是JVM定义的二进制文件。

对象序列化后，很可能另一个文件中需要这个二进制文件，以此数据来创建对象。就需要反序列化。这时使用ObjectInputStream类，ObjectInputStream可用一个输入流创建，然后使用readObject()方法来获得对象，此时得到的对象可强转成对应的类型，并且其中带有序列化的数据。当然，之前用transient关键字修饰的成员只会是默认值，比如0或者null，因为该成员数据之前没有被序列化。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.\*;  **public class** Demo {  **public static void** main(String[] args) **throws** Exception {  ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream(**"d:\\data.obj"**));  *// 强制转换* Student student = (Student) ois.readObject();  System.***out***.println(student.getName()); *// 输出“张三”* System.***out***.println(student.getAge()); *// 输出默认0，因为age没有序列化* ois.close();  } } |

一个对象序列化后，如果现在对象的类代码有了微小的改动，那么在反序列化时就会出现异常，因为对象数据不匹配了。为了兼容性，可在类中添加serialVersionUID标识符变量，即：private static final long serialVersionUID = Xxx。这样就能使不同的类版本间数据兼容。

### 1.3.1 对象的克隆

对象的克隆也就是“拷贝”，即将原对象复制一份保存在另一个对象中。注意，克隆不是引用同一个对象，而是有两个相同的对象，操作一个对象时，不会影响另一个对象。

对象的克隆有两种方式：

（1）浅克隆：会克隆对象的所有变量和值，但是对于该对象中的引用类型的变量，克隆后的对象仍然只指向该对象的地址，换言之，浅克隆仅复制被克隆对象中的内容，而不复制它所引用的对象。我们之前所用的Object的clone()方法就是采用浅克隆。

例子：

Address类：

|  |
| --- |
| **package** com.zhang.bean;  **public class** Address {  **private** String **name**;  **public** Address(String name) {  **this**.**name** = name;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   @Override  **public** String toString() {  **return "Address{"** +  **"name='"** + **name** + **'\''** +  **'}'**;  } } |

Student类（Student类中维护了一个Address属性），该类实现Clonable接口。因为Object的clone()是protected修饰符，为了能让clone()也能在此类的外界访问，我们重写clone()方法时，可将修饰符扩大为public：

|  |
| --- |
| **package** com.zhang.bean;  **public class** Student **implements** Cloneable {  **private** String **name**;  **private** Address **addr**;  **public** Student(String name, Address addr) {  **this**.**name** = name;  **this**.**addr** = addr;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** Address getAddr() {  **return addr**;  }   **public void** setAddr(Address addr) {  **this**.**addr** = addr;  }   @Override  **public** Object clone() **throws** CloneNotSupportedException {  **return super**.clone();  }   @Override  **public** String toString() {  **return "Student{"** +  **"name='"** + **name** + **'\''** +  **", addr="** + **addr** +  **'}'**;  } } |

测试：

|  |
| --- |
| **package** com.zhang.test;  **import** com.zhang.bean.Address; **import** com.zhang.bean.Student;  **public class** Demo {  **public static void** main(String[] args) **throws** CloneNotSupportedException {  Student student1 = **new** Student(**"张三"**, **new** Address(**"江苏"**));  *// 克隆一份* Student student2 = (Student) student1.clone();  *// 分别修改student2的name属性和student1的address的name属性* student2.setName(**"王五"**);  student1.getAddr().setName(**"北京"**);  System.***out***.println(student1);  System.***out***.println(student2);  */\*  输出结果：  Student{name='张三', addr=Address{name='北京'}}  Student{name='王五', addr=Address{name='北京'}}  发现name是克隆的，addr只是引用了地址。所以修改任意一个student中addr的name属性，另一个对象的addr的name就会改变。  但是，如果使用new使对象的addr指向一个新对象，那么他们就不会影响了。比如  student2.setAddr(new Address('北京'));  \*/* } } |

（2）深克隆：会克隆对象的所有变量和值，对于该对象中的引用类型的变量，也同样克隆，不再是指向该对象的地址，换言之，深克隆将把要复制的对象所引用的对象都复制了一遍。我们所用的序列化和反序列化就是采用的深克隆。注意：相关的类都要实现序列化Serializable接口。例如上面的Student和Address都要实现Serializable接口。

# 2. Properties类

Properties是一个和流关联的集合对象。

Properties实现了Map<Object, Object>接口，Properties是没有泛型的。可使用Properties存储键值对。可用Map集合通用的put(Object, Object)和get(key)方法存、取键值对。

Properties提供特有的方法来存、取String类型的键值对，即setProperty(String, String)和getProperty(key)方法。特别是在读取配置文件时，常用这两个方法。同样也提供了stringPropertyNames()方法来获取所有的键的Set集合。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.Iterator; **import** java.util.Properties; **import** java.util.Set;  **public class** Demo {  **public static void** main(String[] args) {  Properties prop = **new** Properties();  prop.setProperty(**"name"**, **"张三"**);  prop.setProperty(**"age"**, **"16"**);  prop.setProperty(**"gender"**, **"男"**);  *// 获得键的set集合* Set<String> keySet = prop.stringPropertyNames();  *// 遍历* **for**(Iterator<String> it = keySet.iterator(); it.hasNext(); ) {  String keyName = it.next();  System.***out***.println(keyName + **" : "** + prop.getProperty(keyName));  }  } } |

Properties类的重点在于其中的store()和load()方法。

这两个方法常用来设置和读取配置文件。

（1）store方法原型：public void store(Writer writer, String comments)：即把集合中的键值对数据存储到文件。参数1需要一个Writer对象，参数2是该文件的注释。此方法的另一个重载中，参数1也可传递输出流，方法比较灵活。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.BufferedWriter; **import** java.io.IOException; **import** java.io.PrintWriter; **import** java.util.Iterator; **import** java.util.Properties; **import** java.util.Set;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  Properties prop = **new** Properties();  prop.setProperty(**"name"**, **"张三"**);  prop.setProperty(**"age"**, **"16"**);  prop.setProperty(**"gender"**, **"男"**);  *// 创建一个Writer对象（只要是Writer实现类即可）* PrintWriter pw = **new** PrintWriter(**"example.properties"**, **"UTF-8"**);  *// 调用store方法* prop.store(pw, **"这是该配置文件的注释"**);  pw.close();  } } |

运行程序后，发现example.properties配置文件是如下内容（其中第二行的时间信息是根据文件创建时间自动生成的）：

|  |
| --- |
| *#\u8FD9\u662F\u8BE5\u914D\u7F6E\u6587\u4EF6\u7684\u6CE8\u91CA #Wed Jan 18 15:02:39 CST 2017* **age**=**16 name**=**张三 gender**=**男** |

发现此properties配置文件的注释部分是用#表示的，是行注释（注释部分中文文字用编码表示了）。properties配置文件中只有这种注释。其他的键值对信息是用等于号连接起来表示的，每个键值对占一行。

这就是store方法生成配置文件的作用。

（2）load()方法原型：public void load(Reader reader):把文件中的数据读取到集合中。此方法就是把配置文件的读取以便程序中使用。load()调用完毕，Property集合中就有了相应数据可供使用。

例子：读取刚刚生成的配置文件

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.\*; **import** java.util.Properties;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 创建Reader对象* BufferedReader br = **new** BufferedReader(**new** InputStreamReader(**new** FileInputStream(**new** File(**"example.properties"**)), **"UTF-8"**));  Properties prop = **new** Properties();  *// load加载* prop.load(br);  *// 加载完即可关闭流对象* br.close();  *// 读取配置文件* System.***out***.println(prop);  *// 单个读取* System.***out***.println(prop.getProperty(**"name"**));  System.***out***.println(prop.getProperty(**"age"**));  System.***out***.println(prop.getProperty(**"gender"**));  System.***out***.println(prop.getProperty(**"hello"**)); *// 不存在的返回null* } } |

如果自己写配置文件，也要遵循用等于号的规则。如果想在程序中改变配置文件，需要setProperty，然后再store，如果没有store，是不会存储的。这就是用Properties读取和设置配置文件。