数据流

数据流`DataInputStream`也是`FilterInputStream`的子类，同样采用装饰者模式，最大的不同是它支持基本数据类型的直接读取：

```
public static void main(String[] args) {
    try (DataInputStream dataInputStream = new DataInputStream(new FileInputStream("test.txt"))){
        System.out.println(dataInputStream.readBoolean());   //直接将数据读取为任意基本数据类型
    }catch (IOException e) {
        e.printStackTrace();
    }
}
```

用于写入基本数据类型：

```
public static void main(String[] args) {
    try (DataOutputStream dataOutputStream = new DataOutputStream(new FileOutputStream("output.txt"))){
        dataOutputStream.writeBoolean(false);
    }catch (IOException e) {
        e.printStackTrace();
    }
}
```

注意，写入的是二进制数据，并不是写入的字符串，使用`DataInputStream`可以读取，一般他们是配合一起使用的。

对象流

既然基本数据类型能够读取和写入基本数据类型，那么能否将对象也支持呢？`ObjectOutputStream`不仅支持基本数据类型，通过对对象的序列化操作，以某种格式保存对象，来支持对象类型的IO，注意：它不是继承自`FilterInputStream`的。

```
public static void main(String[] args) {
    try (ObjectOutputStream outputStream = new ObjectOutputStream(new FileOutputStream("output.txt"));
         ObjectInputStream inputStream = new ObjectInputStream(new FileInputStream("output.txt"))){
        People people = new People("lbw");
        outputStream.writeObject(people);
        outputStream.flush();
        people = (People) inputStream.readObject();
        System.out.println(people.name);
    }catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
```

```
    }
}

static class People implements Serializable{   //必须实现Serializable接口才能被序列化
    String name;

    public People(String name){
        this.name = name;
    }
}
```

在我们后续的操作中，有可能会使得这个类的一些结构发生变化，而原来保存的数据只适用于之前版本的这个类，因此我们需要一种方法来区分类的不同版本：

```
static class People implements Serializable{
    private static final long serialVersionUID = 123456;   //在序列化时，会被自动添加这个属性，它代表当前类的版本，我们也可以手动指定版本。

    String name;

    public People(String name){
        this.name = name;
    }
}
```

当发生版本不匹配时，会无法反序列化为对象：

```
java.io.InvalidClassException: com.test.Main$People; local class incompatible: stream classdesc serialVersionUID = 123456, local class serialVersionUID = 1234567
    at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:699)
    at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:2003)
    at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1850)
    at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:2160)
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1667)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:503)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:461)
    at com.test.Main.main(Main.java:27)
```

如果我们不希望某些属性参与到序列化中进行保存，我们可以添加`transient`关键字：

```
public static void main(String[] args) {
    try (ObjectOutputStream outputStream = new ObjectOutputStream(new FileOutputStream("output.txt"));
         ObjectInputStream inputStream = new ObjectInputStream(new FileInputStream("output.txt"))){
        People people = new People("lbw");
        outputStream.writeObject(people);
        outputStream.flush();
```

```
        people = (People) inputStream.readObject();
        System.out.println(people.name);  //虽然能得到对象，但是name属性并没有保存，因此为null
    }catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }
}

static class People implements Serializable{
    private static final long serialVersionUID = 1234567;

    transient String name;

    public People(String name){
        this.name = name;
    }
}
```

其实我们可以看到，在一些`JDK`内部的源码中，也存在大量的transient关键字，使得某些属性不参与序列化，取消这些不必要保存的属性，可以节省数据空间占用以及减少序列化时间。

⭐ 当然有写就会有读