Java基础IO

本节目标

> - 掌握Java中基本的IO流的基本原理
> - 熟悉Java的IO流的使用

1. 理解文件

1.1 站在课本角度

- 文件是相关记录或者放在一起的数据的集合(懵逼...)

1.2 站在日常实用角度

- 你在windows操作中，经常在硬盘上创建的各种.txt, .doc, .exe, .java, .lib, .mp3等等，都可以称之为文件

1.3 理解文件

- 文件简单的可以理解成，在外设硬盘上面保存数据的一种方式
- 文件一共可以由两部分构成：属性(文件大小，文件名，文件类型等)+内容(就是文件里面放的是什么)
- 所以我们学`JavaIO`，学什么呢？就学对文件的属性和内容进行操作，而实际写入或者读取的过程，我们称之为IO

2. File文件操作类

在`java.io`包之中，用`File`类来对文件进行操作(创建、删除、取得信息等)

2.1 File类使用-准备

- 可以先看看官网文档中关于File类的说明https://docs.oracle.com/javase/8/docs/api/，重点关注构造函数(此处要介绍文档查看方式)
- `java.io.File`类是一个普通的类,如果要实例化对象,则常用到两个构造方法

方法	解释
`public File(String pathname)`	创建指定路径文件对象
`public File(String parent, String child)`	同上，但可指明父路径和子路径

为了方便表述，我们先做两方面准备：1. 创建`FileDemo.java`, 方便测试, 2. 指定并创建文件测试路径为`E:\java_code\file`

```
package com.bittech;

public class FileDemo {
    public static void main(String[] args){
        //注意最后的\\,拼接，不是必须的，仅仅为了让同学们理解路径的构成  文件路径+文件名
        String path = "E:\\java_code\\file\\";
        String name = "demo.txt";
        String pathname = path + name;
    }
}
```

2.2 File类常用方法-基本文件操作

方法	说明
`public boolean exists()`	测试指定路径中文件或者目录是否存在
`public boolean isDirectory()`	判定一个文件是目录
`public boolean isFile()`	判定是否是文件
`public boolean delete()`	删除文件
`public boolean createNewFile() throws IOException`	创建一个新文件

范例1: 检测文件是否存在

```
package com.bittech;

import java.io.File;

public class FileDemo {
    public static void main(String[] args){
        String path = "E:\\java_code\\file\\";
        String name = "demo.txt";
        String pathname = path + name;

        File file = new File(pathname);
        boolean res = file.exists();

        System.out.println("文件" + pathname + "是否存在: " + res);
    }
}
```

结果

```
文件E:\java_code\file\demo.txt是否存在: true
```

为了方便测试，在`E:\java_code\file`下手动创建`demo.txt`

范例2: 检测是否是目录，是否是文件，删除

```
package com.bittech;
```

```
import java.io.File;

public class FileDemo {
    public static void main(String[] args){
        String path = "E:\\java_code\\file\\";
        String name = "demo.txt";
        String pathname = path + name;

        File file = new File(pathname);

        boolean res = file.exists();
        System.out.println("文件" + pathname + "是否存在: " + res);

        res = file.isDirectory();
        System.out.println("文件" + pathname + "是否是目录: " + res);

        res = file.isFile();
        System.out.println("文件" + pathname + "是否是文件: " + res);

        file.delete();
        if(!file.exists()){
            System.out.println("删除文件成功! ");
        }
        else{
            System.out.println("删除文件失败! ");
        }
    }
}
```

结果

```
文件E:\java_code\file\demo.txt是否存在: true
文件E:\java_code\file\demo.txt是否是目录: false
文件E:\java_code\file\demo.txt是否是文件: true
删除文件成功!
```

再去`E:\java_code\file`路径下去看，发现文件已经不存在

范例3：创建新文件

```
package com.bittech;

import java.io.File;
import java.io.IOException;

public class FileDemo {
    public static void main(String[] args) throws IOException {
        String path = "E:\\java_code\\file\\";
        String new_name = "demo1.txt";
        String pathname = path + new_name;

        File file = new File(pathname);
        if(!file.exists()){ //注意'!', 表示取反
            try {
                //有创建失败的风险，需要捕捉异常,异常如果不了解，可以先使用，不必关心细节
                file.createNewFile();
```

```
            }catch (IOException e){ //文件部分异常，常见为IOException
                System.out.println("文件" + pathname + "创建失败");
                //e.printStackTrace();
            }
        }
        else{
            System.out.println("文件" + pathname + "已经存在，不需创建");
        }
    }
}
```

结果：`E:\java_code\file`路径下，新增指定`dome1.txt`文件。可以再次运行，就会看到已经存在的提示

```
文件E:\java_code\file\demo1.txt已经存在，不需创建
```

其中`File`类只是创建文件本身，但是对于其内容并不做处理。

示例4：编写文件的基本操作(如果文件不存在则进行创建；存在则删除)

```
package com.bittech;

import java.io.File;
import java.io.IOException;

public class FileDemo {
    public static void main(String[] args) throws IOException {
        String path = "E:\\java_code\\file\\";
        String new_name = "demo.txt";
        String pathname = path + new_name;

        File file = new File(pathname);
        if(file.exists()){
            //文件存在，删除
            file.delete();
            System.out.println("文件" + pathname + "存在，删除之");
        }
        else{
            //文件不存在，创建
            try {
                file.createNewFile(); //有创建失败的风险，需要捕捉异常
                System.out.println("文件" + pathname + "不存在，创建之");
            }catch (IOException e){ //文件部分异常，常见为IOException
                System.out.println("文件" + pathname + "创建失败");
                e.printStackTrace();
            }
        }
    }
}
```

以上实现了最简化的文件处理操作，但是代码存在两个问题：

> - 实际项目部署环境可能与开发环境不同。那么这个时候路径分隔符是一个很重要的问题。`windows`下使用的是`\`,而`Unix/Linux`系统下使用的是`/`。所以在使用路径分隔符时都会采用File类的一个常量`public static final String separator`来描述。

```java
package com.bittech;

import java.io.File;
import java.io.IOException;

public class FileDemo {
    public static void main(String[] args) throws IOException {
        System.out.println(File.separator);
    }
}
//我的平台是win 10，所以结果是'\'，同学们可以根据该分隔符，来确定自己的平台以及路径分割符
```

> - 在Java中要进行文件的处理操作是要通过本地操作系统支持的，在这之中如果操作的是同名文件，就可能出现延迟的问题。（开发之中尽可能避免文件重名问题，该问题我们后面再研究）

2.3 File类常用方法-目录操作

方法	解释
`public boolean mkdir()`	创建一个空目录
`public boolean mkdirs()`	创建目录(无论有多少级父目录，都会创建)
`public String getParent()`	取得父路径
`public File getParentFile()`	取得父File对象

示例1: 创建指定一个或者多个目录的路径

```java
package com.bittech;

import java.io.File;
import java.io.IOException;

public class FileDemo {
    public static void main(String[] args) throws IOException {
        String path = "E:\\java_code\\file\\";
        String dir_name = "demo_dir"; //想要创建的目录
        //String dir_name = "demo_dir\\dir1\\dir2\\dir3"; //想要创建的目录路径
        String pathname = path + dir_name;

        File file = new File(pathname);
        if(!file.exists()){
            file.mkdir(); //创建一个空目录
            //file.mkdirs(); //创建一个可能具有多个目录的路径
        }
        else{
            System.out.println("路径已经存在，不需创建");
        }
    }
}
```

结果：`E:\java_code\file`路径下，观察路径变化

示例2: 获取父路径

```
package com.bittech;

import java.io.File;
import java.io.IOException;

public class FileDemo {
    public static void main(String[] args) throws IOException {
        String path = "E:\\java_code\\file\\";
        String dir_name = "demo_dir\\dir1\\dir2\\dir3";
        String pathname = path + dir_name;

        File file = new File(pathname);
        System.out.println(file.getParent());
    }
}
```

结果

```
E:\java_code\file\demo_dir\dir1\dir2
```

示例3: 获取父目录`File`对象，并在父目录下，创建文件`demo.java`

```
package com.bittech;

import java.io.File;
import java.io.IOException;

public class FileDemo {
    public static void main(String[] args) throws IOException {
        String path = "E:\\java_code\\file\\";
        String dir_name = "demo_dir\\dir1\\dir2\\demo.java";
        String pathname = path + dir_name;
        File file = new File(pathname);

        File pfile = file.getParentFile(); //获取父目录File对象
        if(!pfile.exists()){//检测路径是否存在，不存在创建
            pfile.mkdirs();
            //获取File对象的绝对路径，后面学，这个先用起来
            System.out.println("路径" + pfile.getAbsolutePath() + "不存在，创建");
        }
        if(!file.exists()){
           file.createNewFile();
        }
    }
}
```

结果

```
路径E:\java_code\file\demo_dir\dir1\dir2不存在，创建
```

以上判断父目录是否存在以及父目录的创建操作非常重要，对于后续开发很重要。

2.4 File类常用方法-文件属性操作

方法	解释
`public long length()`	取得文件大小(字节)
`public long lastModified()`	最后一次修改日期

示例：取得文件信息

```
package com.bittech;

import java.io.File;
import java.io.IOException;
import java.util.Date;

public class FileDemo {
    public static void main(String[] args) throws IOException {
        String path = "E:\\java_code\\file\\";
        String name = "demo.txt";//可以替换成你想检测的文件
        String pathname = path + name;
        File file = new File(pathname);
        if(!file.exists()){
            file.createNewFile();
        }
        System.out.println("文件 " + name + "size : " + file.length());
        //Date类我们后面学
        System.out.println("文件 " + name + "最近修改时间: " +  new
Date(file.lastModified()));
    }
}
```

结果：

```
文件 demo.txtsize : 3
文件 demo.txt最近修改时间: Sun Aug 11 15:50:47 CST 2019
```

2.5 绝对路径与相对路径

绝对路径：

是指目录下的绝对位置，直接到达目标位置，通常是从盘符开始的路径。完整的描述文件位置的路径就是绝对路径 。如：`E:\\javacode\\Java8\\Test.java`。通常：http://www.sun.com/index.htm也代表了一个URL绝对路径。

当前目录：

这个参数还可以使用一些常用的路径表示方法,例如”.”或”./”代表当前目录,这个目录也就是`jvm`启动路径 .所以如下代码能得到当前`.java`文件的当前目录的完整路径:(**注意：**这个目录也是当前`idea`中的`JVM`的启动路径，我们可以修改这个路径：在`Run->edit Configurations->Working directory`可以看到)

```
File f = new File(".");
String absolutePath = f.getAbsolutePath();
System.out.println(absolutePath);
```

`E:\\javacode\Java8\.`注意这个点号。

相对路径:

相对与某个基准目录的路径 。使用相对路径可以为我们带来非常多的便利 。如当前路径为`E:\\javacode`，要描述上述路径(`E:\\javacode\\Java8\\Test.java`)，只需输入:`Java8\\Test.java` 。此时的路径是相对`E:\\javacode`来说的。

我们可以使用相对路径来创建文件,例如:

```
File file = new File("a.txt");
File.createNewFile();
```

此时生成的文件就在你当前目录下面。假设当前的路径为：`E:\\javacode\\Java8`

2.6 File类常用方法-其他操作

方法	解释
`public File[] listFiles()`	列出一个目录指定的全部组成

示例：列出desktop目录中的全部组成

```
package com.bittech;

import java.io.File;
import java.io.IOException;
import java.util.Date;

public class FileDemo {
    public static void main(String[] args) throws IOException {
        // 要操作的文件
        File file = new File("C:\\Users\\whb\\Desktop");
        // 保证是个目录且存在
        if (file.exists() && file.isDirectory()) {
            // 列出目录中的全部内容
            File[] result = file.listFiles();
            for (File file2 : result) {
                System.out.println(file2);
            }
        }
    }
}
```

2.6 综合案例-递归打印文件目录列表

虽然File提供有`listFiles()`方法，但是这个方法本身只能够列出本目录中的第一级信息。如果要求列出目录中所有级的信息，必须自己来处理。这种操作就必须通过递归的模式来完成。

示例：递归列出desktop目录中的全部组成

```
package com.bittech;
import java.io.File;

public class FileDemo {

    public static void main(String[] args) {
```

```
        File file = new File("C:\\Users\\whb\\Desktop") ;
        listAllFiles(file) ; // 从此处开始递归
    }
    /**
     * this methods is used for 列出指定目录中的全部子目录信息 yuisama 2017年11月27日
     * @param file
     */
    public static void listAllFiles(File file) {
        if (file.isDirectory()) { // 现在给定的file对象属于目录
            File[] result = file.listFiles() ; // 继续列出子目录内容
            if (result != null) {
                for (File file2 : result) {
                    listAllFiles(file2) ;
                }
            }
        }else {
            // 给定的file是文件，直接打印
            System.out.println(file) ;
        }
    }
}
```

3.流

3.1 流的概念

流：在Java中所有数据都是使用流读写的。流是一组有顺序的，有起点和终点的字节集合，是对数据传输的总称或抽象。即数据在两设备间的传输称为流，流的本质是数据传输，根据数据传输特性将流抽象为各种类，方便更直观的进行数据操作。

> 1.按照流向分：输入流；输出流
>
> 2.按照处理数据的单位分：字节流(8位的字节)；字符流(16位的字节)

3.2 什么是输入输出流

输入就是将数据从各种输入设备（包括文件、键盘等）中读取到内存中。

输出则正好相反，是将数据写入到各种输出设备（比如文件、显示器、磁盘等）。

例如键盘就是一个标准的输入设备，而显示器就是一个标准的输出设备，**但是文件既可以作为输入设备，又可以作为输出设备。**

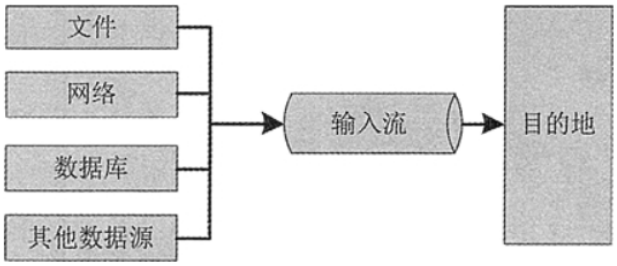


图1 输入流模式

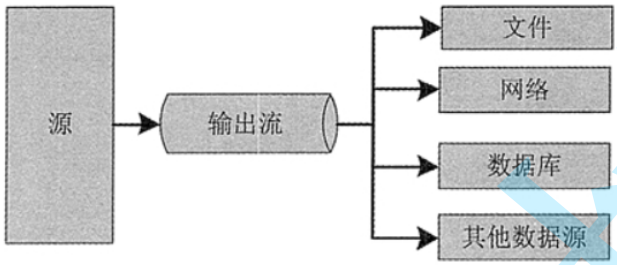


图2 输出流模式

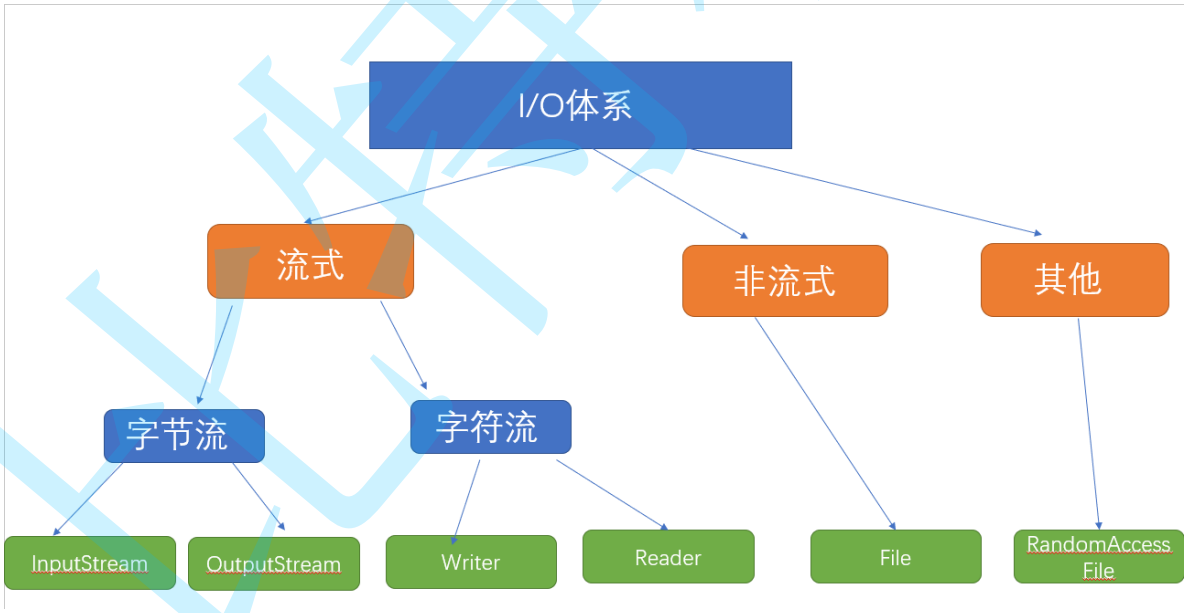
3.3 什么是字节流，字符流

File类不支持文件内容处理，如果要处理文件内容，必须要通过流的操作模式来完成。

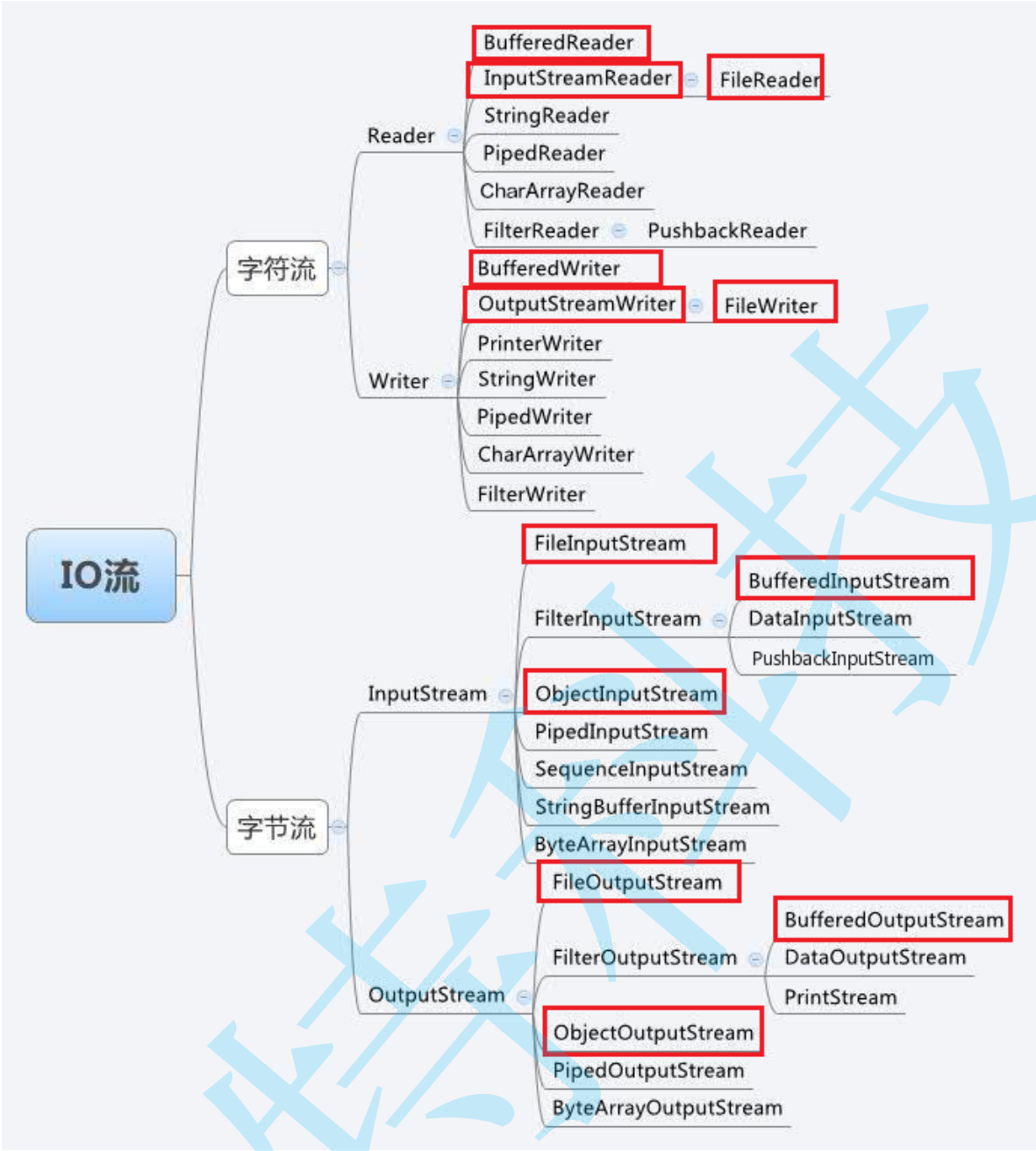
在java.io包中，流分为两种：字节流与字符流

- 字节流：数据流中最小的数据单元是字节 。InputStream、OutputStream
- 字符流：数据流中最小的数据单元是字符， Java中的字符是Unicode编码，一个字符占用两个字节 。Reader、Writer

Java中IO流的体系结构如图



JavaIO类图



3.4 字节流

1、`FileInputStream`和`FileOutputStream`

```
public  class FileInputStream extends InputStream {}
```

- `FileInputStream`从文件系统中的某个文件中获得输入字节。
- `FileInputStream`用于读取诸如图像数据之类的原始字节流。

方法	解释
`FileInputStream(File file)`	通过打开与实际文件的连接创建一个`FileInputStream`，该文件由文件系统中的`File`对象`file`命名
`FileInputStream(String name)`	通过打开与实际文件的连接来创建一个`FileInputStream`，该文件由文件系统中的路径名`name`命名。

```
public class FileOutputStream extends OutputStream
```

- 文件输出流是用于将数据写入到输出流`File`或一个`FileDescriptor`。 文件是否可用或可能被创建取决于底层平台。
- 特别是某些平台允许一次只能打开一个文件来写入一个`FileOutputStream`（或其他文件写入对象）。 在这种情况下，如果所涉及的文件已经打开，则此类中的构造函数将失败。

方法	解释
`FileOutputStream(File file)`	创建文件输出流以写入由指定的`File`对象表示的文件。
`FileOutputStream(String name)`	创建文件输出流以指定的名称写入文件。

示例使用`FileInputStream`和`FileOutputStream`复制图片：

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class Test {
 public static void main(String[] args) throws IOException {
    FileInputStream fin=new FileInputStream("E:\\bit\\bit.jpg");
    FileOutputStream fout=new FileOutputStream("E:\\bit\\bit\\BitCopy.jpg");
    int len=0;
    byte[] buff=new byte[1024];
    while((len=fin.read(buff))!=-1) {
     fout.write(buff, 0, len);
    }
    fin.close();
    fout.close();
 }
}
```

2、字节缓冲流`BufferedInputStream`和`BufferedOutputStream`

问题一：为什么需要有缓冲流?

> 答：当我们用read()读取文件时，每读一个字节，访问一次硬盘，效率很低。文件过大时，操作起来也不是很方便。因此我们需要用到buffer缓存流，**当创建buffer对象时，会创建一个缓冲区数组。**当我们读一个文件时，先从硬盘中读到缓冲区，然后直接从缓冲区输出即可，效率会更高。

```
public class BufferedInputStream extends FilterInputStream
```

- `BufferedInputStream`为另一个输入流添加了功能，即缓冲输入和支持`mark`和`reset`方法的功能。 当创建`BufferedInputStream`时，将创建一个内部缓冲区数组。
- 当从流中读取或跳过字节时，内部缓冲区将根据需要从所包含的输入流中重新填充，一次有多个字节。 `mark`操作会记住输入流中的一点，并且`reset`操作会导致从最近的`mark`操作之后读取的所有字节在从包含的输入流中取出新的字节之前重新读取。

方法	解释
`BufferedInputStream(InputStream in)`	创建一个`BufferedInputStream`并保存其参数，输入流`in`，供以后使用。
`BufferedInputStream(InputStream`	创建`BufferedInputStream`具有指定缓冲区大小，

in, int size)	解释存其参数，输入流 in ，供以后使用。

```
public class BufferedOutputStream  extends FilterOutputStream
```

- 该类实现缓冲输出流。 通过设置这样的输出流，应用程序可以向底层输出流写入字节，而不必为写入的每个字节导致底层系统的调用。

方法	解释
BufferedOutputStream(OutputStream out)	创建一个新的缓冲输出流，以将数据写入指定的底层输出流。
BufferedOutputStream(OutputStream out, int size)	创建一个新的缓冲输出流，以便以指定的缓冲区大小将数据写入指定的底层输出流。

示例使用 BufferedInputStream 和 BufferedOutputStream 实现文件拷贝：

```
import java.io.*;

public class TestBufferStreamCopy {
    public static void main(String[] args) throws IOException {
        File file=new File("bit.txt");
        if(!file.isFile()){   return;   }
        BufferedInputStream bfis=new BufferedInputStream(new
FileInputStream(file));
        BufferedOutputStream  bfos=new BufferedOutputStream(new
FileOutputStream("src\\"+file.getName()));//copy到src目录下
        byte bytes[]=new byte[1024];
        int temp=0;       //边读边写
        while ((temp=bfis.read(bytes))!=-1){//读
            bfos.write(bytes,0,temp);    //写
        }
        bfos.flush();
        bfos.close();
        bfis.close();
        System.out.println("copy成功！");
    }
}
```

有无缓冲效率的对比(以读为例)

```
import java.io.*;

public class BufferByte {
    public static void main(String[] args) throws IOException {
        //bit.txt 大小大约为1500KB
        File file=new File("bit.txt");
        //缓冲流
        BufferedInputStream bfis=new BufferedInputStream(new
FileInputStream(file));
        int temp=0;
        long time=System.currentTimeMillis();//获取当前时间至1970-1-1的毫秒数
        while ((temp=bfis.read())!=-1){
            //System.out.print((char) temp);
```

```
        }
        time=System.currentTimeMillis()-time;
        bfis.close();
        System.out.println("缓冲流读：  "+time+"ms");

        //非缓冲
        FileInputStream fis=new FileInputStream(file);
        temp=0;
        time=System.currentTimeMillis();
        while ((temp=fis.read())!=-1){
            //System.out.print((char) temp);
        }
        time=System.currentTimeMillis()-time;
        fis.close();
        System.out.println("非缓冲区读：  "+time+"ms");
    }
}
```

结果输出：

```
缓冲流读：  37ms
非缓冲区读：  3831ms
```

3.5 字符流

1、字符流`FileReader`和`FileWriter`

```
public class FileReader extends InputStreamReader
```

- 如果要从文件中读取内容，可以直接使用`FileReader`子类。
- `FileReader`是用于读取字符流。 要读取原始字节流，请考虑使用`FileInputStream`。

方法	解释
`FileReader(File file)`	创建一个新的`FileReader`，给出`File`读取。
`FileReader(String fileName)`	创建一个新的`FileReader`，给定要读取的文件的名称。

```
public class FileWriter extends OutputStreamWriter
```

- 如果是向文件中写入内容，应该使用`FileWriter`子类
- `FileWriter`是用于写入字符流。 要编写原始字节流，请考虑使用`FileOutputStream`

方法	解释
`FileWriter(File file)`	给一个File对象构造一个FileWriter对象。
`FileWriter(String fileName)`	构造一个给定文件名的FileWriter对象。

示例使用`FileReader`和`FileWriter`复制文件：

```
public class CopyFileDemo {
    public static void main(String[] args) throws IOException {
```

```
        //创建输入流对象
        FileReader fr = new FileReader("E:\\bit\\bitSrc.java");
        //创建输出流对象
        FileWriter fw = new FileWriter("E:\\bit\\bitCopy.java");

        //读写数据
        int ch;
        while((ch=fr.read())!=-1) {
            fw.write(ch);
        }
        //释放资源
        fw.close();
        fr.close();
    }
```

2、字符缓冲流 `BufferedReader` 和 `BufferedWriter`

为了提高字符流读写的效率，引入了缓冲机制，进行字符批量的读写，提高了单个字符读写的效率。`BufferedReader` 用于加快读取字符的速度，`BufferedWriter` 用于加快写入的速度。

`BufferedReader` 和 `BufferedWriter` 类各拥有 `8192个` 字符的缓冲区。当 `BufferedReader在` 读取文本文件时，会先尽量从文件中读入字符数据并放满缓冲区，而之后若使用read()方法，会先从缓冲区中进行读取。如果缓冲区数据不足，才会再从文件中读取，使用 `BufferedWriter` 时，写入的数据并不会先输出到目的地，而是先存储至缓冲区中。如果缓冲区中的数据满了，才会一次对目的地进行写出。

```
public class BufferedReader extends Reader
```

- 从字符输入流读取文本，缓冲字符，以提供字符，数组和行的高效读取

方法	解释
`BufferedReader(Reader in)`	创建使用默认大小的输入缓冲区的缓冲字符输入流。
`BufferedReader(Reader in, int sz)`	创建使用指定大小的输入缓冲区的缓冲字符输入流。

```
public class BufferedWriter  extends Writer
```

- 将文本写入字符输出流，缓冲字符，以提供单个字符，数组和字符串的高效写入。

方法	解释
`BufferedWriter(Writer out)`	创建使用默认大小的输出缓冲区的缓冲字符输出流。
`BufferedWriter(Writer out, int sz)`	创建一个新的缓冲字符输出流，使用给定大小的输出缓冲区。

示例使用 `BufferedReader` 和 `BufferedWriter` 进行文件拷贝

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
```

```
import java.io.IOException;

public class FileWrite {

 public static void main(String[] args) throws IOException {

     FileReader reader=new FileReader("E:\\BIT\\bit.txt");
     BufferedReader bReader=new BufferedReader(reader);
     FileWriter writer=new FileWriter(":\\BIT\\bit2.txt");
     BufferedWriter bWriter=new BufferedWriter(writer);
     String content="";
     //readLine一行一行的读取
     while((content=bReader.readLine())!=null){
         //\r\n换行
         bWriter.write(content+"\r\n");
     }
     /**
      * 关闭流的顺序：
      * 当A依赖B的时候先关闭A，再关闭B
      * 带缓冲的流最后关闭的时候会执行一次flush
      */
     reader.close();
     bReader.close();
     bWriter.close();
     writer.close();
 }

}
```

3.6 字节流对比字符流

1、字节流操作的基本单元是字节；字符流操作的基本单元为Unicode码元。

2、字节流在操作的时候本身不会用到缓冲区的，是与文件本身直接操作的；而字符流在操作的时候使用到缓冲区的。

3、所有文件的存储都是字节(byte)的存储，在磁盘上保留的是字节。

4、在使用字节流操作中，即使没有关闭资源（close方法），也能输出；而字符流不使用close方法的话，不会输出任何内容。

3.7 字符字节转换流

有时候我们需要进行字节流与字符流二者之间的转换，因为这是两种不同的流，所以，在进行转换的时候我们需要用到`OutputStreamWriter`和`InputStreamReader`。

`InputStreamReader`是Reader的子类，将输入的字节流转换成字符流。

```
public class InputStreamReader extends Reader
```

- `InputStreamReader`是从字节流到字符流的桥：它读取字节，并使用指定的`charset`将其解码为字符。
- 它使用的字符集可以由名称指定，也可以被明确指定，或者可以接受平台的默认字符集。

方法	解释
`InputStreamReader(InputStream in)`	创建一个使用默认字符集的 `InputStreamReader`。
`InputStreamReader(InputStream in, Charset cs)`	创建一个使用给定字符集的 `InputStreamReader`。

```
/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: GAO BO
 * Date: 2019-04-04
 * Time: 11:41
 */
public class TestDemo {
    public static void main(String[] args) {
        // 创建字节流对象 System.in 代表从控制台输入
        InputStream in = System.in;
        // 创建字符流对象
        BufferedWriter bw = null;
        BufferedReader br = null;

        try {
            // 实例化字符流对象 通过 InputStreamReader 将字节输入流转化成字符输入流
            br = new BufferedReader(new InputStreamReader(in));
            //br = new BufferedReader(new InputStreamReader(in,"GBK"));
            bw = new BufferedWriter(new FileWriter("a.txt"));
            // 定义读取数据的行
            String line = null;
            // 读取数据
            while ((line = br.readLine()) != null) {
                // 如果输入的是"exit"就退出
                if("exit".equals(line)){
                    break;
                }
                // 将数据写入文件
                bw.write(line);
                // 写入新的一行
                bw.newLine();
                // 刷新数据缓冲
                bw.flush();
            }
        } catch (Exception e){
            e.printStackTrace();
        } finally {
        // 释放资源
            try {
                if(bw != null)
                    bw.close();
                if (br != null)
                    br.close();
            } catch (IOException e){
```

```
                e.printStackTrace();
            }
        }
    }
}
```

`OutputStreamWriter`是Writer的子类，将输出的字符流转换成字节流。

```
public class OutputStreamWriter  extends Writer
```

- `OutputStreamWriter`是字符的桥梁流以字节流：向其写入的字符编码成使用指定的字节`charset`。
- 它使用的字符集可以由名称指定，也可以被明确指定，或者可以接受平台的默认字符集。

方法	解释
`OutputStreamWriter(OutputStream out)`	创建一个使用默认字符编码的`OutputStreamWriter`。
`OutputStreamWriter(OutputStream out, Charset cs)`	创建一个使用给定字符集的`OutputStreamWriter`。

```
/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: GAO BO
 * Date: 2019-04-04
 * Time: 12:41
 */
public class TestDemo {
    public static void main(String[] args) {
        // 定义字节输出流的对象System.out
        OutputStream out = System.out;
        // 定义字符流的对象
        BufferedReader br = null;
        BufferedWriter bw = null;
        try {
            //通过OutputStreamWriter将字符转流换为字节流对象
            bw = new BufferedWriter(new OutputStreamWriter(out));
            //bw = new BufferedWriter(new OutputStreamWriter(out,"GBK"));
            br = new BufferedReader(new FileReader("a.txt"));
            // 定义读取行的字符串
            String line = null;
            // 读取数据
            while ((line = br.readLine()) != null) {
                // 输出到控制台
                bw.write(line);
                // 新的一行
                bw.newLine();
                // 刷新缓冲
                bw.flush();
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
```

```
            // 释放资源
            try {
                if (bw != null)
                    bw.close();
                if (br != null)
                    br.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

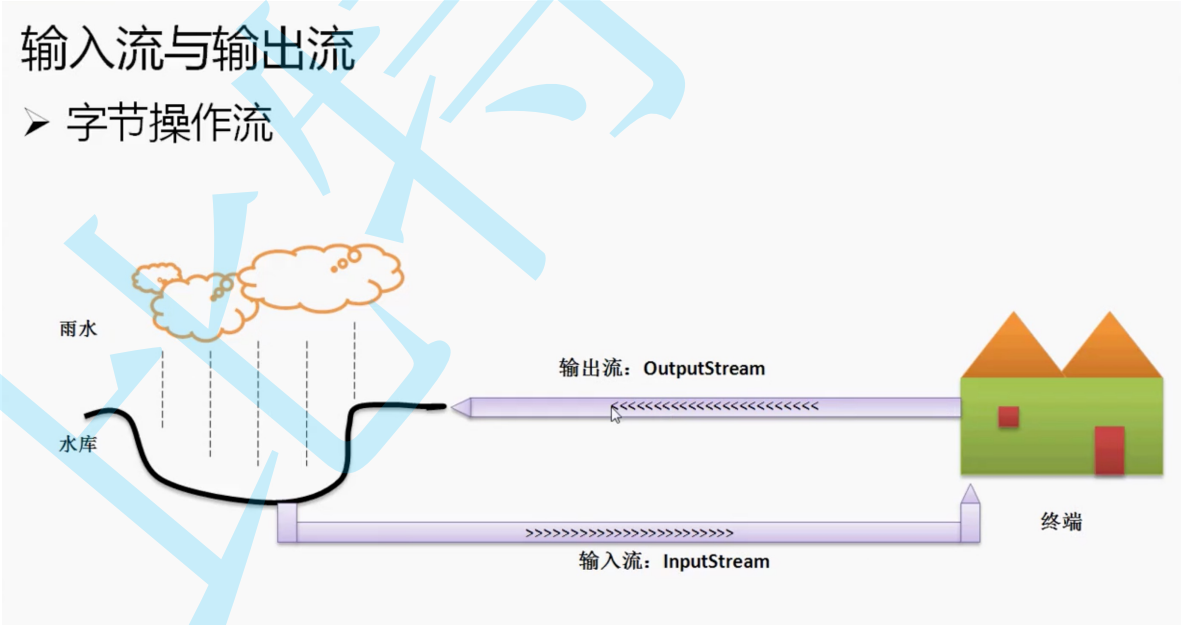
3.8 小结

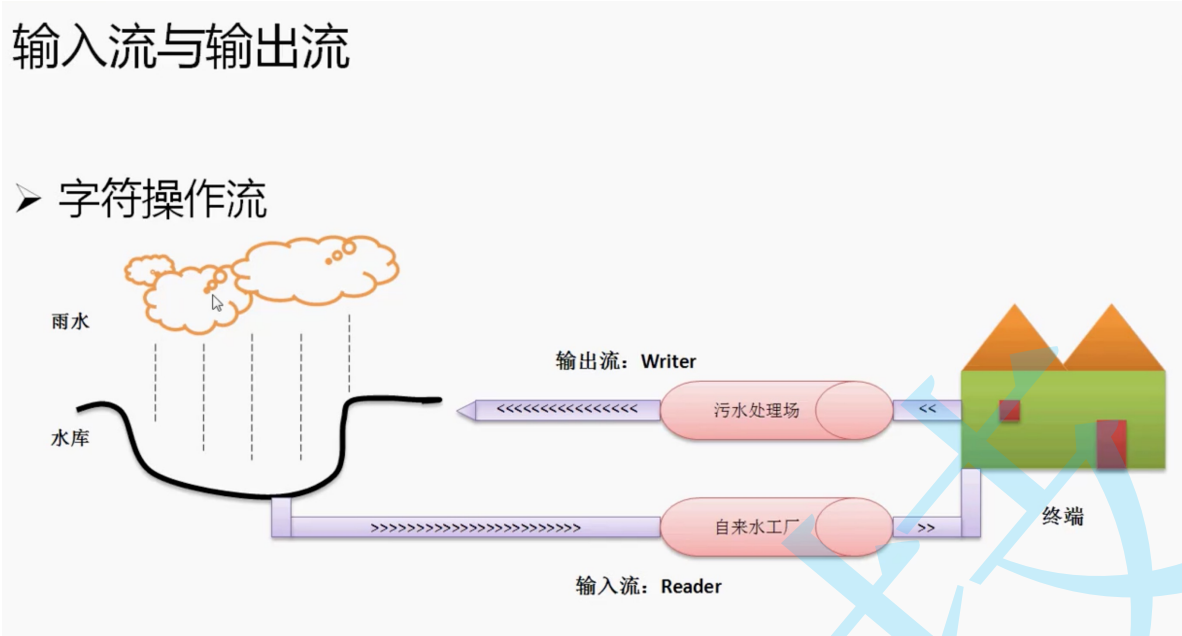
流：在Java中所有数据都是使用流读写的。流是一组有顺序的，有起点和终点的字节集合，是对数据传输的总称或抽象。即数据在两设备间的传输称为流，流的本质是数据传输，根据数据传输特性将流抽象为各种类，方便更直观的进行数据操作。

I/O流的分类

> 1.按照流向分：输入流；输出流
>
> 2.按照处理数据的单位分：字节流(8位的字节)；字符流(16位的字节)
>
> 3.按照流的功能分：
>
> **节点流(低级流)**：可以从一个特定的IO设备上读/写数据的流。(了解)
>
> **处理流(高级流/过滤流)**：是对一个已经存在的流的连接和封装，通过所封装的流的功能调用实现数据读/写操作。**通常处理流的构造器上都会带有一个其他流的参数。**(了解)

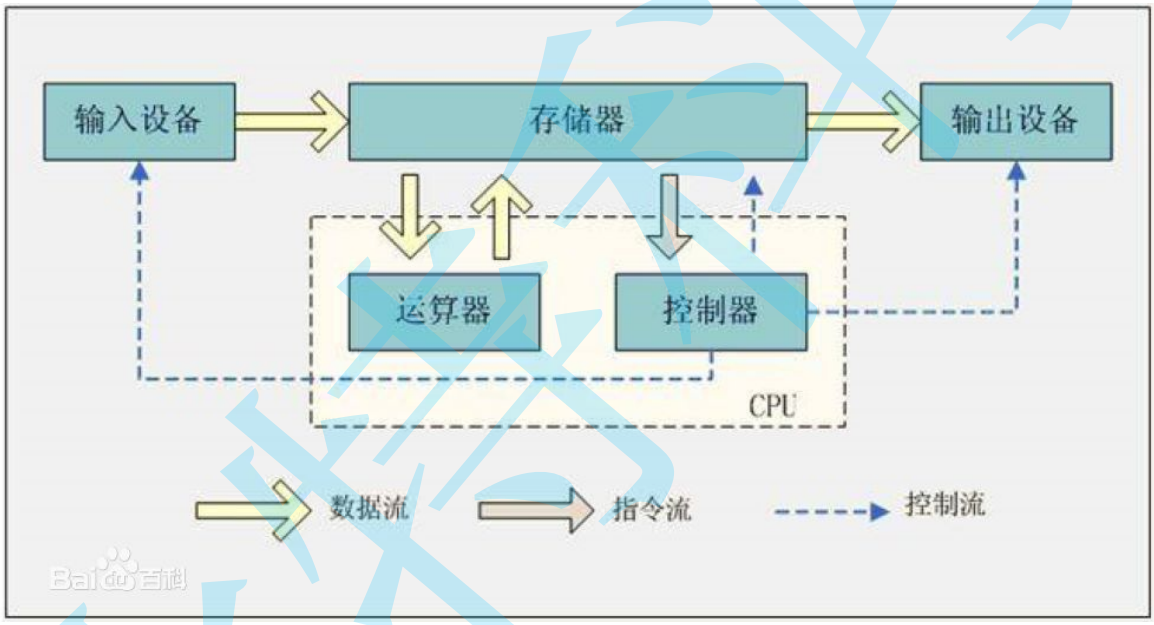
流的作用：为数据源和目的地建立一个输送通道。



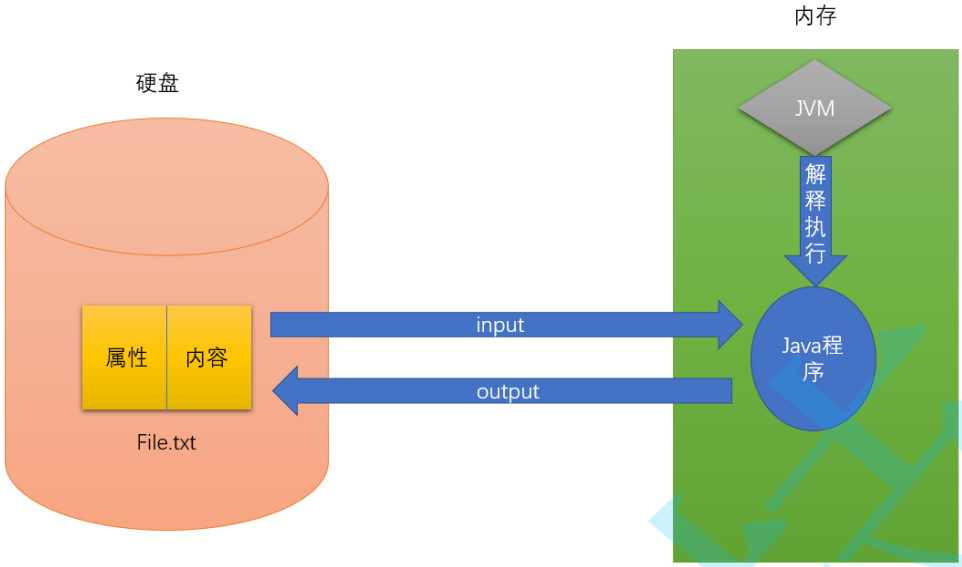


进一步理解冯诺依曼体系：

- 当代PC机基本都遵守冯诺依曼结构



- 计算机硬件结构一共由5大单元组成: 输入设备，输出设备，存储器， CPU运算器，CPU控制器
- 常见输入设备有: 键盘，硬盘，网卡，摄像头等
- 常见的输出设备有: 键盘，显示器，网卡，硬盘等
- 存储器指的就是内存
- 单纯站在数据角度: CPU直接从内存里读取数据，不会和外设打交道，为什么呢？因为硬盘太慢了...
- 所以数据要能够被CPU处理，就必须先被拿到内存，才能够被CPU处理！
- 将数据从输入设备拿到内存，我们称之为`input`，将数据从内存输出到外设，我们称之为`output`，简称这个过程为**IO**
- 那我们经常所说的文件读写，是往哪个硬件设备上读写呢？硬盘！硬盘既可以充当输入设备(input)，又可以充当输出设备(output),注意，`input, output`是站在你的java程序的角度说的！



4. 序列化与反序列化

4.1 什么是序列化和反序列化

序列化：把对象转换为字节序列的过程称为对象的序列化。

反序列化：把字节序列恢复为对象的过程称为对象的反序列化。

有时候我们想把一些信息**持久化**保存起来，那么序列化的意思就是把内存里面的这些对象给变成一连串的字节描述的过程。 常见的就是变成文件。但是问题来了，我就算不序列化，也可以保存到文件当中。有什么问题吗?

4.2 什么时候需要序列化

记住以下几点:

1、把内存中的对象状态保存到一个文件中或者数据库中时候;

2、 用套接字在网络上传送对象的时候;

4.3 实现序列化的方式

一定要牢记实现序列化本身是跟语言无关的:

我们列举出一些其他实现序列化的方式供大家参考:

> 0、Java对象序列化
>
> 1、JSON序列化
>
> 2、XML
>
> 3、Protostuff
>
> 4、Hession（它基于HTTP协议传输，使用Hessian二进制序列化，对于数据包比较大的情况比较友好。）
>
> 5、Dubbo Serialization（阿里dubbo序列化）
>
> 6、FST（高性能、序列化速度大概是JDK的4-10倍，大小是JDK大小的1/3左右）

> 7、自定义协议进行序列化

4.4 如何实现序列化（Java对象序列化）

如下代码，我们自己实现一个Person类，在类当中定义成员变量`name`，`age`，`sex`，`stuId`，`count`,如果要讲Person进行序列化需要实现Serializable接口即可 。

```
import java.io.Serializable
/**
 * Created with IntelliJ IDEA.
 * Description:序列化与反序列化实现
 * User: GAOBO
 * Date: 2019-9-17
 * Time: 15:56
 */
class Person implements Serializable{
    //private static final long serialVersionUID = 1L;
    private String name;
    private int age;
    private String sex;
    //transient修饰的变量，不能被序列化
    transient private int stuId;
    //static修饰的变量，不能被序列化
    private static int count = 99;

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String getSex() {
        return sex;
    }

    public int getStuId() {
        return stuId;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public void setStuId(int stuId) {
        this.stuId = stuId;
    }
```

```
    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", sex='" + sex + '\'' +
                ", stuId=" + stuId +
                ", count=" + count +
                '}';
    }
}
```

序列化反序列化测试：

```
public class TestDemo3 {
    public static void main(String[] args) throws Exception {
        serializePerson();
        Person person = deserializePerson();
        System.out.println(person.toString());
    }
    /**
     * 序列化
     */
    private static void serializePerson() throws IOException {
        Person person = new Person();
        person.setName("bit");
        person.setAge(10);
        person.setSex("男");
        person.setStuId(100);
        // ObjectOutputStream 对象输出流，将 person 对象存储到E盘的
        // person.txt 文件中，完成对 person 对象的序列化操作
        ObjectOutputStream oos = new ObjectOutputStream
                (new FileOutputStream(new File("e:/person.txt")));
        oos.writeObject(person);
        System.out.println("person 对象序列化成功！");
        oos.close();
    }

    /**
     * 反序列化
     */
    private static Person deserializePerson() throws Exception {
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(new
File("e:/person.txt")));
        Person person = (Person) ois.readObject();
        System.out.println("person 对象反序列化成功！");
        return person;
    }
}
```

相关流的说明：

`ObjectOutputStream`代表对象输出流： 它的`writeObject(Object obj)`方法可对参数指定的`obj`对象进行序列化，把得到的字节序列写到一个目标输出流中。`ObjectInputStream`代表对象输入流：它的`readObject()`方法从一个源输入流中读取字节序列，再把它们反序列化为一个对象，并将其返回。

输出结果展示：

```
person 对象序列化成功!
person 对象反序列化成功!
Person{name='bit', age=10, sex='男', stuId=0, count=99}
```

结果分析：

1，他实现了对象的序列化和反序列化。

2，transient 修饰的属性，是不会被序列化的 。我们的 `stuId` 本应是100，可现在为0.**注意**：内置类型为对应0值。引用类型为null;

3、静态变量 好像也被序列化了，但是其实并没有，我们接下来测试一下。

4.5 关于静态数据属性是否被序列化的测试

我们其他代码不变，主函数调用改为如下形式，并且将类中的count变量值改为888：

```
public static void main(String[] args) throws Exception {
    //serializePerson();
    Person person = deserializePerson();
    System.out.println(person.toString());
}
```

运行结果展示：

```
person 对象反序列化成功!
Person{name='bit', age=10, sex='男', stuId=0, count=888}
```

我们可以看到：count的值为888.

因为我们现在的情况是：

1、没有再进行序列化，直接进行反序列化

2、只是改变类中的count变量的值

如果是静态变量参与序列化，那么这个值不应该是888.应该还是上一次序列化的结果99.

结论：

静态变量的值是不会被进行序列化的。

4.6 关于 `serialVersionUID` 的问题

我们将上述代码再执行一遍运行(**序列化和反序列化均要执行**)，接着将Person类中的 `private static final long serialVersionUID = 1L;` 注释取消，然后屏蔽掉序列化的方法：`serializePerson()`，运行主函数。

目的：当我们在类中没有指定 `serialVersionUID` 的时候，编译器会自动赋值，如果序列化是以默认的 `serialVersionUID`，那么反序列化也是会以那个默认的。而我们现在的情况是，以默认的 `serialVersionUID` 进行序列化，以自己赋值的 `serialVersionUID` 进行反序列化，这样代码就会出问题。

代码示例：

```
public static void main(String[] args) throws Exception {
    //serializePerson();
    Person person = deserializePerson();
    System.out.println(person.toString());
}
```

结果展示：

```
Exception in thread "main" java.io.InvalidClassException: Person; local class
incompatible: stream classdesc serialVersionUID = -5
```

注意这个异常：`java.io.InvalidClassException`

4.6 小结

1、一个类如果想被序列化，那么需要实现一个Serializable接口。

2、类中的静态变量的值是不会被进行序列化的，transient 修饰的属性，是不会被序列化的，内置类型为对应0值。引用类型为null;

3、**在实现这个Serializable 接口的时候，一定要给这个 `serialVersionUID` 赋值,最好设置为1L，这个L最好大写来区分，不然小写看起来像是1,不同的 `serialVersionUID` 的值，会影响到反序列化 .**

5.流的使用场景

参考：

6. 字符编码

6.1 常用字符编码

在计算机的世界里面，所有的文字都是通过编码来描述的。对于编码而言，如果没有正确的解码，那么就会产生乱码。

那么要想避免乱码问题，就必须清楚常见的编码有哪些

> 1. GBK、GB2312：表示的是国标编码，GBK包含简体中文和繁体中文，而GB2312只包含简体中文。也就是说，这两种编码都是描述中文的编码。
> 2. UNICODE编码：java提供的16进制编码，可以描述世界上任意的文字信息，但是有个问题，如果现在所有的字母也都使用16进制编码，那么这个编码太庞大了，会造成网络传输的负担。
> 3. ISO8859-1：国际通用编码，但是所有的编码都需要进行转换。
> 4. UTF编码：相当于结合了UNICODE、ISO8859-1，也就是说需要使用到16进制文字使用UNICODE，而如果只是字母就使用ISO8859-1，而常用的就是**UTF-8**编码形式。

在以后的开发之中使用的编码只有一个:UTF-8编码

6.2 乱码产生分析

清楚了常用编码后，下面就可以观察一下乱码的产生。要想观察出乱码，就必须首先知道当前操作系统中默认支持的编码是什么(java默认编码)

范例：读取java运行属性

```
System.getProperties().list(System.out);
```

如果说现在本地系统所用的编码与程序所用编码不同，那么强制转换就会出现乱码。

范例：观察乱码产生

```
package www.bit.java.testthread;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.UnsupportedEncodingException;

public class Test {
    public static void main(String[] args) throws UnsupportedEncodingException,
IOException  {
        OutputStream out = new FileOutputStream(new File("hello.txt")) ;
        out.write("比特欢迎您".getBytes("ISO8859-1")) ;
        out.close();
    }
}
```

乱码产生的本质：编码和解码不统一产生的问题。