# 004.输入和输出流的体系

## java输入/输出流体系中常用的流的分类

| 分类 | 字节输入流 | 字节输出流 | 字符输入流 | 字符输出流 |
|---|---|---|---|---|
| 抽象基类 | InputStream | OutputStream | Reader | Writer |
| 访问文件 | **FileInputStream** | **FileOutputStream** | **FileReader** | **FileWriter** |
| 访问数组 | **ByteArrayInputStream** | **ByteArrayOutputStream** | **CharArrayReader** | **CharArrayWriter** |
| 访问管道 | **PipedInputStream** | **PipedOutputStream** | **PipedReader** | **PipedWriter** |
| 访问字符串 | | | **StringReader** | **StringWriter** |
| 缓冲流 | BufferedInputStream | BufferedOutputStream | BufferedReader | BufferedWriter |
| 转换流 | | | InputStreamReader | OutputStreamWriter |
| 对象流 | ObjectInputStream | ObjectOutputStream | | |
| 抽象基类 | FilterInputStream | FilterOutputStream | FilterReader | FilterWriter |
| 打印流 | | PrintStream | | PrintWriter |
| 推回输入流 | PushbackInputStream | | PushbackReader | |
| 特殊流 | DataInputStream | DataOutputStream | | |

**注：表中粗体字所标出的类代表节点流，必须直接与指定的物理节点关联，**
**表中抽象基类，无法直接创建实例。**

下面是整理出这些IO流的特性及使用方法，只有清楚每个IO流的特性和方法。
才能在不同的需求面前正确的选择对应的IO流进行开发。

## IO体系的基类文件流

## 文件输入流

### FileInputStream和FileReader，

**它们都是节点流——会直接和指定文件关联。**

下面程序示范使用FileInputStream和FileReader。

**使用FileInputStream读取文件：**

```
public class MyClass {
  public  static void main(String[] args)throws IOException{
      FileInputStream fis=null;
      try {
          //创建字节输入流
          fis=new
FileInputStream("E:\\learnproject\\Iotest\\lib\\src\\main\\java\\com\\Test.txt");
          //创建一个长度为1024的竹筒
          byte[] b=new byte[1024];
```

```
            //用于保存的实际字节数
            int hasRead=0;
            //使用循环来重复取水的过程
            while((hasRead=fis.read(b))>0){
                //取出竹筒中的水滴（字节），将字节数组转换成字符串进行输出
              System.out.print(new String(b,0,hasRead));
            }
        }catch (IOException e){
          e.printStackTrace();
        }finally {
            fis.close();
        }
    }
}
```

注：上面程序最后使用了fis.close()来关闭该文件的输入流，程序里面打开的文件IO资源不属于内存的资源，
垃圾回收机制无法回收该资源，所以必须在程序里关闭打开的IO资源。

Java 7改写了所有的IO资源类，它们都实现了AntoCloseable接口，**因此都可以通过自动关闭资源的try语句来关闭这些IO流。**

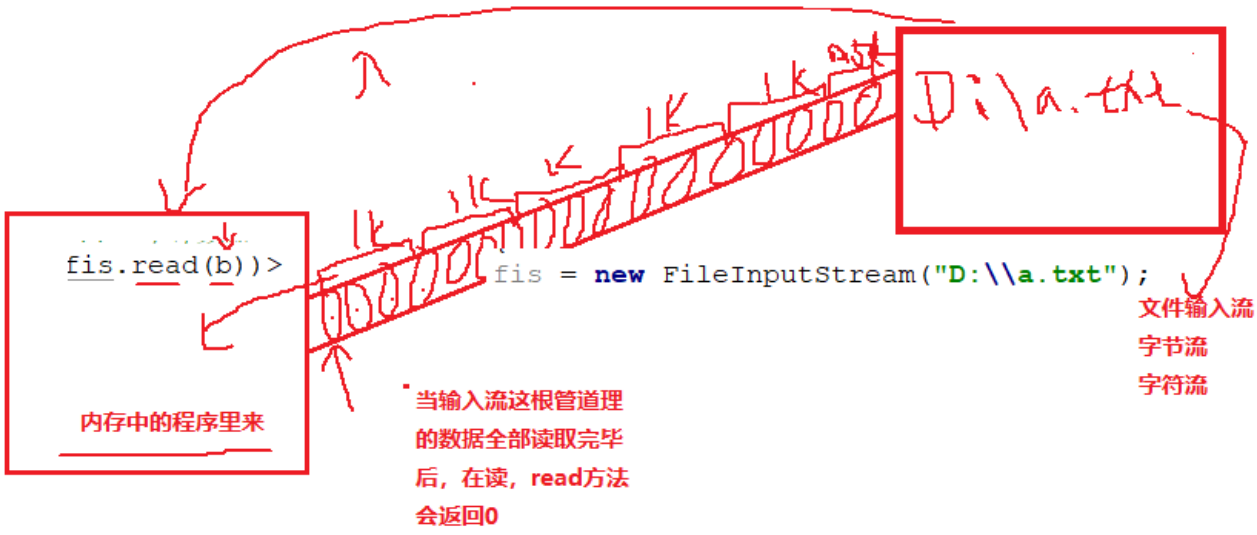
```
public static void main(String[] args) {
    //需要关闭资源的直接写在try（）里面，就不需要手动再去关闭资源了，代码看上去更加简洁
    try (fis=new FileInputStream("E:\\learnproject\\Iotest\\lib\\src\\main\\java\\com\\Test.txt")){

    } catch (IOException e){
        e.printStackTrace();
    }
}
```

**使用FileReader读取文件：**

```
public class FileReaderTest {
    public  static void main(String[] args)throws IOException{
        FileReader fis=null;
        try {
            //创建字符输入流
            fis=new FileReader("E:\\learnproject\\Iotest\\lib\\src\\main\\java\\com\\Test.txt");
            //创建一个长度为1024的竹筒
```

```
            char[] b=new char[1024];
            //用于保存的实际字符数
            int hasRead=0;
            //使用循环来重复取水的过程
            while((hasRead=fis.read(b))>0){
                //取出竹筒中的水滴（字符），将字符数组转换成字符串进行输出
                System.out.print(new String(b,0,hasRead));
            }
        }catch (IOException e){
            e.printStackTrace();
        }finally {
            fis.close();
        }
    }
}
```

可以看出使用FileInputStream和FileReader进行文件的读写并没有什么区别，只是操作单元不同而且。




## 文件输出流

## FileOutputStream/FileWriter

FileOutputStream的用法：

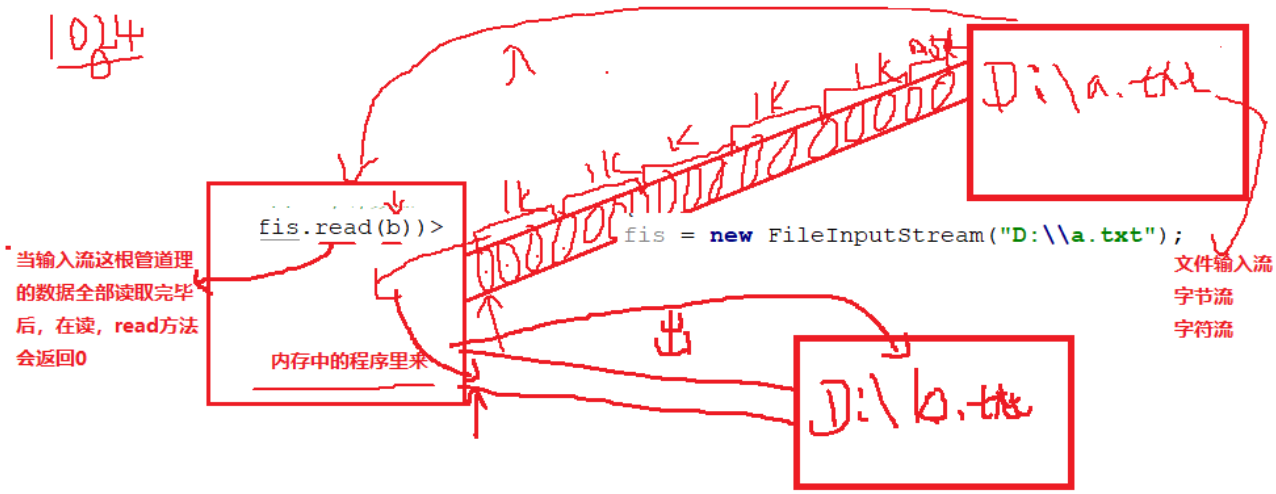
```
public class FileOutputStreamTest {
    public  static void main(String[] args)throws IOException {
        FileInputStream fis=null;
        FileOutputStream fos=null;
        try {
            //创建字节输入流
            fis=new
FileInputStream("E:\\learnproject\\Iotest\\lib\\src\\main\\java\\com\\Test.txt");
```

```
            //创建字节输出流
            fos=new
FileOutputStream("E:\\learnproject\\Iotest\\lib\\src\\main\\java\\com\\newTest.txt")
;

            byte[] b=new byte[1024];
            int hasRead=0;

            //循环从输入流中取出数据
            while((hasRead=fis.read(b))>0){
                //每读取一次，即写入文件输入流，读了多少，就写多少。
                fos.write(b,0,hasRead);
            }
        }catch (IOException e){
            e.printStackTrace();
        }finally {
            fis.close();
            fos.close();
        }
    }
}
```

运行程序可以看到输出流指定的目录下多了一个文件：newTest.txt， 该文件的内容和Test.txt文件的内容完全相同。

FileWriter的使用方式和FileOutputStream基本类似。

注： 使用java的io流执行输出时，不要忘记关闭输出流，关闭输出流除了可以保证流的物理资源被回收之外，还可以将输出流缓冲区中的数据flush到物理节点中里（因为在执行close（）方法之前，自动执行输出流的flush（）方法）。java很多输出流默认都提供了缓存功能，其实我们没有必要刻意去记忆哪些流有缓存功能，哪些流没有，只有正常关闭所有的输出流即可保证程序正常。



## 数组流的使用

字节数组流对象分为输入流和输出流。分别是：ByteArrayInputStream和ByteArrayOutputStream。

字符流和字节流雷同，这里就带过了。字符流分别是：CharArrayReader和CharArrayWriter

数组流：实质上是和内存中的数组相关的一个流，可以将字节数组写到输出流中，也可以将字节数组从输入流中读出来，不涉及磁盘。可以看作是btye[]/char[]和输入/输出流之间的一个数据形式的转换！

### 1.ByteArrayInputStream类

字节数组输入流在内存创建一个字节数组缓冲区，从输入流读取的数据保存在该字节数组缓冲区中。

创建字节数组输入流对象有以下方式：

```
//方法 1
ByteArrayInputStream bArray = new ByteArrayInputStream(byte [] a);
//方法 2
ByteArrayInputStream bArray = new ByteArrayInputStream(byte []a, int off, int len)
```

字节数组输入流对象的方法：

```
public int available()  //显示剩余字节数
```

### 2.ByteArrayOutputStream类

字节数组输出流：在内存中创建一个字节数组缓冲区，所有发送到输出流的数据保存在该字节数组缓冲区中。

创建方式：

```
//方法 1
OutputStream bOut = new ByteArrayOutputStream();
//方法 2
OutputStream bOut = new ByteArrayOutputStream(int a);  //a是制定输出流里的缓冲区大小，本质是内部的数组长度
```

字节数组输出流对象的方法：

```
public byte[] toByteArray()
```

创建一个新分配的字节数组。数组的大小和当前输出流的大小，内容是当前输出流的拷贝。

```
public String toString()
```

将缓冲区的内容转换为字符串，根据平台的默认字符编码将字节转换成字符。

```
public void writeTo(OutputStream outSt)
```

将此字节数组输出流的全部内容写入到指定的输出流参数中。

## 代码演示：

```
private static void inputStreamTest() throws IOException {
    byte[] b1 = new byte[]{1,2,3,4};
    ByteArrayInputStream input = new ByteArrayInputStream(b1);

    System.out.println("剩余字节数: "+input.available());

    byte[] b2 = new byte[2];
    input.read(b2);
    print(b2);
    System.out.println("剩余字节数: " + input.available());

    input.read(b2);
    print(b2);
    System.out.println("剩余字节数: " + input.available() + " 没得读了");

    b2 = new byte[2];
    input.read(b2);
    print(b2);
}

private static void print(byte[] array) {
    System.out.print("长度: " + array.length + " , 内容: ");
    for (byte b : array) {
        System.out.print(b);
    }
    System.out.println();
}
```

**数组输出流，将数据形式转换以后，还是一个可自动扩容的 byte 数组**，默认初始化 32 个字节的大小，最大容量是 2^31-9 个字节（2G）。只要数据不超过2G，都可以往里写。每次写数据之前，会先计算需要的容量大小，如果需要扩容，扩大到 max{原来的两倍，需要的容量大小}

```
//为了验证扩容方式，把其内部缓冲区拿出来
class MyByteArrayOutputStream extends ByteArrayOutputStream {
    public byte[] getBuf() {
        return super.buf;
    }
}

private static void outputStreamTest() throws IOException {

    // 默认缓冲区大小 32 字节
    MyByteArrayOutputStream out = new MyByteArrayOutputStream();
    // 写入 32 个字节，此时
    for(int i = 0; i < 32; i++) {
        out.write(1);
    }
    System.out.println("当前缓冲区长度: " + out.getBuf().length + " 当前数据长度: " +
out.size());

    // 写入 1 个字节，使所需容量为 33 个字节，大于原来 32 字节的容量
    out.write(2);
```

```
    System.out.println("当前缓冲区长度：" + out.getBuf().length + " 当前数据长度：" +
out.size() + " 扩大到原来的两倍了");

    // 取出数据
    byte[] ret = out.toByteArray();
    print(ret);

    // 写入新数据，使其空间正好比原空间的 2 倍大 3 个字节
    out.write(new byte[out.getBuf().length*2-out.size()+3]);
    System.out.println("当前缓冲区长度：" + out.getBuf().length + " 当前数据长度：" +
out.size() + " 扩大到需要的容量大小了");
}
```

## 管道流的使用

管道流是用来在多个线程之间进行信息传递的Java流。

管道流分为字节流管道流和字符管道流。

字节管道流：PipedOutputStream 和 PipedInputStream。

字符管道流：PipedWriter 和 PipedReader。

PipedOutputStream、PipedWriter 是写入者/生产者/发送者；

PipedInputStream、PipedReader 是读取者/消费者/接收者。

这里我们只分析字节管道流，字符管道流原理跟字节管道流一样，只不过底层一个是 byte 数组存储 一个是 char 数组存储的。

java的管道输入与输出实际上使用的是一个循环缓冲数组来实现的。输入流PipedInputStream从这个循环缓冲数组中读数据，输出流PipedOutputStream往这个循环缓冲数组中写入数据。当这个缓冲数组已满的时候，输出流PipedOutputStream所在的线程将阻塞；当这个缓冲数组为空的时候，输入流PipedInputStream所在的线程将阻塞。

在使用管道流之前，需要注意以下要点：

- 管道流仅用于多个线程之间传递信息，若用在同一个线程中可能会造成死锁；
- 管道流的输入输出是成对的，一个输出流只能对应一个输入流，使用构造函数或者connect函数进行连接；
- 一对管道流包含一个缓冲区，其默认值为1024个字节，若要改变缓冲区大小，可以使用带有参数的构造函数；
- 管道的读写操作是互相阻塞的，当缓冲区为空时，读操作阻塞；当缓冲区满时，写操作阻塞；
- 管道依附于线程，因此若线程结束，则虽然管道流对象还在，仍然会报错“read dead end”；
- 管道流的读取方法与普通流不同，只有输出流正确close时，输出流才能读到-1值。

```
public class PipedStreadDemo {
```

```
    public static void main(String[] args) {
        //创建一个线程池
        ExecutorService executorService = Executors.newCachedThreadPool();

        try {
            //创建输入和输出管道流
            PipedOutputStream pos = new PipedOutputStream();
            PipedInputStream pis = new PipedInputStream(pos);

            //创建发送线程和接收线程
            Sender sender = new Sender(pos);
            Reciever reciever = new Reciever(pis);

            //提交给线程池运行发送线程和接收线程
            executorService.execute(sender);
            executorService.execute(reciever);
        } catch (IOException e) {
            e.printStackTrace();
        }
        //通知线程池，不再接受新的任务，并执行完成当前正在运行的线程后关闭线程池。
        executorService.shutdown();
        try {
            //shutdown 后可能正在运行的线程很长时间都运行不完成，这里设置超过1小时，强制执行
Interruptor 结束线程。
            executorService.awaitTermination(1, TimeUnit.HOURS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    static class Sender extends Thread {
        private PipedOutputStream pos;

        public Sender(PipedOutputStream pos) {
            super();
            this.pos = pos;
        }

        @Override
        public void run() {
            try {
                String s = "hello world, amazing java !";
                System.out.println("Sender:" + s);
                byte[] buf = s.getBytes();
                pos.write(buf, 0, buf.length);
                pos.close();
                TimeUnit.SECONDS.sleep(5);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    static class Reciever extends Thread {
        private PipedInputStream pis;

        public Reciever(PipedInputStream pis) {
            super();
            this.pis = pis;
        }

        @Override
```

```
        public void run() {
            try {
                ByteArrayOutputStream baos = new ByteArrayOutputStream();
                byte[] buf = new byte[1024];
                int len = 0;
                while ((len = pis.read(buf)) != -1) {
                    baos.write(buf, 0, len);
                }
                byte[] result = baos.toByteArray();
                String s = new String(result, 0, result.length);
                System.out.println("Reciever:" + s);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

## 字符串流的使用

定义：字符串流，以一个字符为数据源，来构造一个字符流。

作用：在Web开发中，客户端经常要从服务器端上获取数据，

数据返回的格式通过一个字符串（XML、JSON），网络间的数据交互需要用到流，

这个时候我们需要把这个字符串构造为一个字符流，才能在网络上传出去。

**本质上，我们也可以像数组流那样去理解，字符串流就是字符串和流之间的转换！**

### StringWriter特有的方法

返回该字符串缓冲区本身。

public StringBuffer getBuffer()

拼接：将指定的字符序列添加到此 writer。

public StringWriter append(CharSequence csq)

刷新缓存：清理当前编写器的所有缓存区，使所有缓冲数据写入基础设备

public void flush()

**注意：与类型相关的流操作的是内存，内存流的close()是无效的**

### 类java.io.StreamTokenizer

可以获取输入流并将其分析为Token(标记)。

### StringWriter的使用举例：

```
public class StringWriterTest {
    public static void main(String[] args) {
       try {
      //内部维护缓存（string对象）
            StringWriter sw=new StringWriter();
           sw.append("go");
            sw.write("test");
           sw.write('中');
            sw.write("hello".toCharArray());
            sw.flush();
```

```
            //获取缓存中的内容
            System.out.println(sw.toString());
            //获取缓存中的内容，内容装在StringBuffer对像中缓存
            StringBuffer sb=sw.getBuffer();
            //获取StringBuffer中缓存中的内容
            System.out.println(sb.toString());
        } catch (IOException e) {
            // TODO Auto-generated catch block
             e.printStackTrace();
        }
    }
}
```

StringReader的使用举例：

```
public class StringReaderDemo {

    public static void main(String[] args) {
        stringReader();// 输出count = 6
    }

    private static void stringReader() {
        String info = "good good study day day up";
        StringReader sr = new StringReader(info);

        // 流标记器，分析流
        StreamTokenizer st = new StreamTokenizer(sr);
        int count = 0;
        while (st.nextToken() != StreamTokenizer.TT_EOF) {
                //在调用 nextToken 方法之后，ttype字段将包含刚读取的标记的类型
         //ttype字段将包含刚读取的标记的类型,TT_EOF指示已读到流末尾的常量。
            try {
                // 如果是一个单词
                if (st.ttype== StreamTokenizer.TT_WORD) {
          //TT_NUMBER指示已读到一个数字标记的常量 //TT_WORD指示已读到一个文字标记的常量
                    count++;
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        } // while
        sr.close();
        System.out.println("count = " + count);
    }
}
```

## 缓冲流的使用

字节缓冲流的用法

BufferedInputStream继承于FilterInputStream，提供缓冲输入流功能。缓冲输入流相对于普通输入流的优势是，它提供了一个缓冲数组，每次调用read方法的时候，它首先尝试从

缓冲区里读取数据，若读取失败（缓冲区无可读数据），则选择从物理数据源（譬如文件）读取新数据，最后再将缓冲区中的内容部分或全部返回给用户，好处从缓冲区里读取数据远比直接从物理数据源（譬如文件）读取速度快。

**使用举例：**

```
public class BufferedStreamTest {
    public  static void main(String[] args)throws IOException {
        FileInputStream fis=null;
        FileOutputStream fos=null;
        BufferedInputStream bis=null;
        BufferedOutputStream bos=null;
        try {
            //创建字节输入流
            fis=new
FileInputStream("E:\\learnproject\\Iotest\\lib\\src\\main\\java\\com\\Test.txt");
            //创建字节输出流
            fos=new
FileOutputStream("E:\\learnproject\\Iotest\\lib\\src\\main\\java\\com\\newTest.txt")
;
            //创建字节缓存输入流
            bis=new BufferedInputStream(fis);
            //创建字节缓存输出流
            bos=new BufferedOutputStream(fos);

            byte[] b=new byte[1024];
            int hasRead=0;
            //循环从缓存流中读取数据
            while((hasRead=bis.read(b))>0){
                //向缓存流中写入数据，读取多少写入多少
                bos.write(b,0,hasRead);
            }
        }catch (IOException e){
            e.printStackTrace();
        }finally {
            bis.close();
            bos.close();
        }
    }
}
```

可以看到使用字节缓冲流读取和写入数据的方式和文件流（FileInputStream,FileOutputStream）并没有什么不同，

只是把**处理流**套接到**节点流(文件流)**上进行读写。

**注意：**上面代码中我们使用了缓存流和文件流，但是我们只关闭了缓冲流。

这个需要注意一下，当我们使用处理流套接到节点流上的使用的时候，只需要关闭最上层的处理就可以了。

java会自动帮我们关闭下层的节点流。

## 转换流的使用（InputStreamReader/OutputStreamWriter）：

**转换流的特点：**

1. 其是字符流和字节流之间的桥梁
2. 可对读取到的字节数据经过指定编码转换成字符，转换的时候还可以指定我们需要字符的编码方式
3. 可对读取到的字符数据经过指定编码转换成字节

在IDEA中，使用`FileReader` 读取项目中的文本文件。由于IDEA的设置，都是默认的`UTF-8`编码，所以没有任何问题。但是，当读取Windows系统中创建的文本文件时，由于Windows系统的默认是GBK编码，就会出现乱码。

```
public class ReaderDemo {
    public static void main(String[] args) throws IOException {
        FileReader fileReader = new FileReader("E:\\File_GBK.txt");
        int read;
        while ((read = fileReader.read()) != -1) {
            System.out.print((char)read);
        }
        fileReader.close();
    }
}
```

那么如何读取GBK编码的文件呢？

## InputStreamReader类

转换流`java.io.InputStreamReader`，是Reader的子类，是从字节流到字符流的桥梁。它读取字节，并使用指定的字符集将其解码为字符。它的字符集可以由名称指定，也可以接受平台的默认字符集。

## 构造方法

`InputStreamReader(InputStream in)`：创建一个使用默认字符集的字符流。

`InputStreamReader(InputStream in, String charsetName)`：创建一个指定字符集的字符流。

## 指定编码读取

```
public class ReaderDemo2 {
    public static void main(String[] args) throws IOException {
        // 定义文件路径,文件为gbk编码
        String FileName = "E:\\file_gbk.txt";
        // 创建流对象,默认UTF8编码
        InputStreamReader isr = new InputStreamReader(new FileInputStream(FileName));
        // 创建流对象,指定GBK编码
        InputStreamReader isr2 = new InputStreamReader(new FileInputStream(FileName) , "GBK");
        // 定义变量,保存字符
        int read;
        // 使用默认编码字符流读取,乱码
        while ((read = isr.read()) != -1) {
            System.out.print((char)read); //
        }
        isr.close();
```

```
        // 使用指定编码字符流读取,正常解析
        while ((read = isr2.read()) != -1) {
            System.out.print((char)read);// 大家好
        }
        isr2.close();
    }
}
```

## OutputStreamWriter类

转换流`java.io.OutputStreamWriter` ，是Writer的子类，是从字符流到字节流的桥梁。使用指定的字符集讲字符编码为字节。它的字符集可以由名称指定，也可以接受平台的默认字符集。

### 构造方法

`OutputStreamWriter(OutputStream in)`：创建一个使用默认字符集的字符流。

`OutputStreamWriter(OutputStream in, String charsetName)`：创建一个指定字符集的字符流。

## 练习：转换文件编码

将GBK编码的文本文件，转换为UTF-8编码的文本文件。

1. 指定GBK编码的转换流，读取文本文件。
2. 使用UTF-8编码的转换流，写出文本文件。

## 案例实现

```
public class TransDemo {
   public static void main(String[] args) {
        // 1.定义文件路径
        String srcFile = "file_gbk.txt";
        String destFile = "file_utf8.txt";
        // 2.创建流对象
        // 2.1 转换输入流,指定GBK编码
        InputStreamReader isr = new InputStreamReader(new FileInputStream(srcFile) , "GBK");
        // 2.2 转换输出流,默认utf8编码
        OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream(destFile));
        // 3.读写数据
        // 3.1 定义数组
        char[] cbuf = new char[1024];
        // 3.2 定义长度
        int len;
        // 3.3 循环读取
        while ((len = isr.read(cbuf))!=-1) {
            // 循环写出
            osw.write(cbuf,0,len);
        }
        // 4.释放资源
```

```
        osw.close();
        isr.close();
    }
}
```

## 对象流的使用（ObjectInputStream/ObjectOutputStream）的使用：

对象流是对象序列化的工具。

所谓的对象的序列化就是将对象转换成二进制数据流的一种实现手段，通过将对象序列化，可以方便的实现对象的传输及保存。

### 写入对象：

```
 public static void writeObject(){
        OutputStream outputStream=null;
        BufferedOutputStream buf=null;
        ObjectOutputStream obj=null;
        try {
            //序列化文件輸出流
            outputStream=new
FileOutputStream("E:\\learnproject\\Iotest\\lib\\src\\main\\java\\com\\myfile.tmp");
            //构建缓冲流
            buf=new BufferedOutputStream(outputStream);
            //构建字符输出的对象流
            obj=new ObjectOutputStream(buf);
            //序列化数据写入
            obj.writeObject(new Person("A", 21));//Person对象
            //关闭流
            obj.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
}
```

### 读取对象：

```
    public static void readObject() throws IOException {
        try {
            InputStream inputStream=new
FileInputStream("E:\\learnproject\\Iotest\\lib\\src\\main\\java\\com\\myfile.tmp");
            //构建缓冲流
            BufferedInputStream buf=new BufferedInputStream(inputStream);
            //构建字符输入的对象流
            ObjectInputStream obj=new ObjectInputStream(buf);
            Person tempPerson=(Person)obj.readObject();
            System.out.println("Person对象为："+tempPerson);
            //关闭流
            obj.close();
            buf.close();
            inputStream.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
```

### 一些注意事项 ：

1.读取顺序和写入顺序一定要一致，不然会读取出错。

2.在对象属性前面加transient关键字，则该对象的属性不会被序列化。

## 抽象基类FilterInputStream和FilterOutputStream

**FilterInputStream** 的作用是用来“封装其它的输入流，并为它们提供额外的功能”。

它的常用的子类有BufferedInputStream和DataInputStream。

(1) BufferedInputStream的作用就是为**“输入流提供缓冲功能，以及mark()和reset()功能”。**

### InputStream和Reader提供的一些移动指针的方法：

- void mark(int readlimit )；在记录指针当前位置记录一个标记（mark）。
- boolean markSupported()；判断此输入流是否支持mark()操作，即是否支持记录标记。
- void reset()；将此流的记录指针重新定位到上一次记录标记（mark）的位置。
- long skip(long n)；记录指针向前移动n个字节/字符。

*readlimit 参数给出当前输入流在标记位置变为非法前允许读取的字节数。*

这句话的意思是说：mark就像书签一样，用于标记，以后再调用reset时就可以再回到这个mark过的地方。mark方法有个参数，通过这个整型参数，告诉系统，希望在读出多少个字符之前，这个mark保持有效。

比如说mark(10)，那么在read()10个以内的字符时，reset()操作指针可以回到标记的地方，然后重新读取已经读过的数据，如果已经读取的数据超过10个，那reset()操作后，就不能正确读取以前的数据了，mark()打标记已经失效，reset()会报错。

但实际的运行情况却和JAVA文档中的描述并不完全相符。 有时候在BufferedInputStream类中调用mark(int readlimit)方法后，即使读取超过readlimit字节的数据，mark标记仍可能有效，仍然能正确调用reset方法重置。

事实上，mark在JAVA中的实现是和缓冲区相关的。只要缓冲区够大，mark后读取的数据没有超出缓冲区的大小，mark标记就不会失效。如果不够大，mark后又读取了大量的数据，导致缓冲区更新，原来标记的位置自然找不到了。

因此，mark后读取多少字节才失效，并不完全由readlimit参数确定，也和BufferedInputStream类的缓冲区大小有关。 如果BufferedInputStream类的缓冲区大小大于readlimit，在mark以后只有读取超过缓冲区大小的数据，mark标记才会失效。

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

        try {
            // 初始化一个字节数组，内有5个字节的数据
            byte[] bytes={1,2,3,4,5};
```

```
            // 用一个ByteArrayInputStream来读取这个字节数组
            ByteArrayInputStream in=new ByteArrayInputStream(bytes);
            // 将ByteArrayInputStream包含在一个BufferedInputStream，并初始化缓冲区大小为2。
            BufferedInputStream bis=new BufferedInputStream(in,2);
            // 读取字节1
            System.out.print(bis.read()+",");
            // 在字节2处做标记，同时设置readlimit参数为1
            // 根据JAVA文档mark以后最多只能读取1个字节，否则mark标记失效，但实际运行结果不是
这样
            System.out.println("mark");
            bis.mark(1);

            /*
             * 连续读取两个字节，超过了readlimit的大小，mark标记仍有效
             */
            // 连续读取两个字节
            System.out.print(bis.read()+",");
            System.out.print(bis.read()+",");
            // 调用reset方法，未发生异常，说明mark标记仍有效。
            // 因为，虽然readlimit参数为1，但是这个BufferedInputStream类的缓冲区大小为2，
            // 所以允许读取2字节
            System.out.println("reset");
            bis.reset();

            /*
             * 连续读取3个字节，超过了缓冲区大小，mark标记失效。
             * 在这个例子中BufferedInputStream类的缓冲区大小大于readlimit,
             * mark标记由缓冲区大小决定
             */
            // reset重置后连续读取3个字节，超过了BufferedInputStream类的缓冲区大小
            System.out.print(bis.read()+",");
            System.out.print(bis.read()+",");
            System.out.print(bis.read()+",");
            // 再次调用reset重置，抛出异常，说明mark后读取3个字节，mark标记失效
            System.out.println("reset again");
            bis.reset();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

(2) DataInputStream 是用来装饰其它输入流，它**“允许应用程序以与机器无关方式从底层输入流中读取基本 Java 数据类型”。**

应用程序可以使用DataOutputStream(数据输出流)写入由DataInputStream(数据输入流)读取的数据。

FilterOutputStream 的作用是用来**“封装其它的输出流，并为它们提供额外的功能”。**

它主要包括BufferedOutputStream, DataOutputStream和PrintStream。

(1) BufferedOutputStream的作用就是为“输出流提供缓冲功能”。

(2) DataOutputStream 是用来装饰其它输出流，将DataOutputStream和DataInputStream输入流配合使用，

“允许应用程序以与机器无关方式从底层输入流中读写基本 Java 数据类型”。

(3) PrintStream 是用来装饰其它输出流。它能为其他输出流添加了功能，使它们能够方便地打印各种数据值表示形式。

## 抽象基类FilterReader和FilterWriter

字符过滤输入/出流、本质是一个抽象类、为所有装饰类提供一个标准、只是简单重写了父类Reader/Wtiter的所有方法、要求子类必须重写核心方法、和提供具有自己特色的方法、这里没有像字节流那样有很多的子类来实现不同的功能、可能是因为字符流本来就是字节流的一种装饰、所以在这里没有必要再对其进行装饰、只是提供一个扩展的接口而已。

## 打印流的使用

打印流是输出信息最方便的类，注意包含字节打印流PrintStream和字符打印流PrintWriter。

打印流提供了非常方便的打印功能，可以打印任何类型的数据信息，例如：小数，整数，字符串。

之前打印信息需要使用OutputStream但是这样，所有数据输出会非常麻烦，PrintStream可以把OutputStream类重新包装了一下，使之输出更方便。

### 第一个例子：

```
public static void main(String arg[]) throws Exception{
    PrintStream ps = null ;         // 声明打印流对象
    // 如果现在是使用FileOuputStream实例化，意味着所有的输出是向文件之中
    ps = new PrintStream(new FileOutputStream(new File("d:" +
File.separator + "test.txt"))) ;
    ps.print("hello ") ;
    ps.println("world!!!") ;
    ps.print("1 + 1 = " + 2) ;
    ps.close() ;
}
```

### 格式化输出：

JAVA对PrintStream功能进行了扩充，增加了格式化输出功能。直接使用Print即可。但是输出的时候需要指定输出的数据类型。

| 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 | %f | 表示内容为小数 |
| 4 | %c | 表示内容为字符 |

### 第二个例子：

```
public static void main(String arg[]) throws Exception{
    PrintStream ps = null ;         // 声明打印流对象
    // 如果现在是使用FileOuputStream实例化，意味着所有的输出是向文件之中
    ps = new PrintStream(new FileOutputStream(new File("d:" +
File.separator + "test.txt"))) ;
```

```
    String name = "xxxx" ;     // 定义字符串
    int age = 30 ;                  // 定义整数
    float score = 990.356f ;    // 定义小数
    char sex = 'M' ;              // 定义字符
    ps.printf("姓名：%s；年龄：%d；成绩：%f；性别：%c",name,age,score,sex)
;
    ps.close() ;
}
```

如果觉得要使用很多%s,%d,%c无法记住的话，实例可以全部使用“%s”表示。

第三个例子：

```
public static void main(String arg[]) throws Exception{
    PrintStream ps = null ;         // 声明打印流对象
    // 如果现在是使用FileOuputStream实例化，意味着所有的输出是向文件之中
    ps = new PrintStream(new FileOutputStream(new File("d:" +
File.separator + "test.txt"))) ;
    String name = "xxxx" ;     // 定义字符串
    int age = 30 ;                  // 定义整数
    float score = 990.356f ;    // 定义小数
    char sex = 'M' ;              // 定义字符
    ps.printf("姓名：%s；年龄：%s；成绩：%s；性别：%s",name,age,score,sex)
;
    ps.close() ;
}
```

| 分类 | 字节输入流 | 字节输出流 | 字符输入流 | 字符输出流 |
| --- | --- | --- | --- | --- |
| 抽象基类 | InputStream | OutputStream | Reader | Writer |
| 访问文件 | **FileInputStream** | **FileOutputStream** | **FileReader** | **FileWriter** |
| 访问数组 | **ByteArrayInputStream** | **ByteArrayOutputStream** | **CharArrayReader** | **CharArrayWriter** |
| 访问管道 | **PipedInputStream** | **PipedOutputStream** | **PipedReader** | **PipedWriter** |
| 访问字符串 | | | **StringReader** | **StringWriter** |
| 缓冲流 | BufferedInputStream | BufferedOutputStream | BufferedReader | BufferedWriter |
| 转换流 | | | InputStreamReader | OutputStreamWriter |
| 对象流 | ObjectInputStream | ObjectOutputStream | | |
| 抽象基类 | FilterInputStream | FilterOutputStream | FilterReader | FilterWriter |
| 打印流 | | PrintStream | | PrintWriter |
| 推回输入流 | PushbackInputStream | | PushbackReader | |
| 特殊流 | DataInputStream | DataOutputStream | | |

## 推回输入流的使用

通常我们使用输入流的时候，我们读取输入流是顺序读取，**当读取的不是我们想要的怎么办，又不能放回去，**

虽然我们可以使用程序做其他的处理来解决，**但是Java提供了推回输入流来解决这个问**

题，

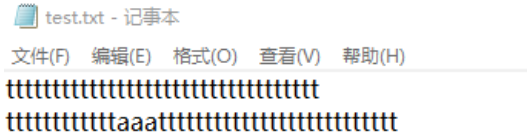
**推回输入流可以做这样子的事情：将已经读取出来的字节或字符数组内容推回到推回缓冲区里面，**

**从而允许重复读取刚刚读取的我们不想要的东西之前内容**

**注意：**当程序创建一个推回输入流时需要指定推回缓冲区的大小，默认的推回缓冲区长度为一，

如果程序推回到推回缓冲区的内容超出了推回缓冲区的大小，将会引发Pushback buffer overflow 异常。

**程序举例：**




```
public class IOTest13 {
    public static void main(String[] args) throws IOException {
        //文件流，输入流，字符流
        Reader reader = new FileReader("d:\\aa.txt");
        //想要支持推回功能，把文件流进一步封装为推回输入流
        PushbackReader pr = new PushbackReader(reader,1024);
        //从输入流中，读取数据
        char[] c = new char[5];
        int hasReadCount = 0;
        String sumString = "";
        while ((hasReadCount=pr.read(c))!=-1){
            //真正的从输入流里获取数据
            //本次循环获取的的数据,转换成字符串
            String curString = new String(c,0,hasReadCount);
            sumString = sumString + curString;

            //sumString获取来的所有字符，差有没有，我不想要的数据的标记aaa
            int aaaIndex = sumString.indexOf("aaa");
            if(aaaIndex>-1){
                //证明读到了aaa,那么我们就要把数据推回去
                pr.unread(sumString.toCharArray());
                //unread，把所有的内容都还回去了
                //重新把我要的内容读出来
                if(aaaIndex > 5){
                    c = new char[aaaIndex];
                }
                pr.read(c,0,c.length);
                //现在c里就是我们要的内容
                System.out.println("我们的要的内容：" + new String(c));
                //我要的内容都拿到了，后面都是我不要的内容，循环还有意义
                break;
            }else{
                //读到的东西输出
                System.out.println(new String(c));
            }
        }
    }
}
```

**推回输入流在进行取数据的时后先从推回缓冲区读取，当推回缓冲区中的数据读取完了之后才从输入流中读取数据。**

| 分类 | 字节输入流 | 字节输出流 | 字符输入流 | 字符输出流 |
|---|---|---|---|---|
| 抽象基类 | InputStream | OutputStream | Reader | Writer |
| 访问文件 | **FileInputStream** | **FileOutputStream** | **FileReader** | **FileWriter** |
| 访问数组 | **ByteArrayInputStream** | **ByteArrayOutputStream** | **CharArrayReader** | **CharArrayWriter** |
| 访问管道 | **PipedInputStream** | **PipedOutputStream** | **PipedReader** | **PipedWriter** |
| 访问字符串 | | | **StringReader** | **StringWriter** |
| 缓冲流 | BufferedInputStream | BufferedOutputStream | BufferedReader | BufferedWriter |
| 转换流 | | | InputStreamReader | OutputStreamWriter |
| 对象流 | ObjectInputStream | ObjectOutputStream | | |
| 抽象基类 | FilterInputStream | FilterOutputStream | FilterReader | FilterWriter |
| 打印流 | | PrintStream | | PrintWriter |
| 推回输入流 | PushbackInputStream | | PushbackReader | |
| 特殊流 | DataInputStream | DataOutputStream | | |

## 数据流DataInputStream和DataOutputStream的使用

DataOutputStream数据输出流允许应用程序将基本Java数据类型写到基础输出流中，而DataInputStream数据输入流允许应用程序以机器无关的方式从底层输入流中读取基本的Java类型.

### DataInputStream的内部方法：

`read(byte b[])`---从数据输入流读取数据存储到字节数组b中.

`read(byte b[],int off,in len)`---从数据输入流中读取数据存储到数组b里面,位置从off开始,长度为len个字节.

`readFully(byte b[])`---从数据输入流中循环读取b.length个字节到数组b中.

`readFully(byte b[],int off,in len )`---从数据输入流中循环读取len个字节到字节数组b中.从b的off位置开始放

`skipBytes(int b)`---跳过n个字节.

`readBoolean()`---从数据输入流读取布尔类型的值.

`readByte()`---从数据输入流中读取一个字节.

`readUnsignedByte()`---从数据输入流中读取一个无符号的字节,返回值转换成int类型.

`readShort()`---从数据输入流读取一个short类型数据.

`readUnsignedShort()`---从数据输入流读取一个无符号的short类型数据.

`readChar()`---从数据输入流中读取一个字符数据

`readInt()`---从数据输入流中读取一个int类型数据.

readLong()---从数据输入流中读取一个long类型的数据.

readFloat()---从数据输入流中读取一个float类型的数据.

readDouble()---从数据输入流中读取一个double类型的数据.

readUTF()---从数据输入流中读取用UTF-8格式编码的UniCode字符格式的字符串.

### DataOutputStream的内部方法

intCount(int value)---数据输出流增加的字节数.

write(int b)---将int类型的b写到数据输出流中.

write(byte b[],int off, int len)---将字节数组b中off位置开始,len个长度写到数据输出流中.

flush()---刷新数据输出流.

writeBoolean()---将布尔类型的数据写到数据输出流中,底层是转化成一个字节写到基础输出流中.

writeByte(int v)---将一个字节写到数据输出流中(实际是基础输出流).

writeShort(int v)---将一个short类型的数据写到数据输出流中,底层将v转换2个字节写到基础输出流中.

writeChar(int v)---将一个charl类型的数据写到数据输出流中,底层是将v转换成2个字节写到基础输出流中.

writeInt(int v)---将一个int类型的数据写到数据输出流中,底层将4个字节写到基础输出流中.

writeLong(long v)---将一个long类型的数据写到数据输出流中,底层将8个字节写到基础输出流中.

writeFloat(flloat v)---将一个float类型的数据写到数据输出流中,底层会将float转换成int类型,写到基础输出流中.

writeDouble(double v)---将一个double类型的数据写到数据输出流中,底层会将double转换成long类型,写到基础输出流中.

writeBytes(String s)---将字符串按照字节顺序写到基础输出流中.

writeChars(String s)---将字符串按照字符顺序写到基础输出流中.

writeUTF(String str)---以机器无关的方式使用utf-8编码方式将字符串写到基础输出流中.

size()---写到数据输出流中的字节数.

**程序举例:**

```
public static void main(String[] args) throws IOException {
    DataOutputStream dos = new DataOutputStream(new
FileOutputStream("D:\\java.txt"));
    dos.writeUTF("α");
    dos.writeInt(1234567);
    dos.writeBoolean(true);
    dos.writeShort((short)123);
```

```
    dos.writeLong((long)456);
    dos.writeDouble(99.98);
    DataInputStream dis = new DataInputStream(new
FileInputStream("D:\\java.txt"));
    System.out.println(dis.readUTF());
    System.out.println(dis.readInt());
    System.out.println(dis.readBoolean());
    System.out.println(dis.readShort());
    System.out.println(dis.readLong());
    System.out.println(dis.readDouble());
    dis.close();
    dos.close();
}
```

### 在开发中正确使用Io流

了解了java io的整体类结构和每个类的一下特性后，我们可以在开发的过程中根据需要灵活的使用不同的Io流进行开发。

下面是整理的2点原则:

- 如果是操作二进制文件那我们就使用字节流，如果操作的是文本文件那我们就使用字符流。
- 尽可能的多使用处理流，这会使我们的代码更加灵活，复用性更好。