

# 第十一章 File类、IO流

## 1 File类

### 1.1 概述

`java.io.File` 类是文件和目录路径名的抽象表示，主要用于文件和目录的创建、查找和删除等操作。

#### 1) 构造方法

```
1  package java.io;
2
3  public class File
4      implements Serializable, Comparable<File>
5  {
6      //通过将给定路径名字符串来创建新的 File实例
7      public File(String pathname) {
8          if (pathname == null) {
9              throw new NullPointerException();
10         }
11         this.path = fs.normalize(pathname);
12         this.prefixLength = fs.prefixLength(this.path);
13     }
14
15     //从**父抽象路径名和子路径名字符串**创建新的 File实例
16     public File(String parent, String child) {
17         //省略...
18     }
19
20     public File(File parent, String child) {
21         //省略...
22     }
```

```
23
24     //省略...
25 }
```

## 2) 实例对象

```
1  package com.briup.chap11.test;
2
3  import java.io.File;
4
5  public class Test011_File {
6      public static void main(String[] args) {
7          // 文件路径名
8          String pathname = "D:\\aaa.txt";
9          File file1 = new File(pathname);
10
11         // 文件路径名
12         String pathname2 = "D:\\aaa\\bbb.txt";
13         File file2 = new File(pathname2);
14
15         // 通过父路径和子路径字符串
16         String parent = "D:\\aaa";
17         String child1 = "bbb.txt";
18         File file3 = new File(parent, child1);
19
20         // 通过父级File对象和子路径字符串
21         File parentDir = new File("D:\\aaa");
22         String child2 = "bbb.txt";
23         File file4 = new File(parentDir, child2);
24     }
25 }
```

小贴士:

1. 一个File对象代表硬盘中实际存在的一个文件或者目录。
2. 无论该路径下是否存在文件或者目录，都不影响File对象的创建。

## 1.2 使用

### 1) 路径获取

```
1 //File绝对路径名字符串
2 public String getAbsolutePath();
3 //File文件构造路径
4 public String getPath();
5 //File文件或目录的名称
6 public String getName();
7 //File文件或目录的长度
8 public long length();
```

案例如下：

```
1 package com.briup.chap11.test;
2
3 import java.io.File;
4
5 public class Test012_File {
6     public static void main(String[] args) {
7         //针对文件
8         File f = new File("D:/aaa/Test1101_File.java");
9         //返回此File的绝对路径名字符串
10        System.out.println("文件绝对路径:" +
11        f.getAbsolutePath());
12        //将此File转换为路径名字符串
13        System.out.println("文件构造路径:" + f.getPath());
14        //返回由此File表示的文件或目录的名称
```

```

14      System.out.println("文件名称:" + f.getName());
15      //返回由此File表示的文件的长度
16      System.out.println("文件长度:" + f.length() + "字节");
17      //针对目录
18      File f2 = new File("D:/aaa");
19      System.out.println("目录绝对路径:" +
    f2.getAbsolutePath());
20      System.out.println("目录构造路径:" + f2.getPath());
21      System.out.println("目录名称:" + f2.getName());
22      System.out.println("目录长度:" + f2.length());
23  }
24  }
25
26  //输出结果
27  文件绝对路径:D:\aaa\Test1101_File.java
28  文件构造路径:D:\aaa\Test1101_File.java
29  文件名称:Test1101_File.java
30  文件长度:650字节
31  目录绝对路径:D:\aaa
32  目录构造路径:D:\aaa
33  目录名称:aaa
34  目录长度:0

```

API中说明：length()，表示文件的长度。但是File对象表示目录，则返回值未指定。

## 2) 路径操作

- **绝对路径**：从盘符开始的路径，这是一个完整的路径。
- **相对路径**：相对于项目目录的路径，这是一个便捷的路径，开发中经常使用。

## 案例描述：

- 1 案例描述：输入具体文件路径，以及只输入文件名字，通过File的绝对路径方法，验证结果

案例如下：

```
1 package com.briup.chap11.test;
2
3 import java.io.File;
4
5 public class Test013_File {
6     public static void main(String[] args) {
7         // D盘下的Test1101_File.java文件
8         File f = new File("D:\\Test1101_File.java");
9         System.out.println(f.getAbsolutePath());
10
11         // 项目下的Test1101_File.java文件（不包含包名）
12         File f2 = new File("Test1101_File.java");
13         System.out.println(f2.getAbsolutePath());
14     }
15 }
16
17 //输出结果：
18 D:\\Test1101_File.java
19 E:\\work\\2023-CoreJava\\Test1101_File.java
```

### 3) 判断操作

```
1 //判断文件或目录是否存在
2 public boolean exists();
3 //判断是否是文件
4 public boolean isFile();
5 //判断是否是目录
6 public boolean isDirectory();
```

案例如下：

```

1  package com.briup.chap11.test;
2
3  import java.io.File;
4
5  public class Test014_File {
6      public static void main(String[] args) {
7          File f = new File("D:\\aaa\\bbb.java");
8          File f2 = new File("D:\\aaa");
9          // 判断是否存在
10         System.out.println("D:\\aaa\\bbb.java 是否存在:"
+ f.exists());
11         System.out.println("D:\\aaa 是否存在:" + f2.exists());
12         // 判断是文件还是目录
13         System.out.println("D:\\aaa 文件?:" + f2.isFile());
14         System.out.println("D:\\aaa 目录?:" + f2.isDirectory());
15     }
16 }
17
18 //输出结果:
19 D:\\aaa\\bbb.java 是否存在:true
20 D:\\aaa 是否存在:true
21 D:\\aaa 文件?:false
22 D:\\aaa 目录?:true

```

#### 4) 创建删除操作

```

1  //当且仅当具有该名称的文件尚不存在时，创建一个新的空文件
2  public boolean createNewFile();
3  //创建目录
4  public boolean mkdir();
5  //创建多级目录
6  public boolean mkdirs();
7  //文件或目录的删除
8  public boolean delete();

```

## 案例如下:

```
1  package com.briup.chap11.test;
2
3  import java.io.File;
4  import java.io.IOException;
5
6  public class Test015_File {
7      public static void main(String[] args) {
8          try {
9              // 文件的创建
10             File f = new File("aaa.txt");
11             System.out.println("是否存在:" + f.exists()); //
false
12             //当且仅当具有该名称的文件尚不存在时, 创建一个新的空文件
13             System.out.println("是否创建:" +
f.createNewFile()); // true
14             System.out.println("是否存在:" + f.exists()); //
true
15
16             // 目录的创建(创建由此File表示的目录)
17             File f2 = new File("newDir");
18             System.out.println("是否存在:" + f2.exists()); //
false
19             System.out.println("是否创建:" + f2.mkdir()); //
true
20             System.out.println("是否存在:" + f2.exists()); //
true
21
22             // 创建多级目录(创建由此File表示的目录, 包括任何必需但不
存在的父目录)
23             File f3 = new File("newDira\\newDirb");
24             System.out.println(f3.mkdir()); // false
25             File f4 = new File("newDira\\newDirb");
26             System.out.println(f4.mkdirs()); // true
27
```

```

28         // 文件的删除
29         System.out.println(f.delete()); // true
30
31         // 目录的删除
32         System.out.println(f2.delete()); // true
33         System.out.println(f4.delete()); // false
34     } catch (IOException e) {
35         // TODO Auto-generated catch block
36         e.printStackTrace();
37     }
38 }
39 }

```

API中说明：delete方法，如果此File表示目录，则目录必须为空才能删除。

## 5) 目录遍历操作

```

1  package java.io;
2
3  public class File
4      implements Serializable, Comparable<File>
5  {
6      //省略...
7
8      //目录文件调用该方法，获取目录中所有子文件名，返回String数组
9      //其他文件调用该方法，返回null
10     public String[] list();
11
12     //目录文件调用该方法，获取目录中所有子文件，返回File数组
13     //其他文件调用该方法，返回null
14     public File[] listFiles();
15
16     //目录文件调用该方法，获取目录中符合筛选条件的子文件，返回File数组
17     //其他文件调用该方法，返回null
18     public File[] listFiles(FileFilter filter);
19

```



```
20      //省略...
21  }
```

## 案例展示:

准备目录 `D:\test` , 放入各类文件, 编码对其遍历。

窗 > 学习 (D:) > test

名称	修改日期	类型	大小
13-类加载、反射	2023/8/17 19:36	文件夹	
readme	2023/8/17 19:36	文件夹	
01-Java基础入门.pdf	2023/7/19 23:55	PDF Document	4,732 KB
01-Java基础入门思路.mp4	2023/7/19 17:55	MP4 - MPEG-4 ...	72,452 KB
11-File、IO流.pdf	2023/8/16 17:37	PDF Document	1,053 KB
readme.pdf	2023/8/15 20:21	PDF Document	574 KB

```
1  package com.briup.chap11.test;
2
3  public class Test016_File {
4      public static void main(String[] args) {
5          //1.准备目录文件
6          String dirPath = "D:\\test";
7          File dirFile = new File(dirPath);
8
9          //2.获取目录中所有子文件名称, 并遍历输出
10         String[] list = dirFile.list();
11         for (String s : list) {
12             System.out.println(s);
13         }
14
15         System.out.println("-----");
16
17         //3.准备普通文件
18         String fileName = "readme.pdf";
19         File file = new File(dirFile, fileName);
20
21         //4.普通文件调用list返回null
22         String[] list2 = file.list();
```

```

23         System.out.println(list2);
24
25         System.out.println("-----");
26
27         //5.获取目录中所有子文件对象，并遍历输出
28         File[] listFiles = dirFile.listFiles();
29         for (File f : listFiles) {
30             System.out.println(f);
31         }
32
33         System.out.println("-----");
34
35         //6.使用文件过滤器，获取目录下所有普通文件，并遍历输出
36         File[] listFiles2 = dirFile.listFiles(new FileFilter()
37         {
38             @Override
39             public boolean accept(File f) {
40                 if(f.isFile())
41                     return true;
42                 return false;
43             }
44         });
45
46         for (File f : listFiles2) {
47             //输出文件名即可
48             System.out.println(f.getName());
49         }
50     }

```

输出结果：

```
Problems Javadoc Declaration Console Progress Properties Navigator (Deprecated) Git Staging Git Reflog History Servers
<terminated> Test016_File [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年8月17日 下午7:42:49 - 下午7:42:51)
01-Java基础入门.pdf
01-Java基础入门思路.mp4
11-File、IO流.pdf      输出子文件名(含目录)
13-类加载、反射
readme
readme.pdf
-----
null
-----
D:\test\01-Java基础入门.pdf
D:\test\01-Java基础入门思路.mp4
D:\test\11-File、IO流.pdf
D:\test\13-类加载、反射      输出File对象
D:\test\readme
D:\test\readme.pdf
-----
01-Java基础入门.pdf
01-Java基础入门思路.mp4
11-File、IO流.pdf      过滤后，输出文件名
readme.pdf
```

## 2 IO流

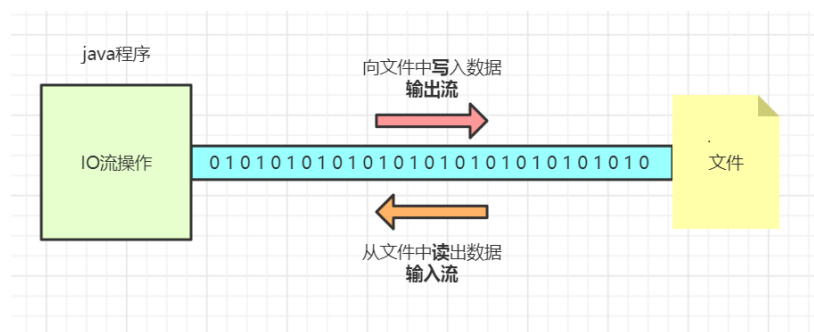
### 2.1 流的概念

在计算机中，流是个抽象的概念，是对输入输出设备的抽象。在Java程序中，对于数据的输入/输出操作，都是以"流"的方式进行

数据以二进制的形式在**程序与设备**之间流动传输，就像水在管道里流动一样，所以就把这种数据传输的方式称之为输入流、输出流。这里描述的设备，可以是文件、网络、内存等

流具有方向性，可以分为输入和输出。

以java程序本身作为参照点，如果数据是从程序“流向”文件，那么这个流就是输出流，如果数据是从文件“流向”程序，那么这个流就是输入流。例如：



注意，这里是以文件进行举例，java程序中还可以把数据写入到网络中、内存中等

## 2.2 流的分类

Java中的IO流可以根据很多不同的角度进行划分，最常见的是以数据的流向和数据的类型来划分

根据**数据的流向**分为：输入流和输出流

- 输入流：把数据从其他设备上读取到程序中的流
- 输出流：把数据从程序中写出到其他设备上的流

根据**数据的类型**分为：字节流和字符流

- 字节流：以字节为单位（byte），读写数据的流
- 字符流：以字符为单位（char），读写数据的流

	输入流	输出流
字节流	字节输入流	字节输出流
字符流	字符输入流	字符输出流

- 字节输入流，在程序中，以字节的方式，将设备（文件、内存、网络等）中的数据读进来
- 字节输出流，在程序中，以字节的方式，将数据写入到设备（文件、内存、网络等）中
- 字符输入流，在程序中，以字符的方式，将设备（文件、内存、网络等）中的数据读进来
- 字符输出流，在程序中，以字符的方式，将数据写入到设备（文件、内存、网络等）中

注意，字节指的是byte，字符指的是char

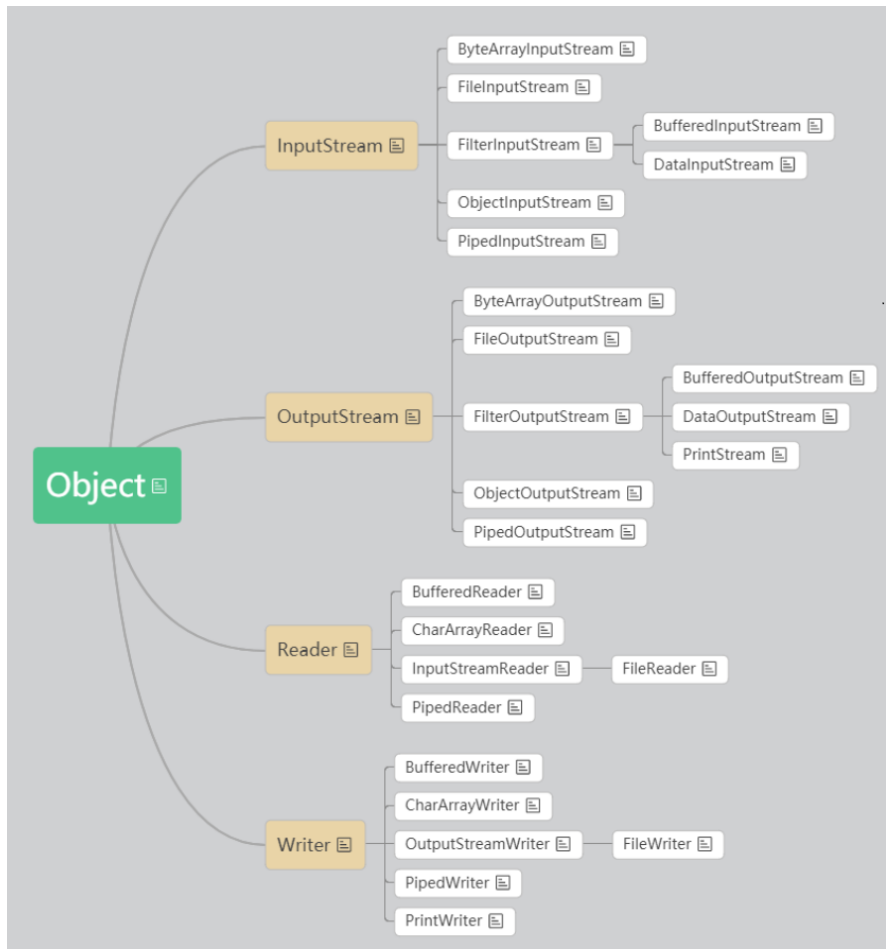
## 2.3 流的结构

在Java中，和IO流相关的类，主要是在 `java.io` 包下定义的

几乎所有的流，都是派生自四个**抽象**的父类型：

- `InputStream`，代表字节输入流类型
- `OutputStream`，代表字节输出流类型
- `Reader`，代表字符输入流类型
- `Writer`，代表字符输出流类型

## Java中常用的流及其继承结构：



注意，一般情况下，一个流，会具备最起码的三个特点：

- 是输入还是输出
- 是字节还是字符
- 流的目的地

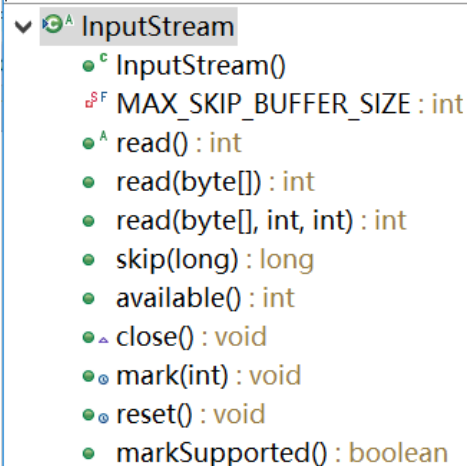
其中，流的目的地指定是：如果是输入流，就表示这个流从什么地方读数据，如果是输出流，就表示这个流把数据写到什么地方！

根据这个规律，再来查看上面的流的关系结构图，就会更加清楚了。

## 2.4 字节流

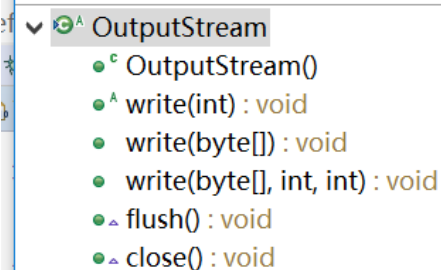
一切文件数据(文本、图片、视频等)在存储时，都是以二进制数字的形式保存，都一个一个字节的，那么传输时一样如此。所以，字节流可以传输任意文件数据。在操作流的时候，我们要时刻明确，无论使用什么样的流对象，底层传输的始终为二进制数据。

`java.io.InputStream` 是所有**字节输入流**的抽象父类型：



```
InputStream
├── InputStream()
├── MAX_SKIP_BUFFER_SIZE : int
├── read() : int
├── read(byte[]) : int
├── read(byte[], int, int) : int
├── skip(long) : long
├── available() : int
├── close() : void
├── mark(int) : void
├── reset() : void
└── markSupported() : boolean
```

`java.io.OutputStream` 是所有**字节输出流**的抽象父类型：



```
OutputStream
├── OutputStream()
├── write(int) : void
├── write(byte[]) : void
├── write(byte[], int, int) : void
├── flush() : void
└── close() : void
```

一般情况，使用字节流来操作数据的时候，往往是使用一对，一个字节输入流，负责读取数据，一个字节输出流，负责将数据写出去，而这些流都将是 `InputStream` 和 `OutputStream` 的子类型

在代码中，使用流操作数据的的基本步骤是：

1. 声明流
2. 创建流
3. 使用流
4. 关闭流

注意，基本所有流的使用，都是这个套路

`InputStream`, `OutputStream` 有很多子类，我们从最简单的一个子类开始。

## 1) 文件输入流

文件字节输入流 `FileInputStream`，用于从文件中读取字节数据。

源码参考：

```
1 package java.io;
2
3 public
4 class FileInputStream extends InputStream
5     //省略...
6
7     //通过File对象来创建一个 FileInputStream
8     public FileInputStream(File file) throws
FileNotFoundException;
9
10    //通过文件路径名(字符串)实例化FileInputStream对象
11    public FileInputStream(String name) throws
FileNotFoundException;
```



```

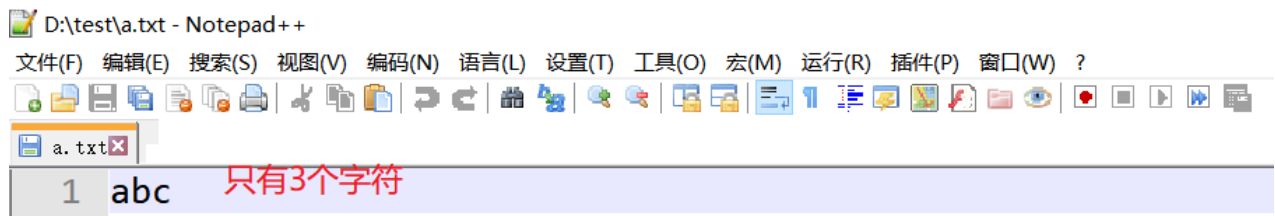
12
13     //逐个字节读取,返回值为读取的单个字节
14     public int read() throws IOException;
15     //小数组读取,将结果存入数组,返回值为读取的字节个数
16     public int read(byte b[]) throws IOException;
17     //小数组读取,存入数组指定位置,返回值为读取的字节个数
18     public int read(byte b[], int off, int len) throws
    IOException;
19
20     //省略...
21 }
22

```

## 案例展示:

使用3种read方法,读取文件 `D:\\test\\a.txt` 内容,掌握read的用法。

`a.txt` 文件内容如下:



按照下图编写测试代码:

```

public class Test024_Read {
    public static void main(String[] args) {
        //1.创建流对象【IO流对象 跟 文件进行关联】
        InputStream is = new FileInputStream("D:\\test\\a.txt");
        System.out.println("is: " + is);

        //2.读取文件内容
        // 1次读1个字节返回,如果到文件末尾,则返回-1
        // int read();
        int r = is.read();
        System.out.println("第1个字节: " + r);
        r = is.read();
        System.out.println("第2个字节: " + r);
        r = is.read();
        System.out.println("第3个字节: " + r);
        r = is.read();
        System.out.println("第4个字节: " + r);    //-1

        //3.关闭流对象,释放资源
        is.close();
    }
}

```

注意: IO操作的每一个方法都可能会抛出异常,必须处理才能通过编译  
 处理方式1: throws声明当前方法抛出  
 处理方式2: try-catch捕获处理

**注意: IO操作每个方法都可能产生异常, 具体如上图。**

我们暂时采用方式1, 借助throws关键字在当前方法中声明抛出异常。

```

1  package com.briup.chap11.test;
2
3  public class Test024_Read {
4      public static void main(String[] args) throws IOException
5      {
6          //1.创建流对象【IO流对象 跟 文件进行关联】
7          InputStream is = new
8          FileInputStream("D:\\test\\a.txt");
9          System.out.println("is: " + is);
10
11         //2.读取文件内容
12         // 1次读1个字节返回,如果到文件末尾,则返回-1
13         // int read();
14         int r = is.read();
15         System.out.println("第1个字节: " + r);
16         r = is.read();
17         System.out.println("第2个字节: " + r);
18     }
19 }

```

```

16         r = is.read();
17         System.out.println("第3个字节: " + r);
18         r = is.read();
19         System.out.println("第4个字节: " + r);    //-1
20
21         //3.关闭流对象,释放资源
22         is.close();
23     }
24 }
25
26 //输出结果:
27 is: java.io.FileInputStream@7852e922
28 第1个字节: 97
29 第2个字节: 98
30 第3个字节: 99
31 第4个字节: -1

```

## 注意事项:

创建FileInputStream对象时, 必须传入一个有效文件路径, 否则抛出  
**FileNotFoundException** !

**int read(byte[] arr);** 方法测试:

```

1  public static void main(String[] args) throws IOException {
2      //1.创建流对象【IO流对象 跟 文件进行关联】
3      InputStream is = new FileInputStream("D:\\test\\a.txt");
4      System.out.println("is: " + is);
5
6      //2.读取文件内容
7      // int read(byte[] arr);
8      // 读多个字节, 放入arr数组, 返回成功读取字节数目,
9      // 如果到文件末尾, 则返回-1
10     byte[] arr = new byte[10];
11     int len = is.read(arr);

```

```

12     System.out.println("成功读取字节数目: " + len);
13
14     //遍历数组有效内容
15     for(int i = 0; i < len; i++)
16         System.out.println(arr[i]);
17
18     System.out.println("-----");
19
20     //再次读取
21     len = is.read(arr);
22     System.out.println("第二次读取: " + len);    // -1
23
24     //3.关闭流对象,释放资源
25     is.close();
26 }
27
28 //输出结果:
29 is: java.io.FileInputStream@7852e922
30 成功读取字节数目: 3
31 97
32 98
33 99
34 -----
35 第二次读取: -1

```

`int read(byte[] arr,int off,int len);` 方法测试:

```

1  public static void main(String[] args) throws Exception {
2      //1.创建流对象
3      InputStream is = new FileInputStream("D:\\test\\a.txt");
4      System.out.println("is: " + is);
5
6      //2.读取
7      // 读取5个字节往arr中 往后偏移3个位置 放入

```

```

8      // 如果读取成功，则返回实际读取长度
9      // 如果返回-1，则表示读取到文件末尾
10     byte[] arr = new byte[10];
11     int len = is.read(arr,3,5); //arr[ , , , a, b, c, ...]
12
13     System.out.println("成功读取: " + len);
14
15     //遍历数组所有内容
16     for(int i = 0; i < arr.length; i++)
17         System.out.print(arr[i] + " ");
18
19     //3.关闭资源
20     is.close();
21 }
22
23 //输出结果:
24 is: java.io.FileInputStream@7852e922
25 成功读取: 3
26 0 0 0 97 98 99 0 0 0 0

```

## 2) 文件输出流

文件字节输出流，`FileOutputStream`，用于写入字节数据到文件中。

`FileOutputStream`源码：

```

1  package java.io;
2
3  public
4  class FileOutputStream extends OutputStream
5      //创建文件输出流以写入由指定的 File对象表示的文件。
6      public FileOutputStream(File file) throws
7      FileNotFoundException;
8
9      //创建文件输出流以指定的名称写入文件

```

```

9      public FileOutputStream(String name) throws
      FileNotFoundException;
10
11     public FileOutputStream(File file, boolean append) throws
      FileNotFoundException;
12
13     public void write(int b) throws IOException;
14     public void write(byte b[]) throws IOException;
15     public void write(byte b[], int off, int len) throws
      IOException;
16
17     //省略...
18 }

```

### 注意事项:

- 创建一个输出流对象时传入的文件路径可以不存在，不会抛出异常，系统会自动创建该文件。
- 如果有这个文件，系统默认会清空这个文件的数据

### 案例描述:

提前创建好目录 `src/dir`，然后使用文件输出流write字节到 `src/dir/a.txt` 中。

```

1  package com.briup.chap11.test;
2
3  public class Test024_Write {
4      public static void main(String[] args) throws Exception {
5          //1.关联流对象和文件
6          // 实例化输出流时，目标文件a.txt不存在不会抛异常，系统会自动
          创建
7          // 但src/dir目录必须存在，系统不会自动创建多级目录

```

```

8         OutputStream os = new
FileOutputStream("src/dir/a.txt");
9         System.out.println("os: " + os);
10
11         //2.写数据
12         os.write(97);    //a
13         os.write(98);    //b
14         os.write(99);    //c
15
16         //3.关闭资源
17         os.close();
18     }
19
20     public static void main02(String[] args) throws Exception
21     {
22         //1.关联流对象和文件
23         OutputStream os = new
FileOutputStream("src/dir/a.txt");
24         System.out.println("os: " + os);
25
26         //2.写数据
27         String str = "abcd";
28         byte[] arr = str.getBytes();
29         //将arr所有元素全部写入文件
30         //写入 会 覆盖 文件原有内容
31         os.write(arr);
32
33         //3.关闭资源
34         os.close();
35     }
36
37     public static void main03(String[] args) throws Exception
38     {
39         //1.关联流对象和文件
40         OutputStream os = new
FileOutputStream("src/dir/a.txt");

```

```

39         System.out.println("os: " + os);
40
41         //2.写数据      '1''2''3''4''5'
42         byte[] arr = {49,50,51,52,53,54,55};
43         //os.write(arr,0,arr.length);
44
45         //从arr[2]开始，获取arr数组的3个字节，即[51,52,53]，然后写
        入a.txt
46         //写入 会 覆盖 文件原有内容
47         os.write(arr,2,3);
48
49         // 写出一个换行，换行符号转成数组写出
50         os.write("\r\n".getBytes());
51
52         //3.关闭资源
53         os.close();
54     }
55 }

```

注意：不同操作系统中回车、换行符是不同的

- 回车符 `\r` 和换行符 `\n`
  - 回车符：回到一行的开头 (return)
  - 换行符：下一行 (newline)
- 系统中的换行：
  - Windows系统里，每行结尾是 回车+换行 ，即 `\r\n`；
  - Unix系统里，每行结尾只有 换行 ，即 `\n`；
  - Mac系统里，每行结尾是 回车 ，即 `\r`。从 Mac OS X开始与Linux统一。



## 综合案例：

拷贝 `src/dir/a.txt` 内容到 `src/dir/b.txt` 中，a.txt文件内容如下。

```
1  hello world
2  1、确认过眼神，我遇上的人。我策马出征，马蹄声如泪奔。青石板上的月光照进这山城。
3  我一路的跟，你轮回声，我对你用情极深。 --方文山 《醉赤壁》
4  2、无关风月 我题序等你回 悬笔一绝 那岸边浪千叠 --方文山 《兰亭序》
5  3、那画面太美，我不敢看 --方文山 《布拉格广场》
6  4、你说 想哭就弹琴 想起你就写信情绪来了就不用太安静 --方文山 《想你就写信》
```

**原理:从已有文件中读取字节,将该字节写出到另一个文件中**



## 代码实现：

```
1  package com.briup.chap11.test;
2
3  public class Test024_Copy {
4      public static void main(String[] args) throws Exception {
5          //1.关联文件和流对象
6          InputStream is = new FileInputStream("src/dir/a.txt");
7          OutputStream os = new
8              FileOutputStream("src/dir/b.txt");
9
10         //2.拷贝
11         //2.1 逐个字节拷贝
12         // int r;
```

```

12 //      while((r = is.read()) != -1) {
13 //          os.write(r);
14 //      }
15
16 //2.2 小数组拷贝，使用最多
17 byte[] arr = new byte[8];
18 int len;
19 while((len = is.read(arr)) != -1) {
20     //注意事项：读取多少个字节 就写出多少个字节
21     os.write(arr,0,len);
22 }
23
24 System.out.println("拷贝完成");
25
26 //3.关闭资源
27 //注意：先关闭后打开的，后关闭先打开的
28 os.close();
29 is.close();
30 }
31 }

```

### 注意事项：

- 逐个字节拷贝效率太低，推荐使用小数组拷贝
- 关闭资源的顺序应该和打开资源顺序相反：先打开的后关闭，后打开的先关闭

### 3) 文件追加

回顾之前案例，我们发现每次创建文件输出流对象，在操作时都会清空目标文件中的数据。

思考：如何保留目标文件中数据，在原有文件内容的后面添加新数据呢？

- `public FileOutputStream(File file, boolean append);`

创建文件输出流以写入由指定的 File对象表示的文件。

- `public FileOutputStream(String name, boolean append);`

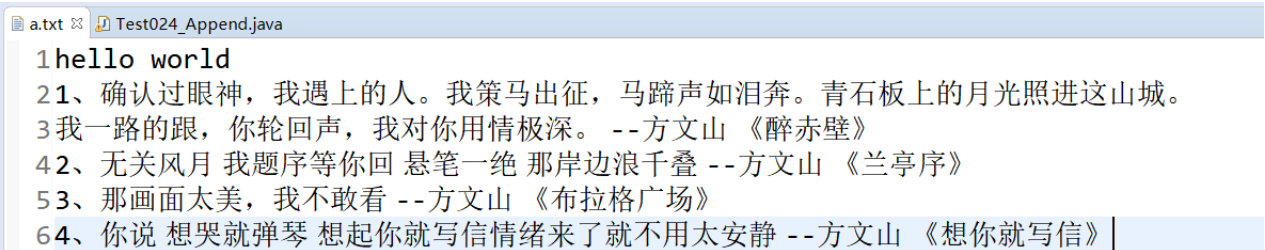
创建文件输出流以指定的名称写入文件。

上面两个构造方法，参数中都需要传入一个boolean类型的值，`true` 表示追加数据，`false` 表示清空原有数据。这样创建的输出流对象，就可以指定是否追加续写了。

### 案例演示：

从键盘录入一行字符串，将其追加到 `src/dir/a.txt` 中。

a.txt文件追加数据前：



```
a.txt Test024_Append.java
1hello world
21、确认过眼神，我遇上的人。我策马出征，马蹄声如泪奔。青石板上的月光照进这山城。
3我一路的跟，你轮回声，我对你用情极深。 -- 方文山 《醉赤壁》
42、无关风月 我题序等你回 悬笔一绝 那岸边浪千叠 -- 方文山 《兰亭序》
53、那画面太美，我不敢看 -- 方文山 《布拉格广场》
64、你说 想哭就弹琴 想起你就写信情绪来了就不用太安静 -- 方文山 《想你就写信》
```

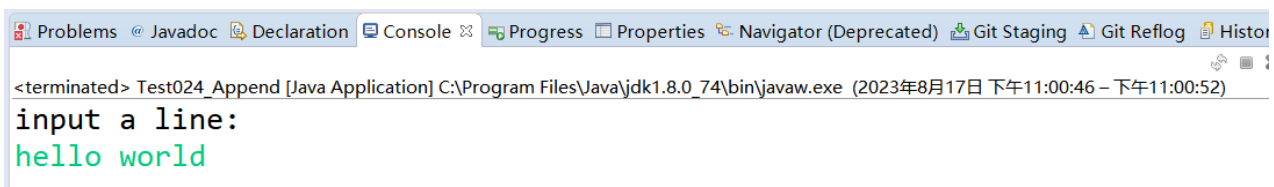
```
1 package com.briup.chap11.test;
2
3 public class Test024_Append {
4     public static void main(String[] args) throws IOException
5     {
6         // 1.实例化输出流对象,设置追加
7         OutputStream os = new
8         FileOutputStream("src/dir/a.txt",true);
9
10        // 2.从键盘录入整行字符串
11        Scanner sc = new Scanner(System.in);
12        System.out.println("input a line: ");
13        String line = sc.nextLine();
```

```

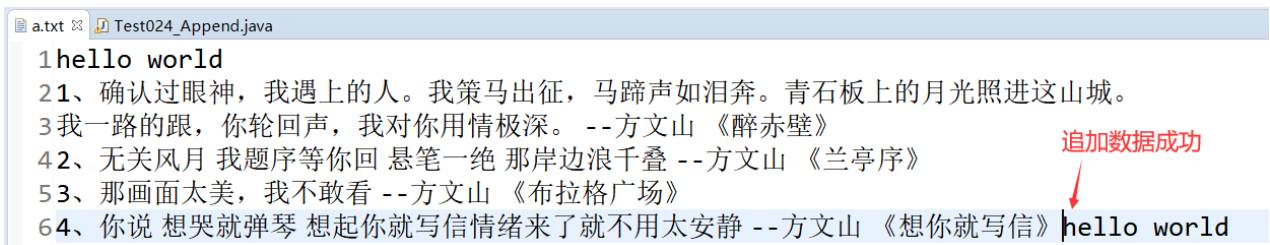
12
13         // 3.将字符串转换为字节数组，并追加到a.txt文件尾
14         byte[] arr = line.getBytes();
15         os.write(arr);
16
17         // 4.关闭资源
18         os.close();
19     }
20 }

```

运行程序：



a.txt内容：



## 4) 内存输出流

使用文件流，我们可以操作文件中的数据。

**使用内存流，我们可以操作内存中字节数组中的数据。**

内存字节流，也称为**字节数组流**，主要有下面两种：

- `java.io.ByteArrayOutputStream`

内存输出流，负责把数据写入到内存中的字节数组中

- `java.io.ByteArrayInputStream`

内存输入流，负责从内存中的字节数组中读取数据

`ByteArrayOutputStream` 源码分析：

```
1  package java.io;
2
3  public class ByteArrayOutputStream extends OutputStream {
4      //存储数据的数组
5      protected byte buf[];
6
7      //存入字节数组的元素(字节)个数
8      protected int count;
9
10     //无参构造器创建的字节数组输出流，数组大小为32个字节
11     public ByteArrayOutputStream() {
12         this(32);
13     }
14
15     //关键方法：获取内存输出流中存储的数据，返回字节数组
16     public synchronized byte toByteArray()[] {
17         return Arrays.copyOf(buf, count);
18     }
19
20     //省略...
21 }
```

### 案例展示：

小数组方式读取 `src/dir/a.txt` 文件中的所有内容，写入到字节数组输出流中，然后从字节输出流中获取所有数据，最后转换成String字符串输出。

```
1  package com.briup.chap11.test;
```

```

2
3 public class Test024_ByteArrayOutput {
4     public static void main(String[] args) throws IOException
5     {
6         //1.关联流对象和文件
7         // 创建内存输出流对象[new byte[32]]
8         InputStream is = new FileInputStream("src/dir/a.txt");
9         ByteArrayOutputStream os = new
10        ByteArrayOutputStream();
11
12        //2.读取文件内容 然后写入到 内存输出流中
13        byte[] arr = new byte[8];
14        int len;
15        while((len = is.read(arr)) != -1) {
16            //写入 内存输出流
17            os.write(arr, 0, len);
18        }
19        System.out.println("拷贝完成!");
20
21        //3.关键方法：获取内存输出流中的数据
22        byte[] byteArray = os.toByteArray();
23
24        //4.将byte[] --> String 并输出
25        System.out.println(new String(byteArray));
26
27        //5.注意：内存流不需要close()释放资源
28    }
29 }

```

运行结果：

```
Problems Javadoc Declaration Console Progress Properties Navigator (Deprecated) Git Staging Git Reflog History Servers
<terminated> Test024_ByteArray [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年8月17日 下午11:22:49 - 下午11:22:50)
拷贝完成!
hello world
1、确认过眼神，我遇上的人。我策马出征，马蹄声如泪奔。青石板上的月光照进这山城。
我一路的跟，你轮回声，我对你用情极深。 --方文山 《醉赤壁》
2、无关风月 我题序等你回 悬笔一绝 那岸边浪千叠 --方文山 《兰亭序》
3、那画面太美，我不敢看 --方文山 《布拉格广场》
4、你说 想哭就弹琴 想起你就写信情绪来了就不用太安静 --方文山 《想你就写信》 hello world
```

## 注意：内存流使用完不需要close()去释放资源！

因为内存流的底层数据源是内存中的字节数组，不需要进行资源释放或关闭操作。当使用完 `ByteArrayXxxStream` 后，可以选择不进行任何操作，它会自动被垃圾回收机制回收。

## 5) 内存输入流

`ByteArrayInputStream` 类源码：

```
1 package java.io;
2
3 public
4 class ByteArrayInputStream extends InputStream {
5     protected byte buf[];
6
7     protected int pos;
8
9     protected int count;
10
11     //关键构造器
12     public ByteArrayInputStream(byte buf[]) {
13         this.buf = buf;
14         this.pos = 0;
15         this.count = buf.length;
16     }
```

```
17
18     //省略...
19 }
```

## 案例展示:

- 1.从键盘录入1行字符串，将其转换为byte[]
- 2.由byte[]构建一个内存输入流对象
- 3.从内存输入流中用小数组方式读取数据，并写入到 `src\dir\b.txt` 文件中
- 4.关闭流、释放资源

```
1  package com.briup.chap11.test;
2
3  public class Test024_ByteArrayInput {
4      public static void main(String[] args) throws Exception {
5          //1.实例化sc对象 并录入一行字符串
6          Scanner sc = new Scanner(System.in);
7          System.out.println("input line: ");
8          String line = sc.nextLine();
9
10         //2.将字符串转换成字节数组
11         byte[] bytes = line.getBytes();
12         //由 字节数组 构建 内存输入流对象
13         InputStream is = new ByteArrayInputStream(bytes);
14         //创建文件输出流对象
15         OutputStream os = new
16             FileOutputStream("src/dir/b.txt");
17
18         //3.小数组方式读取内存流数据
19         byte[] arr = new byte[5];
20         int len;
21         while((len = is.read(arr)) != -1) {
22             //4.写入b.txt文件中
```



```
22         os.write(arr, 0, len);
23     }
24
25     System.out.println("文件操作完成！");
26
27     //5.关闭流、释放资源
28     os.close();
29     // 注意：内存流不需要关闭
30 }
31 }
```

## 内存流总结：

- 如果要操作文件，则需要使用文件流
- 如果要操作内存中的数据，则可以选择内存流

内存字节流提供了一种在内存中进行数据读取和写入的便捷方式。**内存流在实际开发中有专门的应用场景，我们使用不多，了解即可。**

以下是一些常见的应用场景：

1. 数据的临时存储：内存字节流可以将数据暂时存储在内存中的字节数组中，而不需要写入到磁盘或网络中。这在一些临时性的数据处理场景中非常有用，例如在内存中对数据进行加密、解密、压缩、解压缩等操作。
2. 数据的转换：内存字节流可以用于将数据从一种格式转换为另一种格式。例如，可以将一个对象序列化为字节数组，然后再将字节数组反序列化为对象。这在一些需要将数据在内存中进行格式转换的场景中非常有用。
3. 测试和调试：内存字节流可以用于测试和调试目的，例如模拟输入流或输出流的行为。通过将数据写入内存字节流，可以方便地检查数据的内容和格式，而无需依赖外部资源。

4. 单元测试：内存字节流在单元测试中也非常有用。可以使用内存字节流模拟输入和输出流，以便在测试中验证代码的正确性和可靠性，而无需依赖外部文件或网络连接。

需要注意的是，由于内存字节流将数据存储在内存中的字节数组中，因此在处理大量数据时可能会占用较多的内存。在这种情况下，需要谨慎使用内存字节流，以避免内存溢出的问题。

## 2.5 字符流

在 Java 中，字符流是一种用于处理字符数据的输入和输出流。相比于字节流，字符流提供了更方便和高效的字符处理方式。

字符流以字符为单位进行读写操作，可以直接读写字符，而**不需要进行字节与字符的转换**。这使得字符流更适合处理文本数据，可以方便地进行字符的查找、替换、拼接等操作。

### 字节流与字符流关系：

本质上，字符流底层借助字节流实现功能。通过借助字节流，字符流可以在更高的抽象层次上处理字符数据，提供更方便和高效的字符处理方式。**字节流提供了底层的数据传输能力，而字符流在此基础上提供了字符编码、字符集转换、缓冲功能以及其他高级功能。**

`java.io.Reader` 是所有**字符输入流**的抽象父类型：

```
▼ Reader
  ◦ lock : Object
  ◦ Reader()
  ◦ Reader(Object)
  ● read(CharBuffer) : int
  ● read() : int
  ● read(char[]) : int
  ● read(char[], int, int) : int
  ⚠ maxSkipBufferSize : int
  ⚠ skipBuffer : char[]
  ● skip(long) : long
  ● ready() : boolean
  ● markSupported() : boolean
  ● mark(int) : void
  ● reset() : void
  ● close() : void
```

`java.io.Writer` 抽象类是表示用于写出字符流的所有类的超类，将指定的字符信息写出到目的地。它定义了字节输出流的基本共性功能方法。

```
▼ Writer
  ⚠ writeBuffer : char[]
  ⚠ WRITE_BUFFER_SIZE : int
  ◦ lock : Object
  ◦ Writer()
  ◦ Writer(Object)
  ● write(int) : void
  ● write(char[]) : void
  ● write(char[], int, int) : void
  ● write(String) : void
  ● write(String, int, int) : void
  ● append(CharSequence) : Writer
  ● append(CharSequence, int, int) : Writer
  ● append(char) : Writer
  ● flush() : void
  ● close() : void
```

- `void write(int c)` 写入单个字符。
- `void write(char[] cbuf)` 写入字符数组。
- `abstract void write(char[] cbuf, int off, int len)` 写入字符数组的某一部分,off数组的开始索引,len写的字符个数。
- `void write(String str)` 写入字符串。
- `void write(String str, int off, int len)` 写入字符串的某一部分,off字符串的开始索引,len写的字符个数。

- `void flush()` 刷新该流的缓冲。
- `void close()` 关闭此流，但要先刷新它。

这些字符流中的读和写的方法，与字节流中的读和写方法类似，掌握了字节流的使用，这些很容易理解

## 1) 文件字符流

`Reader`, `Writer` 有很多子类，我们从最简单的一个子类开始。

`java.io.FileReader` 类是读取字符文件的便利类。构造时使用系统默认的字符编码和默认字节缓冲区。

`java.io.FileWriter` 类是写出字符到文件的便利类。构造时使用系统默认的字符编码和默认字节缓冲区。

```
1  package java.io;
2
3  public class FileReader extends InputStreamReader {
4      public FileReader(String fileName) throws
        FileNotFoundException;
5
6      public FileReader(File file) throws FileNotFoundException;
7
8      //省略...
9  }
10
11 public class FileWriter extends OutputStreamWriter {
12     public FileWriter(String fileName) throws IOException;
13
14     public FileWriter(String fileName, boolean append) throws
        IOException;
15
16     public FileWriter(File file) throws IOException;
```

```
17
18     public FileWriter(File file, boolean append) throws
        IOException;
19
20     //省略...
21 }
```

### 注意事项:

创建一个输入流对象时, 必须传入一个有效文件路径, 否则会抛出 `FileNotFoundException`

创建一个输出流对象时, 传入的文件路径可以不存在, 系统会自动创建该文件。如果有这个文件, 默认会清空这个文件的数据

### 案例描述:

- 1 使用文件字符流拷贝a.txt文件内容到b.txt文件末尾

### 案例实现:

```
1  package com.briup.chap11.test;
2
3  public class Test025_FileReaderWriter {
4      public static void main(String[] args) throws IOException
5      {
6          // 1.实例化流对象
7          File file1 = new File("src/dir/a.txt");
8          File file2 = new File("src/dir/b.txt");
9          Reader reader = new FileReader(file1);
10         // 设置文件追加
11         Writer writer = new FileWriter(file2,true);
12
13         // 2.使用流进行文件拷贝
14         int len = -1;
15         char[] buf = new char[8];
```

```

15
16         while ((len = reader.read(buf)) != -1) {
17             writer.write(buf, 0, len);
18         }
19         //刷新流
20         writer.flush();
21
22         // 3.关闭流
23         writer.close();
24         reader.close();
25     }
26 }

```

运行效果自行测试。

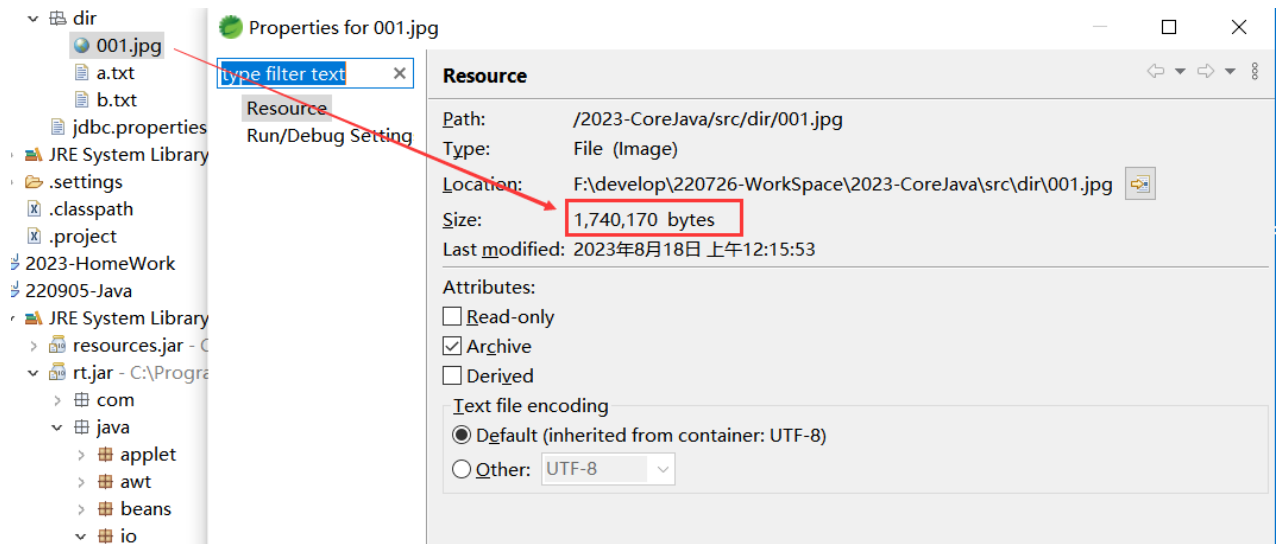
## 2) 操作字节文件

### 案例描述：

- 1 使用文件字符流完成图片的拷贝，看结果发现问题

**原理:从已有文件中读取字节,将该字节写出到另一个文件中**





## 案例实现：

修改上面 `Test025_FileReaderWriter` 案例：

- 1. 将操作的文件换成 `src/dir/001.jpg` 和 `src/dir/002.jpg`
- 2. 去除输出流追加标志

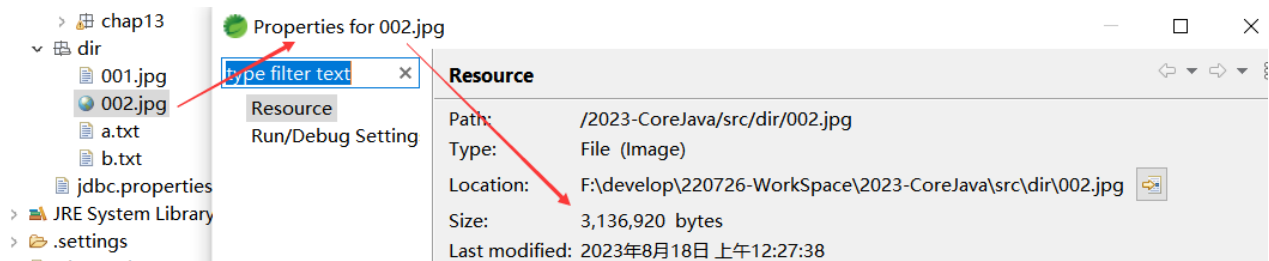
```
10 public class Test025_FileReaderWriter {
11     public static void main(String[] args) throws IOException {
12         // 1. 实例化流对象
13         //File file1 = new File("src/dir/a.txt");
14         File file1 = new File("src/dir/001.jpg");
15         //File file2 = new File("src/dir/b.txt");
16         File file2 = new File("src/dir/002.jpg");
17         Reader reader = new FileReader(file1);
18         // 设置文件追加
19         Writer writer = new FileWriter(file2);
```

修改代码

运行代码，得到002.jpg后打开，发现问题：文件打不开！



查看文件属性，发现其与001.jpg大小不同：



**结论：字符流，只能操作文本文件，不能操作图片，视频等非文本文件。**

当我们单纯读或者写纯文本文件时，使用字符流

其他情况（图片、音频、doc、xls、ppt等），使用字节流

### 3) 异常处理

之前我们对异常的处理，都是直接throws抛出，实际开发中并不能这样处理，建议使用 `try...catch...finally` 代码块处理异常部分代码。

修改 `Test025_FileReaderWriter` 案例，自行捕获异常处理：

```
1 package com.briup.chap11.test;
2
3 public class Test026_Catch {
4     public static void main(String[] args) {
5         Reader reader = null;
6         Writer writer = null;
7
8         try {
9             // 1.实例化流对象
10            reader = new FileReader("src/dir/a.txt");
11            // 设置文件追加
12            writer = new FileWriter("src/dir/b.txt",true);
13
14            // 2.使用流进行文件拷贝
```



```

15         int len = -1;
16         char[] buf = new char[8];
17
18         while ((len = reader.read(buf)) != -1) {
19             writer.write(buf, 0, len);
20         }
21         //刷新流
22         writer.flush();
23     }catch(IOException e) {
24         e.printStackTrace();
25     }finally {
26         // 3.关闭流，先创建的后关闭，后创建的先关闭
27         try {
28             if(writer != null)
29                 writer.close();
30         } catch (IOException e) {
31             e.printStackTrace();
32         }
33
34         //每个流对象，都要单独进行try-catch异常捕获
35         //这使得writer.close如果出现异常，不影响reader.close
36         try {
37             if(reader != null)
38                 reader.close();
39         } catch (IOException e) {
40             e.printStackTrace();
41         }
42     }
43 }
44 }

```

上面就是标准的异常处理写法，后续IO代码中如果要自行捕获异常，按上述格式书写即可。

## 扩展知识：JDK7新增异常处理方式（了解即可）

JDK7优化后的 `try-with-resource` 语句，该语句确保了每个资源在语句结束时关闭。

所谓的资源（resource）是指在程序完成后，必须关闭的对象。

格式：

```
1  try (创建流对象语句, 如果存在多个, 使用 ';' 隔开) {  
2      // 读写数据  
3  } catch (IOException e) {  
4      e.printStackTrace();  
5  }
```

代码使用演示：

```
1  package com.briup.chap11.test;  
2  
3  public class Test027_Catch {  
4      public static void main(String[] args) {  
5          // 1.实例化流对象,这种格式，系统会自动释放资源  
6          try(Reader reader = new FileReader("src/dir/a.txt");  
7              Writer writer = new  
8              FileWriter("src/dir/b.txt",true);) {  
9  
10             // 2.使用流进行文件拷贝  
11             int len = -1;  
12             char[] buf = new char[8];  
13  
14             while ((len = reader.read(buf)) != -1) {  
15                 writer.write(buf, 0, len);  
16             }  
17             //刷新流  
18             writer.flush();  
19  
20             System.out.println("拷贝完成...");  
21         }  
22     }  
23 }
```

```
20
21         }catch(IOException e) {
22             e.printStackTrace();
23         }
24     }
25 }
```

运行结果自行测试！

## 4) 节点流总结

在 Java 中，IO 流按照功能划分可以分为两类：

- 节点流（原始流）
- 增强流（包装流）

**节点流（Node Streams）是最基本的 IO 流，直接与数据源或目标进行交互，但缺乏一些高级功能。**

它们提供了最底层的读写功能，可以直接读取或写入底层数据源或目标，如文件、网络连接等。

上面我们学习的所有流都是节点流：

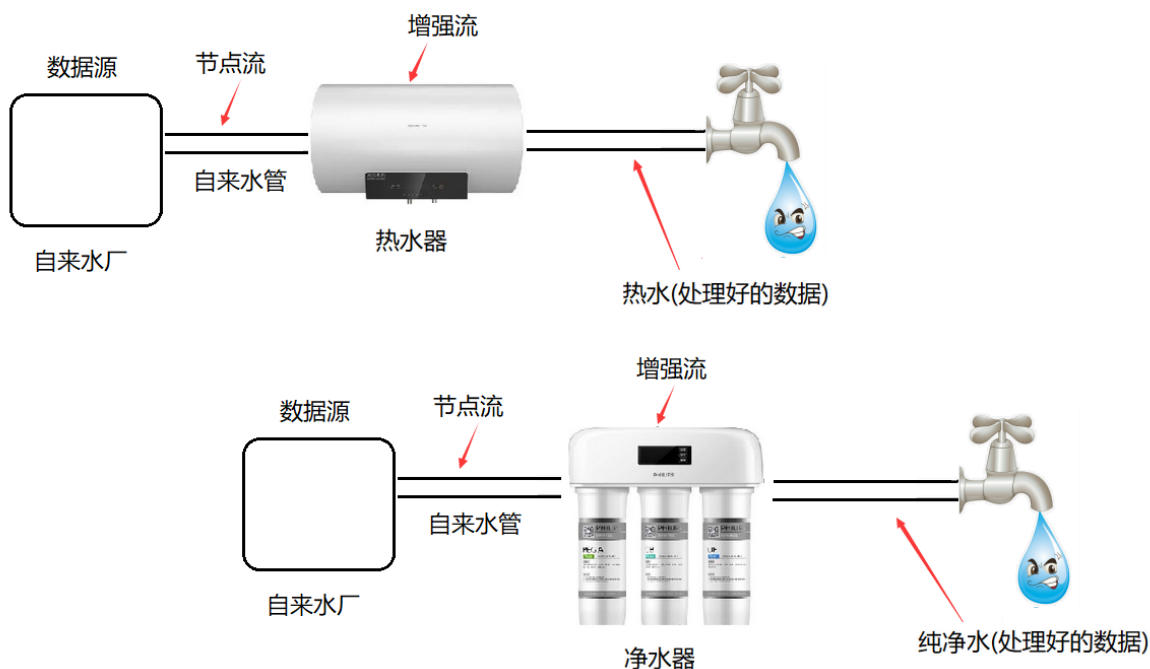
- `FileInputStream` 和 `FileOutputStream`：用于读取和写入文件的字节流。
- `FileReader` 和 `FileWriter`：用于读取和写入文件的字符流
- `ByteArrayInputStream` 和 `ByteArrayOutputStream`：用于读取和写入字节数组的流
- `CharArrayReader` 和 `CharArrayWriter`：用于读取和写入字符数组的流（省略，与 `ByteArray` 字节流类似，可自学）

节点流只能提供最基本的读写功能，实际开发中我们很少单独使用它们，也解决复杂IO功能。

### 增强流也称为包装流：

- 其在节点流的基础上提供了额外的功能和操作
- 增强流提供了更高级的操作和便利性，使得 IO 操作更加方便、高效和灵活
- 增强流通过装饰器模式包装节点流，可以在节点流上添加缓冲、字符编码转换、对象序列化等功能

### 节点流、增强流理解：



在实际开发中，通常会使用增强流来提供更高级的功能和操作，以便更方便地进行 IO 操作。

**节点流则作为增强流的基础，提供最底层的读写功能。**

## 2.6 缓冲流

**注意：从本章开始，后面讲解的IO流，大多数都为增强流。**

### 1) 缓冲思想

在 Java 的 I/O 流中，缓冲思想是一种常见的优化技术，**用于提高读取和写入数据的效率**。它通过在内存中引入缓冲区（Buffer）来减少实际的 I/O 操作次数，从而提高数据传输的效率。

缓冲思想的基本原理是将数据暂时存储在内存中的缓冲区中，然后按照一定的块大小进行读取或写入操作。相比于直接对磁盘或网络进行读写操作，使用缓冲区可以减少频繁的 I/O 操作，从而提高效率。

### 缓冲流概述：

缓冲流（Buffered Streams）也叫**高效流**，是一种非常有用的**增强流**，提供了缓冲功能，可以提高 IO 操作的效率。

缓冲流它们通过在内存中创建一个缓冲区，将数据暂时存储在缓冲区中，然后批量读取或写入数据，减少了频繁的磁盘或网络访问，从而提高了读写的性能。

### 缓冲流理解：

生活案例：你需要从楼下小超市买30颗鸡蛋

- **文件节点流逐个字节传输**

跟老板约定好，通过一条专门的运输通道(电梯)来运输鸡蛋，老板每次往电梯里放1个鸡蛋，你在9楼电梯口等着拿鸡蛋)，运输30个鸡蛋，需要电梯上下30次。

- **缓冲增强流传输数据**

仍旧通过原来的运输通道(电梯)来运输鸡蛋，但对电梯做增强：里面放了个小篮子(能装20颗鸡蛋)。第一次老板往篮子里装鸡蛋，装20个篮子满了，让电梯上楼，你取走全部鸡蛋；第二次老板往篮子里装剩下的10颗鸡蛋，让电梯上楼，你取走，所有鸡蛋全部传输完成。相对之前，这种方式电梯只需要上下2次，传输效率大大提高。

### 常见缓冲流：

- `BufferedInputStream` 缓冲字节输入流
- `BufferedOutputStream` 缓冲字节输出流
- `BufferedReader` 缓冲字符输入流
- `BufferedWriter` 缓冲字符输出流

## 2) 缓存字节流

- `public BufferedInputStream(InputStream in)`：创建一个 新的缓冲输入流。
- `public BufferedOutputStream(OutputStream out)`：创建一个新的缓冲输出流。

### 源码分析：

```
1  package java.io;
2
3  public
4  class BufferedInputStream extends FilterInputStream {
5      //省略...
6
7      //默认缓冲区大小8192字节
8      private static int DEFAULT_BUFFER_SIZE = 8192;
9      //字节数组
```

```

10     protected volatile byte buf[];
11     //缓冲区最大字节数
12     private static int MAX_BUFFER_SIZE = Integer.MAX_VALUE -
13         8;
14
15     //缓冲输入流，又称为包装流，使用时需要传入基本输入字节流对象
16     public BufferedInputStream(InputStream in);
17     //size，代表设置读取缓冲大小
18     public BufferedInputStream(InputStream in, int size);
19 }
20
21 public
22 class BufferedOutputStream extends FilterOutputStream {
23     //省略...
24
25     protected byte buf[];
26
27     //使用时需要传入基本输出字节流对象，默认缓冲区大小8192字节
28     public BufferedOutputStream(OutputStream out) {
29         this(out, 8192);
30     }
31
32     //size，代表设置写入缓冲大小
33     public BufferedOutputStream(OutputStream out, int size);
34 }

```

**注意：缓冲流是增强流，其底层借助其他流实现功能，所以构造器要求一定要传入一个字节流对象！**

思考：如何验证使用缓冲流来增强文件字节流的功能，提高读写效率？

**案例展示：**

准备图片 `D:\\test\\1.png`，拷贝到 `D:\\test\\2.png`，分别使用文件子节点（节点流）和缓冲流拷贝图片，对比两种方式拷贝所需时间。

```
1  package com.briup.chap11.test;
2
3  public class Test026_Buffered {
4      public static void main(String[] args) throws Exception {
5          //1.创建流对象
6          FileInputStream fis = new
FileInputStream("D:\\test\\1.png");
7          FileOutputStream fos = new
FileOutputStream("D:\\test\\2.png");
8
9          //2.借助节点流流 进行 逐行读取
10         long start = System.currentTimeMillis();
11
12         int data;
13         while((data = fis.read()) != -1) {
14             fos.write(data);
15         }
16
17         long end = System.currentTimeMillis();
18         System.out.println("拷贝完成,拷贝时长: " + (end - start)
+ "ms");
19
20         //3.只需要关闭 最后的增强流对象即可
21         fos.close();
22         fis.close();
23     }
24 }
25
26 //输出结果: 文件大小5M左右, 注意每次运行结果稍微不同
27 拷贝完成,拷贝时长: 36410ms
```

使用缓冲流拷贝:



```

1  public static void main(String[] args) throws Exception {
2      //1.创建缓冲流
3      FileInputStream fis = new
FileInputStream("D:\\test\\1.png");
4      // 注意：缓冲流是增强流，其底层借助节点流实现功能，所以创建时须传
入一个节点流对象
5      BufferedInputStream bis = new BufferedInputStream(fis);
6      BufferedOutputStream bos =
7          new BufferedOutputStream(new
FileOutputStream("D:\\test\\2.png"));
8
9      //2.借助缓冲流 进行 逐行读取
10     long start = System.currentTimeMillis();
11
12     byte[] buff = new byte[1024];
13     int len;
14     while((len = bis.read(buff)) != -1) {
15         bos.write(buff, 0, len);
16
17         //刷新缓冲流
18         bos.flush();
19     }
20
21     long end = System.currentTimeMillis();
22     System.out.println("拷贝完成,拷贝时长: " + (end - start) +
"ms");
23
24     //3.只需要关闭 最后的增强流对象即可
25     bos.close();
26     bis.close();
27 }
28
29 //输出结果：文件大小5M左右，注意每次运行结果稍微不同
30 拷贝完成,拷贝时长：35ms

```

从对比两个程序运行结果可知，缓冲流能明显提高IO效率。

### 3) 缓冲字符流

- `public BufferedReader(Reader in)` : 创建一个新的缓冲输入流。
- `public BufferedWriter(Writer out)` : 创建一个新的缓冲输出流。

源码分析:

```
1  package java.io;
2
3  public class BufferedReader extends Reader {
4      //省略...
5
6      //基础流
7      private Reader in;
8      //字符数组
9      private char cb[];
10     //默认缓冲区大小
11     private static int defaultCharBufferSize = 8192;
12
13     //缓冲输入流，又称为包装流，使用时需要传入基本输入字符流对象
14     public BufferedReader(Reader in);
15     //size, 代表设置读取缓冲大小
16     public BufferedReader(Reader in, int size);
17
18     //新增功能：逐行读取数据
19     //如果没有下一行数据了，返回`null`
20     public String readLine() throws IOException;
21
22 }
23
24 public class BufferedWriter extends Writer {
25     //省略...
26
27     //基础流
```

```

28     private Writer out;
29     //字符数组
30     private char cb[];
31     //默认缓冲区大小
32     private static int defaultCharBufferSize = 8192;
33
34     //缓冲输出流，又称为包装流，使用时需要传入基本输出字符流对象
35     public BufferedWriter(Writer out);
36     //size，代表设置写入缓冲大小
37     public BufferedWriter(Writer out, int size);
38
39     //新增功能：写入一个换行符，windows、linux等都适用
40     public void newLine() throws IOException;
41
42 }

```

可以看出，字符缓冲流的构造器，要求一定要传入一个字符流对象，然后缓冲流就可以对这个字符流的功能进行增强，提供缓冲数据的功能，从而提高读写的效率

## 案例如下：

使用缓冲字符流完成字符文件的拷贝，注意使用缓冲流的新方法。

```

1  package com.briup.chap11.test;
2
3  public class Test026_BufferedChar {
4      public static void main(String[] args) throws Exception {
5          //1.
6          BufferedReader br =
7              new BufferedReader(new
8                  FileReader("src/dir/a.txt"));
9      }
10 }

```

```

8         BufferedWriter bw =
9             new BufferedWriter(new
FileWriter("src/dir/b.txt"));
10
11         //2.逐行读取 输出 最后拷贝
12         String line;
13         //读取整行数据 不包含 换行符
14         while((line = br.readLine()) != null) {
15             //输出读取整行数据
16             System.out.println("read: " + line);
17             //将读取的整行数据 写入b.txt
18             bw.write(line);
19
20             //额外写换行符：如果后续没有数据了，则不要再写换行符
21             // ready()返回false可理解为：马上要读取到文件尾
22             if(br.ready())
23                 bw.newLine();
24         }
25
26         //3.
27         bw.close();
28         br.close();
29     }
30 }

```

运行效果省略！

注意：第22行输入流的ready()方法，如果流还有数据，ready返回true，如果读取到文件末尾，ready返回false。

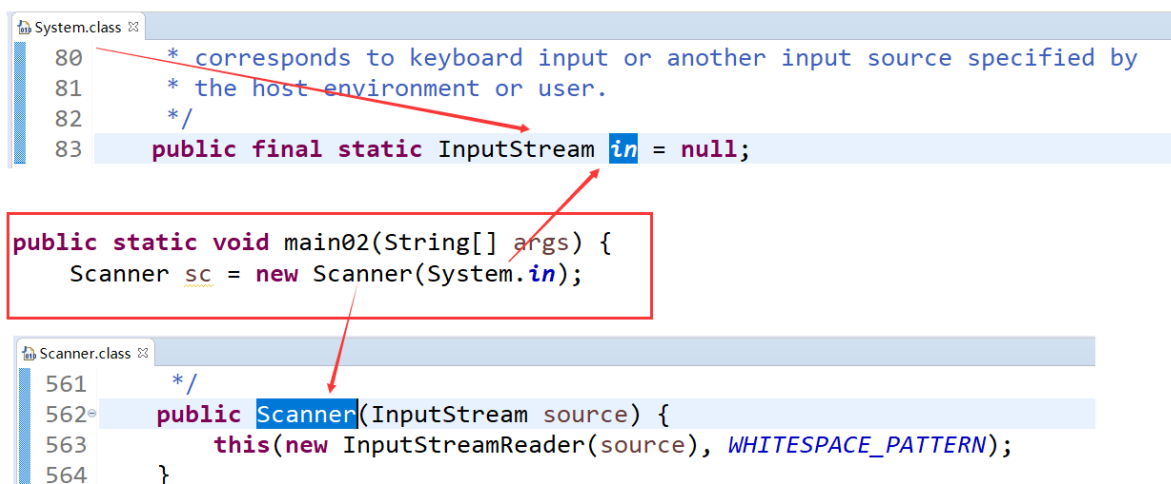
## 2.7 标准流

在 Java 中，标准流（Standard Streams）是指三个预定义的流对象，用于**处理标准输入、标准输出和标准错误**。这些标准流在 Java 中是自动创建的，无需显式地打开或关闭。

以下是 Java 中的标准流：

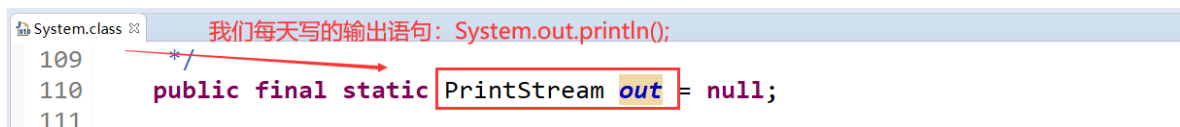
### 1. 标准输入流（System.in）

它是一个字节流（InputStream），用于从标准输入设备（通常是**键盘**）读取数据。可以使用 `Scanner` 或 `BufferedReader` 等类来读取标准输入流的数据。



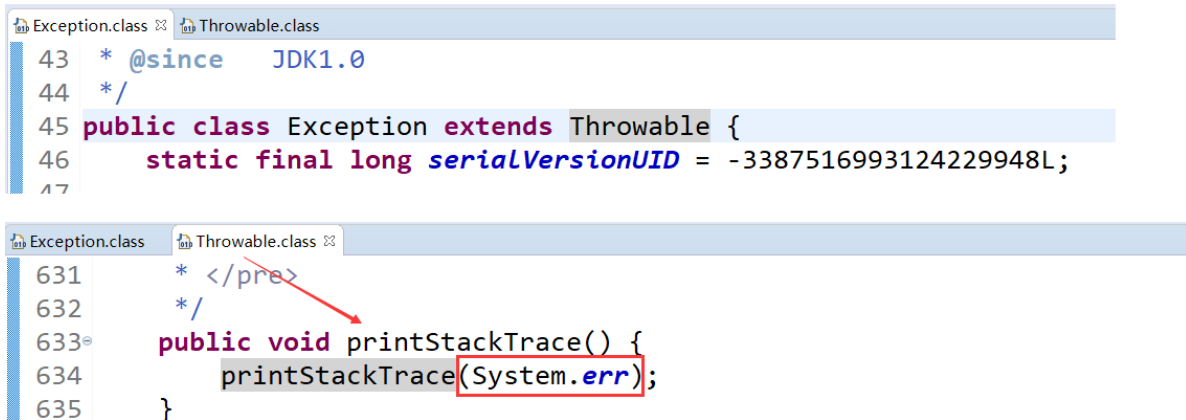
### 2. 标准输出流（System.out）

它是一个字节流（PrintStream），用于向标准输出设备（通常是**控制台**）输出数据。可以使用 `System.out.println()` 或 `System.out.print()` 等方法来输出数据到标准输出流。



### 3. 标准错误流（System.err）

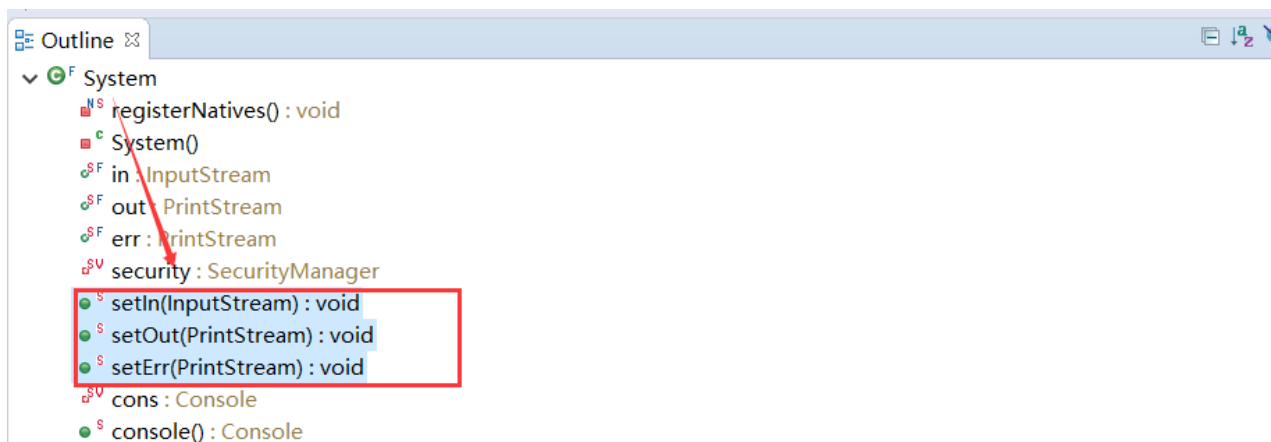
也是一个字节流（PrintStream），用于向标准错误设备（通常是控制台）输出错误信息。与标准输出流相比，标准错误流通常用于输出错误、警告或异常信息。



The image shows two snippets of Java source code. The top snippet is from `Exception.class` and shows lines 43 to 47, including the `@since JDK1.0` annotation and the `serialVersionUID` field. The bottom snippet is from `Throwable.class` and shows lines 631 to 635, including the `printStackTrace()` method which calls `printStackTrace(System.err)`. A red arrow points from the `printStackTrace()` call in the bottom snippet to the `System.err` field in the `System` class outline shown in the next image.

```
43 * @since   JDK1.0
44 */
45 public class Exception extends Throwable {
46     static final long serialVersionUID = -3387516993124229948L;
47
631 * </pre>
632 */
633 public void printStackTrace() {
634     printStackTrace(System.err);
635 }
```

我们可以借助 `System` 类中的 `setXxx` 方法 修改标准输入输出流的源头和目的地。



The image shows the 'Outline' view of the Java IDE, displaying the `System` class. A red box highlights the `setIn(InputStream) : void`, `setOut(PrintStream) : void`, and `setErr(PrintStream) : void` methods. A red arrow points from the `printStackTrace()` call in the previous code snippet to the `System.err` field in this outline.

```
System
├── registerNatives() : void
├── System()
├── in : InputStream
├── out : PrintStream
├── err : PrintStream
├── security : SecurityManager
├── setIn(InputStream) : void
├── setOut(PrintStream) : void
├── setErr(PrintStream) : void
├── cons : Console
└── console() : Console
```

## PrintStream打印流:

`PrintStream`是一个字节流，也是一个增强流，相对于普通节点流，它提供了一系列便捷的方法，可以方便地输出各种数据类型的值到输出流。

它提供了一系列的 `print` 和 `println` 方法，可以输出各种数据类型的值，并自动转换为字符串形式。通常用于向标准输出流（`System.out`）输出数据。

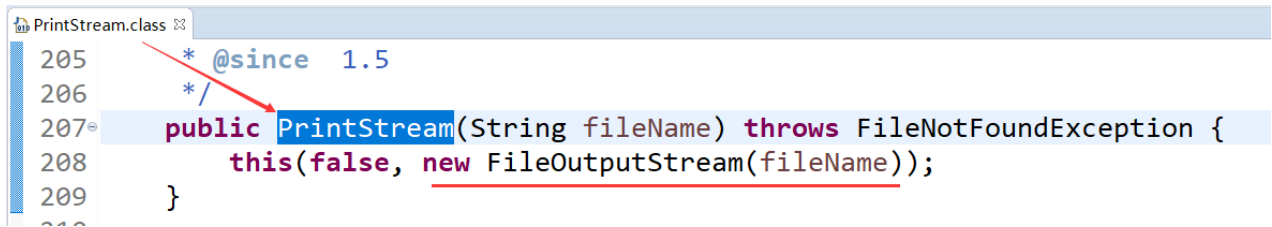
## 案例展示:

文件拷贝的最终版本代码。

```
1  package com.briup.chap11.test;
2
3  public class Test027_PrintStream {
4      public static void main(String[] args) throws Exception {
5          //1.实例化流对象
6          BufferedReader br =
7              new BufferedReader(new
8              FileReader("src/dir/a.txt"));
9
10         //2.拷贝
11         //逐行拷贝 逐行写出
12         String line;
13         while((line = br.readLine()) != null) {
14             if(br.ready())
15                 ps.println(line);
16             else
17                 ps.print(line);
18         }
19         System.out.println("拷贝完成...");
20
21         //3.关闭资源
22         ps.close();
23         br.close();
24     }
25 }
```

注意：有同学会有疑问，PrintStream是增强流，为什么它实例化对象代码为  
`PrintStream ps = new PrintStream("src/dir/b.txt");`

查看源码即可解决困惑：



```

205  * @since 1.5
206  */
207  public PrintStream(String fileName) throws FileNotFoundException {
208      this(false, new FileOutputStream(fileName));
209  }

```

## 标准流案例：

先按照默认的标准输出流输出内容，然后改变输出方向，进行测试

```

1  package com.briup.chap11.test;
2
3  public class Test027_StandardIO {
4      public static void main(String[] args) throws IOException
5      {
6          System.out.println("main start ...");
7
8          //实例化打印流对象
9          PrintStream ps = new PrintStream("src/dir/c.txt");
10
11          System.out.println("before 修改标准输出: ");
12          System.out.println(ps);
13
14          System.out.println("即将修改 标准输出!");
15          //修改 标准输出流 关联文件为 c.txt文件
16          System.setOut(ps);
17
18          //再次标准输出
19          System.out.println("after setOut, ps: " + ps);
20      }
21  }
22  //运行效果:
23  main start ...
24  before 修改标准输出:
25  java.io.PrintStream@7852e922

```



```
c.txt ✕  
1 after setOut, ps: java.io.PrintStream@7852e922  
2
```

**注意：**对于这3个标准流，我们知道它们的应用场景，清楚它们的用法即可，不用花费过多精力。

## 2.8 转换流

### 1) 字符编码和字符集

计算机中储存的信息都是用二进制数表示的，而我们在屏幕上看到的数字、英文、标点符号、汉字等字符是二进制数转换之后的结果。

按照某种规则，将字符存储到计算机中，称为**编码**。然后将存储在计算机中的二进制数按照某种规则解析显示出来，称为**解码**。

比如说，按照A规则存储，同样按照A规则解析，那么就能显示正确的文本符号。但如果按照A规则存储，按照B规则解析，就会导致乱码现象。

- **字符编码 Character Encoding**：一套自然语言字符与二进制数（码点）之间的对应规则。
- **字符集 Charset**：也叫编码表。是一个系统支持的所有字符的集合，包括各国家文字、标点符号、图形符号、数字等。

计算机要准确的存储和识别各种字符集符号，需要进行字符编码，一套字符集必然至少有一套字符编码。常见字符集有ASCII字符集、GBK字符集、Unicode字符集等。



可见，当指定了**编码**，它所对应的**字符集**自然就指定了，所以**编码**才是我们最终要关心的。

## 2) 编码引出的问题

使用 `FileReader` 读取项目中的文本文件。由于软件的设置，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。

### 案例展示：

准备文件 `D:\\test\\File_GBK.txt`，编码为gbk编码，然后用FileReader读取文件内容输出。

注意项目默认编码为utf-8。

```
1 package com.briup.chap11.test;
2
3 public class Test028_GbkToUtf8 {
4     public static void main(String[] args) throws IOException
5     {
6         FileReader fileReader = new
7         FileReader("D:\\test\\File_GBK.txt");
8         int read;
9         while ((read = fileReader.read()) != -1) {
10             System.out.print((char)read);
11         }
12         fileReader.close();
13     }
14 }
```

```
11     }
12 }
13
14 输出结果:
15  ???
```

那么如何读取GBK编码的文件呢？

### 3) 字节字符转换

`java.io.OutputStreamWriter`，可以将字节输出流转换为字符输出流，并指定编码

`java.io.InputStreamReader`，可以将字节输入流转换为字符输入流，并指定编码

```
1  package java.io;
2
3  public class InputStreamReader extends Reader {
4      //使用默认编码转换
5      public InputStreamReader(InputStream in);
6      //使用指定编码转换
7      public InputStreamReader(InputStream in, String
      charsetName);
8
9      //省略...
10 }
11
12 public class OutputStreamWriter extends Writer {
13     //使用默认编码转换
14     public OutputStreamWriter(OutputStream in);
15
16     //使用指定编码转换
17     public OutputStreamWriter(OutputStream in, String
      charsetName);
```

```
18
19     //省略...
20 }
```

可以看出，它的构造器参数，要求传入一个需要转换的字节输出流，和一个指定的字符编码

### 指定编码读取案例：

使用转换流、缓冲流解决上述案例中的问题。

```
1  package com.briup.chap11.test;
2
3  public class Test028_InputStreamReader {
4      public static void main(String[] args) throws IOException
5      {
6          // 1.创建转换流对象，指定使用GBK编码
7          File file = new File("D:\\test\\File_GBK.txt");
8          InputStreamReader isr =
9              new InputStreamReader(new
10             FileInputStream(file), "GBK");
11
12          // 2.读取文件内容并输出
13          char[] buff = new char[8];
14          int len;
15          while ((len = isr.read(buff)) != -1) {
16              System.out.print(new String(buff,0,len));
17          }
18
19          //3.关闭流 释放资源
20          isr.close();
21      }
22  }
```

运行效果省略。

## 指定编码写入案例：

按照GBK编码读取 `D:\\test\\File_GBK.txt` 文件内容，然后写入UTF-8编码文件 `D:\\test\\File_UTF8.txt`。注意拷贝效率，注意新文件中不要出现多余的空行。

```
1  package com.briup.chap11.test;
2
3  public class Test028_Copy {
4      //优化转换流 读取，实现整行读取
5      // 可以对流 进行 反复增强
6      //  FileInputStream --> InputStreamReader -->
        BufferedReader
7      public static void main(String[] args) throws Exception {
8          String file1 = "D:\\test\\File_GBK.txt";
9          String file2 = "D:\\test\\File_UTF8.txt";
10
11          InputStream is = new FileInputStream(file1);
12          InputStreamReader isr = new
        InputStreamReader(is, "gbk");
13          //准备缓冲字符流 进一步增强 性能
14          BufferedReader br = new BufferedReader(isr);
15
16          //简化写法，获取缓冲字符流对象(指定使用utf-8编码)
17          BufferedWriter bw =
18              new BufferedWriter(new OutputStreamWriter(new
        FileOutputStream(file2), "utf-8"));
19
20          //2.借助缓冲流 进行 逐行读取
21          String line;
22          while((line = br.readLine()) != null) {
23              bw.write(line);
24          }
```

```
25          //注意：如果读取的行不是最后一行，则额外写入换行
26          if(br.ready())
27              bw.newLine();
28      }
29
30      System.out.println("输出完成！");
31
32      //3.只需要关闭 最后的增强流对象即可
33      bw.close();
34      br.close();
35  }
36 }
```

运行效果省略。

## 2.9 对象流

### 1) 序列化机制

Java 提供了一种对象序列化的机制，可以将对象和字节序列之间进行转换

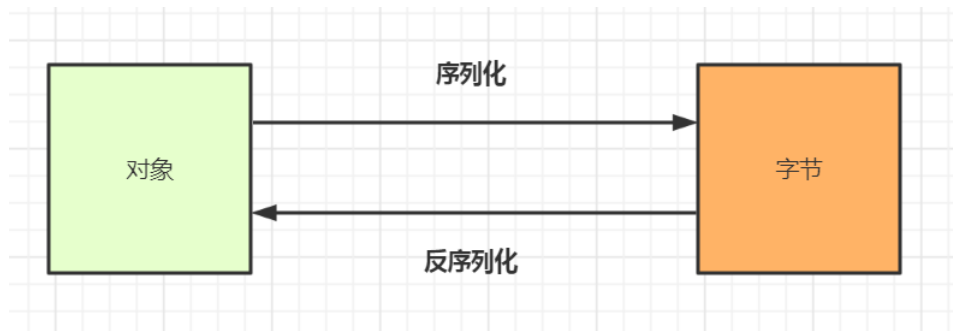
- **序列化**

程序中，可以用一个字节序列来表示一个对象，该字节序列包含了对象的类型、对象中的数据等。如果这个字节序列写出到文件中，就相当于在文件中持久保存了这个对象的信息

- **反序列化**

相反的过程，从文件中将这个字节序列读取回来，在内存中重新生成这个对象，对象的类型、对象中的数据等，都和之前的那个对象保持一致。（注意，这时候的对象和之前的对象，内存地址可能是不同的）

如图：



完成对象的序列化和反序列化，就需要用到对象流了

## 2) 对象流介绍

- `java.io.ObjectOutputStream`

将Java对象转换为字节序列，并输出到内存、文件、网络等地方

- `java.io.ObjectInputStream`

从某一个地方读取对象的字节序列，并生成对应的对象

```
1  package java.io;
2
3  public class ObjectOutputStream
4      extends OutputStream implements ObjectOutput,
      ObjectOutputStreamConstants
5  {
6      //省略...
7
8      public ObjectOutputStream(OutputStream out) throws
      IOException;
9
10     //序列化方法
11     public final void writeObject(Object obj) throws
      IOException;
12 }
```

```

13
14 public class ObjectInputStream
15     extends InputStream implements ObjectInput,
    ObjectOutputStreamConstants
16 {
17     //省略...
18
19     public ObjectInputStream(InputStream in) throws
        IOException;
20
21     //反序列化方法
22     public final Object readObject() throws IOException,
        ClassNotFoundException;
23 }

```

## 写入对象案例：

准备Student类，使用对象流将学生对象保存在文件中，并读取出来。



## 基础类：

```

1 package com.briup.chap11.bean;
2
3 //基础类
4 public class Student {
5     private String name;
6     private int age;
7
8     public Student() {}
9
10    public Student(String name, int age) {
11        this.name = name;

```



```

12         this.age = age;
13     }
14
15     public String getName() {
16         return name;
17     }
18     public void setName(String name) {
19         this.name = name;
20     }
21     public int getAge() {
22         return age;
23     }
24     public void setAge(int age) {
25         this.age = age;
26     }
27
28     @Override
29     public String toString() {
30         return "Student{" +
31             "name='" + name + '\'' +
32             ", age=" + age +
33             '}';
34     }
35 }

```

测试类:

```

1  package com.briup.chap11.test;
2
3  public class Test029_WriteObject {
4      public static void main(String[] args) throws IOException
5      {
6          //把对象保存文件
7          ObjectOutputStream oos = new ObjectOutputStream(

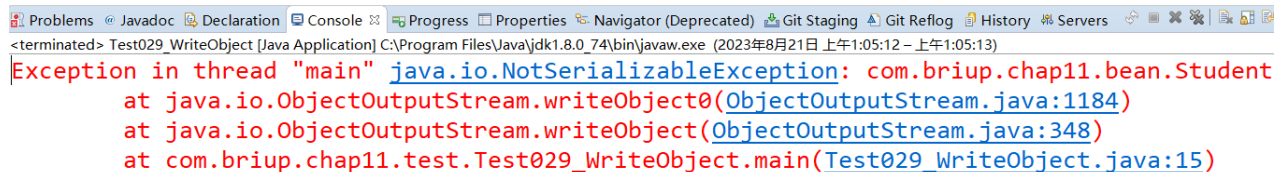
```

```

7         new
      FileOutputStream("src/com/briup/chap11/test/stu.txt"));
8         Student stu = new Student("tom",20);
9
10        oos.writeObject(stu);
11        System.out.println("writeObject success!");
12
13        //操作完成后，关闭流
14        oos.close();
15    }
16 }

```

运行代码时会抛出异常 `java.io.NotSerializableException`



```

<terminated> Test029_WriteObject [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年8月21日 上午1:05:12 - 上午1:05:13)
Exception in thread "main" java.io.NotSerializableException: com.briup.chap11.bean.Student
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184)
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348)
    at com.briup.chap11.test.Test029_WriteObject.main(Test029_WriteObject.java:15)

```

异常信息为Student的类型无法进行序列化，那Student类型还需要做哪些操作才可以完成序列化呢？

### 3) 序列化接口

在Java中，只有实现了 `Serializable` 接口的对象才可以进行进行序列化和反序列化。

`java.io.Serializable` 接口

```

1  package java.io;
2
3  public interface Serializable {
4  }

```

```

169 public interface Serializable {
170 }

```

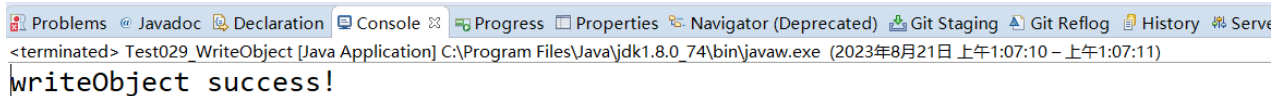
注意：这只是一个“标识”接口，接口中没有抽象方法。

## 上述问题解决：

让Student类实现序列化接口，就可以解决上述问题。

```
1 //注意，必须实现接口
2 public class Student implements Serializable {
3     //省略...
4
5 }
```

再次运行程序，成功，效果如下：



The screenshot shows an IDE window with a console tab. The console output indicates that the program has terminated successfully. The text in the console is: `<terminated> Test029_WriteObject [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年8月21日 上午1:07:10 - 上午1:07:11)` followed by `writeObject success!`.

结论，对象流操作的基础类，一定要实现序列化接口

## 读取对象案例：

从上述stu.txt文件中读取Student对象，并输出。

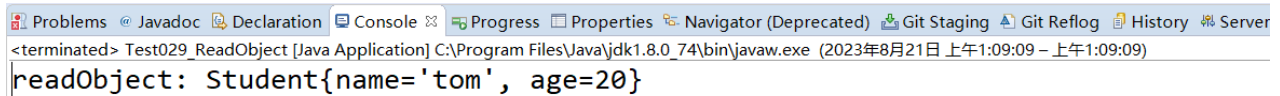
```
1 package com.briup.chap11.test;
2
3 public class Test029_ReadObject {
4     public static void main(String[] args) throws IOException,
5         ClassNotFoundException {
6         //读取文件中对象
7         ObjectInputStream ois = new ObjectInputStream(
8             new
9             FileInputStream("src/com/briup/chap11/test/stu.txt"));
```

```

8      Student s1 = (Student) ois.readObject();
9
10     System.out.println("readObject: " + s1);
11
12     //操作完成后，关闭流
13     ois.close();
14 }
15 }

```

运行效果：



The screenshot shows an IDE interface with a console window. The console output is: `readObject: Student{name='tom', age=20}`. The IDE title bar includes tabs for Problems, Javadoc, Declaration, Console, Progress, Properties, Navigator (Deprecated), Git Staging, Git Reflog, History, and Server. The console title is `<terminated> Test029_ReadObject [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年8月21日 上午1:09:09 - 上午1:09:09)`.

注意：写入对象的次数，和读取对象的次数应该一致，如果读取次数超过实际写入的对象个数，则会出现 `EOFException` 异常！

### EOFException案例：

上述案例中stu.txt被写入了1个对象，接下来我们从stu.txt中读取2次对象，观察程序运行效果。

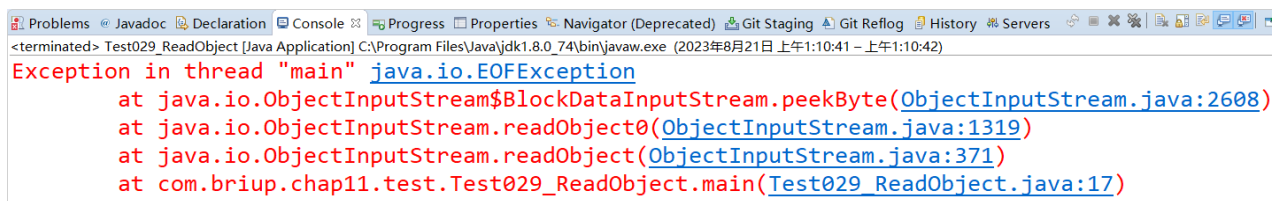
```

1  package com.briup.chap11.test;
2
3  public class Test029_ReadObject {
4      public static void main(String[] args) throws IOException,
5          ClassNotFoundException {
6          //读取文件中对象
7          ObjectInputStream ois = new ObjectInputStream(
8              new
9              FileInputStream("src/com/briup/chap11/test/stu.txt"));
10
11         //第一次读取成功
12         Student s1 = (Student) ois.readObject();
13         //第二次读取失败，因为文件中只写入了1个对象

```

```
12         Student s2 = (Student) ois.readObject();
13
14         System.out.println("readObject: " + s1);
15         System.out.println("readObject: " + s2);
16
17         //操作完成后，关闭流
18         ois.close();
19     }
20 }
```

运行效果：



The screenshot shows an IDE window with a console tab. The console displays a red error message: "Exception in thread "main" java.io.EOFException". Below the error message, the stack trace is shown, including the following lines: "at java.io.ObjectInputStream\$BlockDataInputStream.peekByte(ObjectInputStream.java:2608)", "at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1319)", "at java.io.ObjectInputStream.readObject(ObjectInputStream.java:371)", and "at com.briup.chap11.test.Test029\_ReadObject.main(Test029\_ReadObject.java:17)".

## 4) 集合序列化

实际开发中往往有以下场景：

程序员A往文件中写入多个对象；程序员B需要从该文件中读取所有对象；但B不知道文件中对象个数，这种情况下如何获取所有对象，同时避免EOFException异常？

推荐方案：

序列化多个对象时，先将所有对象添加到一个集合中，然后序列化集合对象；

反序列化时，从文件中读取单个集合对象，再从集合中获取所有对象。

## 集合序列化案例：

往list.txt中写入多个对象，然后再从文件中读取所有对象并遍历输出。

写入list.txt功能实现：

```
1  package com.briup.chap11.test;
2
3  public class Test029_WriteList {
4      public static void main(String[] args) throws Exception {
5          //1.实例化流对象
6          ObjectOutputStream oos = new ObjectOutputStream(
7              new
8              FileOutputStream("src/com/briup/chap11/test/list.txt"));
9
10         //2.准备多个Student对象并加入List集合
11         Student s1 = new Student("tom",20);
12         Student s2 = new Student("zs",21);
13         Student s3 = new Student("jack",19);
14
15         List<Student> list = new ArrayList<>();
16         list.add(s1);
17         list.add(s2);
18         list.add(s3);
19
20         //3.将集合写入文件
21         oos.writeObject(list);
22         System.out.println("write list success!");
23
24         //4.操作完成后，关闭流
25         oos.close();
26     }
27 }
28 //运行结果：
29 write list success!
```

## 读取功能实现：

```
1  package com.briup.chap11.test;
2
3  public class Test029_ReadList {
4      public static void main(String[] args) throws IOException,
5      ClassNotFoundException {
6          ObjectInputStream oos = new ObjectInputStream(
7              new
8              FileInputStream("src/com/briup/chap11/test/list.txt"));
9
10         //2.读取集合对象
11         List<Student> list = (List<Student>) oos.readObject();
12         if(list == null) {
13             System.out.println("read null");
14             return;
15         }
16
17         System.out.println("read list size: " + list.size());
18
19         //3.遍历输出
20         for (Student stu : list) {
21             System.out.println(stu);
22         }
23
24         //4.关闭资源
25         oos.close();
26     }
27 }
28
29 //运行结果:
30 read list size: 3
31 Student{name='tom', age=20}
32 Student{name='zs', age=21}
33 Student{name='jack', age=19}
```

## 5) transient

`transient` 的单词含义就是短暂的、转瞬即逝。在Java中的关键字 `transient` 可以修饰类中的属性，使对象在进行序列化的时候，忽略掉指定的属性。

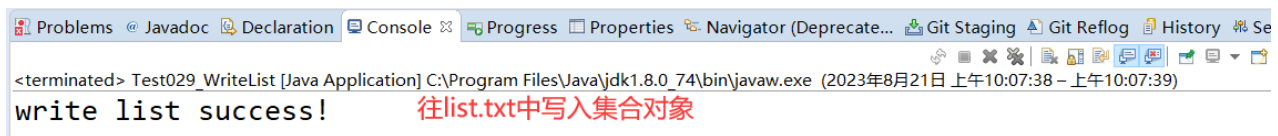
常用在一些敏感属性的修饰，例如对象中的password属性，我们并不想将这个敏感属性的值进行序列化保存，那么就可以使用 `transient` 来对他进行修饰。

案例展示：

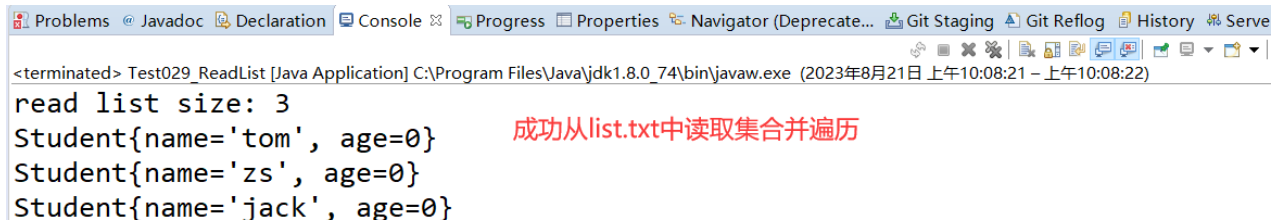
修改上面案例中Student类的属性age，添加 `transient` 修饰符测试

```
1 public class Student implements Serializable {
2     private String name;
3     private transient int age;
4
5     //省略...
6 }
```

再次运行程序，先写入集合对象到list.txt，再从list.txt读取集合对象并遍历：



```
<terminated> Test029_WriteList [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年8月21日 上午10:07:38 - 上午10:07:39)
write list success! 往list.txt中写入集合对象
```



```
<terminated> Test029_ReadList [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年8月21日 上午10:08:21 - 上午10:08:22)
read list size: 3
Student{name='tom', age=0}
Student{name='zs', age=0}
Student{name='jack', age=0} 成功从list.txt中读取集合并遍历
```

这时候，再进行序列化操作，对象中的age属性的值将会被忽略掉

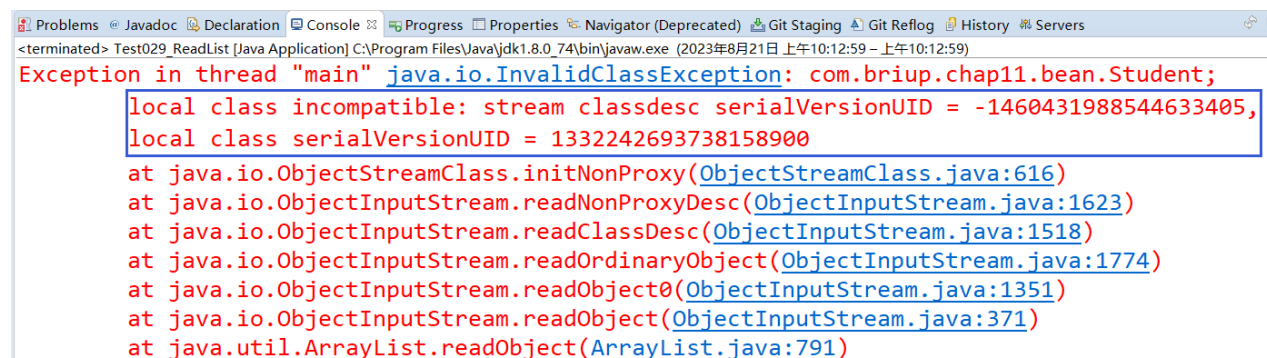
思考，查看JavaAPI的的哪些类中用到了transient关键字？



## 6) 序列化版本号

### 问题引出:

重新恢复上述Student类，去除transient关键字修饰。然后直接运行读取List集合案例。我们发现程序会抛出 `java.io.InvalidClassException`，即无效的类。详细异常信息如下：



```
<terminated> Test029_ReadList [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年8月21日 上午10:12:59 - 上午10:12:59)
Exception in thread "main" java.io.InvalidClassException: com.briup.chap11.bean.Student;
    local class incompatible: stream classdesc serialVersionUID = -1460431988544633405,
    local class serialVersionUID = 1332242693738158900
    at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:616)
    at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1623)
    at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1518)
    at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1774)
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1351)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:371)
    at java.util.ArrayList.readObject(ArrayList.java:791)
```

异常具体信息如下：

```
1 Exception in thread "main" java.io.InvalidClassException:
  com.briup.chap11.bean.Student; local class incompatible: stream
  classdesc serialVersionUID = -1460431988544633405, local class
  serialVersionUID = 1332242693738158900
```

这里出现了关键的内容 `serialVersionUID`，代表序列化版本号。

### 序列化版本号:

在 Java 中，`serialVersionUID` 是一个用于序列化的静态变量，用于标识序列化类的版本号。它是一个长整型数字，用于确保序列化和反序列化过程中类的版本一致性。

当一个类实现 `Serializable` 接口并进行序列化时，如果不显式地定义 `serialVersionUID`，系统会根据类的结构和内容自动生成一个默认的序列化版本号（根据类的名称、实现的接口、成员变量等信息计算得出）。

但是，如果类的结构发生了变化（如新增或删除了成员变量、修改了类的继承关系等），则默认的版本号会发生变化，这可能导致反序列化失败（比如上述场景）。

为了确保在类的结构发生变化时仍能成功反序列化，可以显式地定义 `serialVersionUID`。通过手动定义 `serialVersionUID`，可以确保序列化和反序列化的类版本一致，即使类的结构发生了变化。

### 上述异常解决：

给Student类添加 `serialVersionUID`，再重复上面的实验操作，就不会反序列化的时候报错了，因为版本号一直都是手动固定写死的。

- 步骤1：给Student类添加序列化ID值



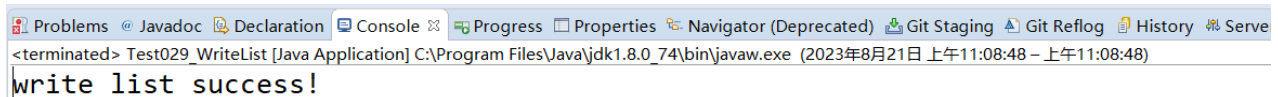
Student类添加 `serialVersionUID` 后代码如下：

```

1  public class Student implements Serializable {
2      // 默认值
3      private static final long serialVersionUID = 1L;
4      // 根据属性生成的值（二者选其一）
5      // private static final long serialVersionUID =
        -1460431988544633405L;
6
7      private String name;
8      private int age;
9      //transient private int age;
10
11     // 略...
12 }

```

- 步骤2: 运行 `Test029_WriteList` 案例



```

<terminated> Test029_WriteList [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年8月21日 上午11:08:48 – 上午11:08:48)
write list success!

```

- 步骤3: 修改Student类，给age属性添加transient修饰

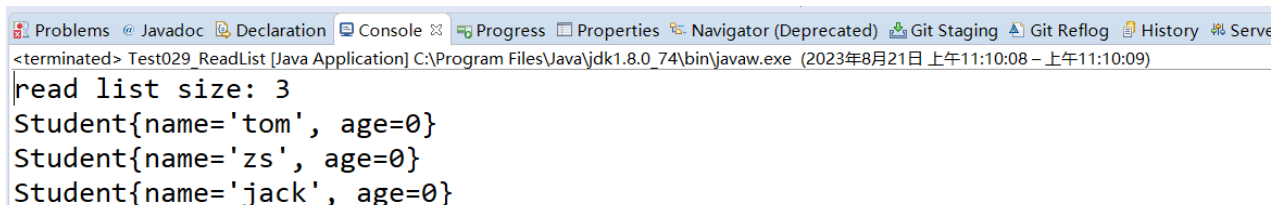
```

6  public class Student implements Serializable {
7      private static final long serialVersionUID = 1L;
8      //private static final long serialVersionUID = -1460431988544633405L;
9
10     private String name;
11     //private int age;
12     transient private int age;

```

添加transient关键字修饰

- 步骤4: 运行 `Test029_ReadList` 案例



```

<terminated> Test029_ReadList [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年8月21日 上午11:10:08 – 上午11:10:09)
read list size: 3
Student{name='tom', age=0}
Student{name='zs', age=0}
Student{name='jack', age=0}

```

程序运行成功，但age值为0，说明反序列化时已经自动忽略。

思考，查看JavaAPI的的哪些实现了这个序列化接口？

```

31 public class String extends UnaryOperation
32 {
33     static final long serialVersionUID = 2973374377453022888L;

106 public class ArrayList<E> extends AbstractList<E>
107     implements List<E>, RandomAccess, Cloneable, java.io.Serializable
108 {
109     private static final long serialVersionUID = 8683452581122892189L;

```

## 本章小结:

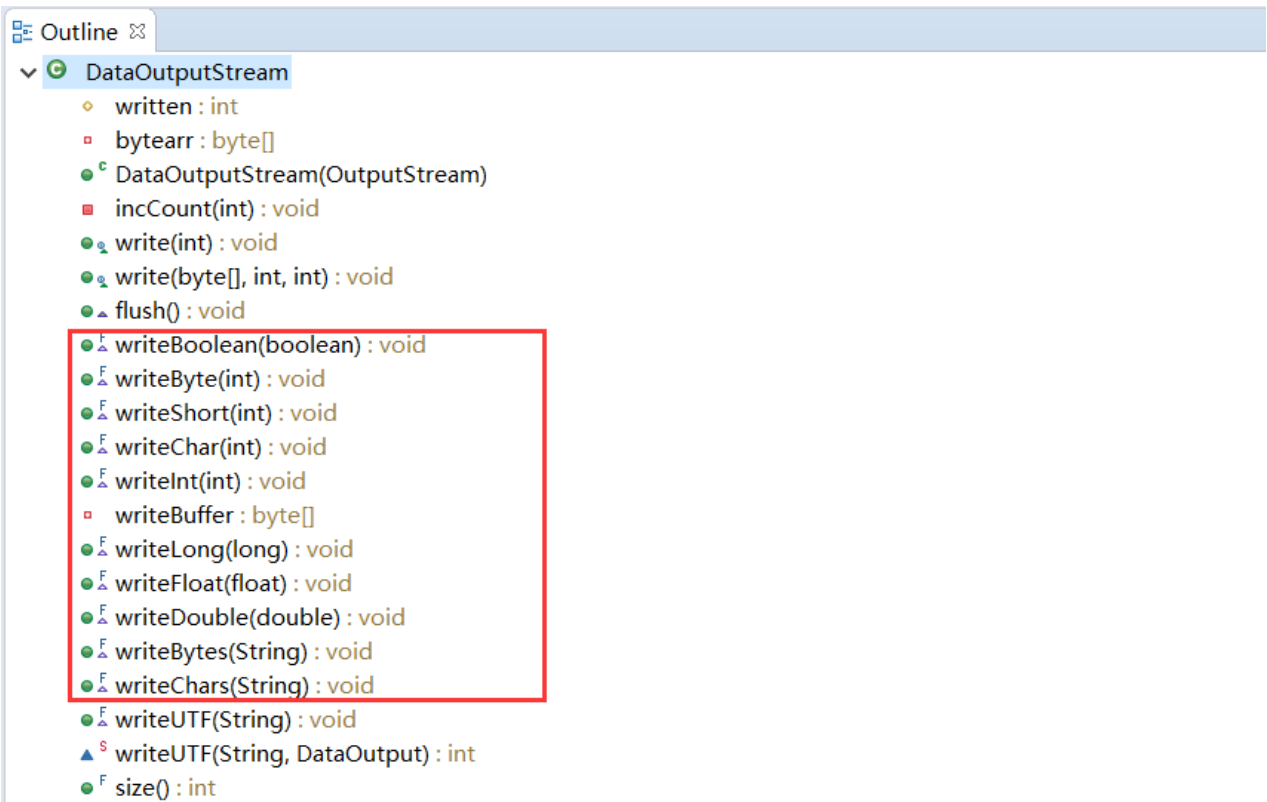
如果不显式地定义 `serialVersionUID`，并且类的结构发生了变化，可能会导致反序列化失败。因此，**建议在需要序列化的类中显式地定义 `serialVersionUID`，以确保序列化和反序列化的兼容性。**

至此，IO中关键的流都已经讨论完毕，接下来我们讨论下几个特殊场景下需要使用的IO流！

## 2.10 数据流

在 Java 中，数据流（Data Streams）提供了一种方便、高效和一致的方式来读写基本数据类型和字符串。虽然 Java 提供了字符流和字节流来处理输入和输出，但字符流主要用于处理文本数据，而字节流则用于处理二进制数据。而数据流则提供了一种专门用于读写基本数据类型和字符串的流。

`DataOutputStream` 负责把指定类型的数据，转化为字节并写出去



**DataInputStream** 负责把读取到的若干字节，转化为指定类型的数据



## 案例描述:

- 1 使用数据流完成对应基本数据类型的文件存储，然后按照对应基本数据类型读取文件。注意观察文件内容格式是否和平常文本直接存入数据是否效果一致，原因为何？

## 案例如下:

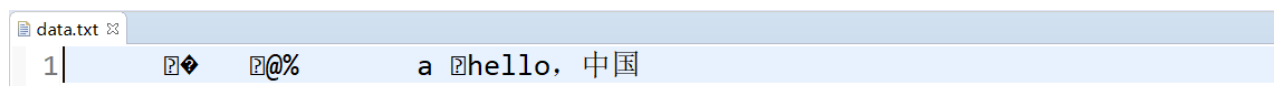
```
1 package com.briup.chap11.test;
2
3 public class Test0210_DataStream {
4     public static void main(String[] args) throws IOException
5     {
6         DataOutputStream out = null;
7         DataInputStream in = null;
8
9         // 基本数据以及字符串写入文件
10        File file = new
11        File("src/com/briup/chap11/test/data.txt");
12
13        // “包裹”文件字节输出流，增强数据写出功能
14        out = new DataOutputStream(new
15        FileOutputStream(file));
16        out.writeLong(1000L);
17        out.writeInt(5);
18        out.writeDouble(10.5D);
19        out.writeChar('a');
20        out.writeUTF("hello, 中国");
21        out.flush();
22
23        // 操作完成后及时关闭流
24        // 注意，下面输入流关联的文件和out关联文件一样，所以out一定要
25        先close
26        out.close();
27
28        // 基本数据以及字符串读取文件
```

```

25         // “包裹”文件字节输入流，增强读取数据功能
26         in = new DataInputStream(new FileInputStream(file));
27         // 注意，数据读出来的顺序要，和之前写进去的顺序一致
28         System.out.println(in.readLong());
29         System.out.println(in.readInt());
30         System.out.println(in.readDouble());
31         System.out.println(in.readChar());
32         System.out.println(in.readUTF());
33
34         in.close();
35     }
36 }
37
38 //运行结果：
39 1000
40 5
41 10.5
42 a
43 hello, 中国

```

data.txt文件内容：



## 2.11 随机访问流

`java.io.RandomAccessFile` 是JavaAPI中提供的对文件进行随机访问的流

```
1  package java.io;
2
3  public class RandomAccessFile implements DataOutput,
    DataInput, Closeable {
4      //省略...
5
6      //传入文件以及读写方式
7      public RandomAccessFile(File file, String mode) throws
    FileNotFoundException;
8
9      //跳过pos字节
10     public void seek(long pos) throws IOException;
11 }
```

**注意：** `RandomAccessFile` 并没有继承之前介绍到的那四个抽象父类型！

之前使用的每一个流，那么是读数据的，要么是写数据的。而随机访问流，它即可读取文件内容，又可以往文件中写入数据，同时它还可以任意定位到文件的某一个位置进行读写操作。

创建该类的对象时，需要指定要操作的文件和操作的模式：

- “r” 模式，以只读方式来打开指定文件夹。如果试图对该RandomAccessFile执行写入方法，都将抛出IOException异常。
- “rw” 模式，以读写方式打开指定文件。如果该文件尚不存在，则试图创建该文件。
- “rws” 模式，以读写方式打开指定文件。相对于“rw” 模式，还要求对文件内容或元数据的每个更新都同步写入到底层设备。
- “rwd” 默认，以读写方式打开指定文件。相对于“rw” 模式，还要求对文件内容每个更新都同步写入到底层设备。



程序中，以r和rw模式最为常用，r模式表示只读，rw模式表示既能读又能写

### 案例描述：

a.txt文件内容为：hello world123

使用随机访问流进行内容替换，完成后文件内容：hello briup123

### 案例如下：

```
1  package com.briup.chap11.test;
2
3  public class Test0211_RandomAccessFile {
4      public static void main(String[] args) throws IOException
5      {
6          // 1.实例化随机流对象，设为读写模式
7          File file = new
8          File("src/com/briup/chap11/test/a.txt");
9          RandomAccessFile randomAccessFile = new
10         RandomAccessFile(file, "rw");
11
12         // 文件中要替换数据的位置
13         int replacePos = 6;
14         // 文件中要插入的内容
15         String replaceContent = "briup";
16
17         // 2.定位到要替换数据的位置
18         randomAccessFile.seek(replacePos);
19         // 在指定位置，写入需要替换的内容，覆盖原来此位置上的内容
20         randomAccessFile.write(replaceContent.getBytes());
21         System.out.println("替换成功!");
22
23         // 3.定位到文件的开始位置，读文件中的所有内容，并输出到控制台
24         randomAccessFile.seek(0);
25         byte[] buf = new byte[8];
```

```

23         int len = -1;
24         while ((len = randomAccessFile.read(buf)) != -1) {
25             System.out.print(new String(buf,0,len));
26         }
27
28         //4.关闭资源
29         randomAccessFile.close();
30     }
31 }
32
33 //运行结果:
34 hello briup123

```

思考：如何向文件的某个位置插入一段数据？

## 2.12 Properties类

在 Java 中，`Properties` 类是一个用于处理配置文件的工具类。配置文件通常以 `.properties` 扩展名，用于存储配置信息，如应用程序的设置、数据库连接参数等。

`Properties` 类继承自 `Hashtable` 类，因此它具有 `Hashtable` 的所有功能，同时还提供了一些特定于属性文件的方法。

```

121 public
122 class Properties extends Hashtable<Object,Object> {
123     /**
124      * use serialVersionUID from JDK 1.1.X for interoperability
125      */
126     private static final long serialVersionUID = 4112578634029874840L;

```

### 案例展示：

通过`Properties`类加载`db.properties`文件，获取文件内容并输出，然后修改配置并写入。

db.properties文件内容如下:

```
1 driver=com.briup.Driver
2 url=127.0.0.1:8888
3 username=briup
4 password=briup
```

代码实现:

```
1 package com.briup.chap11.test;
2
3 public class Test0212_Properties {
4     public static void main(String[] args) throws IOException
5     {
6         //1.准备输入流对象
7         File file = new
8         File("src/com/briup/chap11/test/db.properties");
9         InputStream in = new FileInputStream(file);
10
11         //2.创建工具类对象
12         Properties p = new Properties();
13
14         //3.从输入流加载key-value到工具类对象中
15         p.load(in);
16
17         //4.解析key-value并输出
18         String driver = p.getProperty("driver");
19         String url = p.getProperty("url");
20         String username = p.getProperty("username");
21         String password = p.getProperty("password");
22
23         System.out.println("driver: " + driver);
24         System.out.println("url: " + url);
25         System.out.println("username: " + username);
26         System.out.println("password: " + password);
27     }
28 }
```

```

26         System.out.println("-----");
27
28         //5.如果不知道key名称的话，则获取所有key，再根据key获取对应
        value，进行遍历
29         //Enumeration<?> names = p.propertyNames();
30         Set<String> keySet = p.stringPropertyNames();
31         for (String key : keySet) {
32             System.out.println(key + ": " +
        p.getProperty(key));
33         }
34
35         //6.使用完，关闭输入流对象
36         in.close();
37
38         System.out.println("-----");
39
40         //7.修改属性值
41         p.setProperty("username", "root");
42         p.setProperty("password", "root");
43
44         //8.将工具类对象属性值，存储到指定文件中
45         OutputStream os = new FileOutputStream(file);
46         p.store(os, "this is comment msg");
47         System.out.println("存储属性值完成!");
48
49         //9.关闭资源
50         os.close();
51     }
52 }
53
54
55 //程序输出:
56 driver: com.briup.Driver
57 url: 127.0.0.1:8888
58 username: briup
59 password: briup

```

```
60  -----
61  url: 127.0.0.1:8888
62  password: briup
63  driver: com.briup.Driver
64  username: briup
65  -----
66  存储属性值完成!
```

运行后db.properties内容:

```
db.properties
1#this is comment msg
2#Mon Aug 21 13:38:59 CST 2023
3password=root
4url=127.0.0.1\:8888
5driver=com.briup.Driver
6username=root
7
```