# 第13章 I/O

I/O是输入/输出(Input/Output) 的缩写；

输入输出的参照物是当前程序。输入指的是读取数据到当前程序中，输出值得是把数据写到程序外部；

注意：IO不局限于文件、磁盘文件、内存中，包括网络也属于IO

Java 的 I/O 大概可以分成以下几类：

磁盘操作：File

字节操作：InputStream 和 OutputStream

字符操作：Reader 和 Writer

对象操作：Serializable

网络操作：Socket

新的输入/输出：NIO

## 13.1 java.io.File类

1. File类是java.io包下的；
2. File类代表与平台无关的文件和目录；
3. File类能新建、删除、重命名文件和目录。
4. 在API中File的解释是文件和目录路径名的抽象表示形式，即**File类是文件或目录的路径，而不是文件本身。**因此File类不能直接访问文件内容本身，如果需要访问文件内容本身，则需要使用输入/输出流。
5. 对于File对象来说，输入和输出操作只能针对文件，目录（文件夹）是不可的。

### 13.1.1 路径名

提示：File类代表磁盘或网络中某个文件或目录的路径名称，如：

/atguigu/javase/io/佟刚.jpg

但他不能直接通过File对象读取和写入数据，如果要操作数据，需要IO流。就好比“地址”不代表“水库”，要“存取”里面的水到你“家里”，需要“管道”。

File类可以使用文件路径字符串来创建File实例，该文件路径字符串既可以是绝对路径，也可以是相对路径，默认情况下，系统总是依据用户的工作路径来解释相对路径，这个路径由系统属性“user.dir”指定，通常也就是运行Java虚拟机时所作的路径。

1. String getName()：返回此File对象的所表示的文件名（返回最后一级）
2. String getPath()：返回此File对象所对应的路径名。
3. String getAbsolutePath()：返回此File对象所对应的绝对路径名。
4. File getAbsoluteFile()：返回此File对象所对应的绝对路径名所对应的File对象
5. String getCanonicalPath()：返回此File对象所对应的规范路径名。所谓规范路径名，即对路径中的“..”等进行解析后的路径名。
6. File getCanonicalFile()：返回此File对象所对应的规范路径名所对应的File对象
7. String getParent()：返回此File对象的父目录名
8. File getParentFile()：返回此File对象的父目录名所对应的File对象

|  |
| --- |
| @Test  **public** **void** test1() **throws** IOException{  File f1 = **new** File("d:\\atguigu\\javase\\HelloIO.java");  System.***out***.println("文件/目录的名称：" + f1.getName());  System.***out***.println("文件/目录的路径名：" + f1.getPath());  System.***out***.println("文件/目录的绝对路径名：" + f1.getAbsolutePath());  System.***out***.println("文件/目录的规范路径名：" + f1.getCanonicalPath());  System.***out***.println("文件/目录的父目录名：" + f1.getParent());  } |
| 文件/目录的名称：HelloIO.java  文件/目录的路径名：d:\atguigu\javase\HelloIO.java  文件/目录的绝对路径名：d:\atguigu\javase\HelloIO.java  文件/目录的规范路径名：D:\atguigu\javase\HelloIO.java  文件/目录的父目录名：d:\atguigu\javase |
| @Test  **public** **void** test2() **throws** IOException{  File f2 = **new** File("HelloIO.java");  System.out.println("user.dir =" + System.getProperty("user.dir"));  System.***out***.println("文件/目录的名称：" + f2.getName());  System.***out***.println("文件/目录的路径名：" + f2.getPath());  System.***out***.println("文件/目录的绝对路径名：" + f2.getAbsolutePath());  System.***out***.println("文件/目录的规范路径名：" + f2.getCanonicalPath());  System.***out***.println("文件/目录的父目录名：" + f2.getParent());  } |
| user.dir =D:\atguigu\temp\JDK18\test\_io  文件/目录的名称：HelloIO.java  文件/目录的路径名：HelloIO.java  文件/目录的绝对路径名：D:\atguigu\temp\JDK18\test\_io\HelloIO.java  文件/目录的规范路径名：D:\atguigu\temp\JDK18\test\_io\HelloIO.java  文件/目录的父目录名：null |
| @Test  **public** **void** test3() **throws** IOException{  File f3 = **new** File("../../HelloIO.java");  System.out.println("user.dir =" + System.getProperty("user.dir"));  System.***out***.println("文件/目录的名称：" + f3.getName());  System.***out***.println("文件/目录的路径名：" + f3.getPath());  System.***out***.println("文件/目录的绝对路径名：" + f3.getAbsolutePath());  System.***out***.println("文件/目录的规范路径名：" + f3.getCanonicalPath());  System.***out***.println("文件/目录的父目录名：" + f3.getParent());  } |
| user.dir =D:\atguigu\temp\JDK18\test\_io  文件/目录的名称：HelloIO.java  文件/目录的路径名：..\..\HelloIO.java  文件/目录的绝对路径名：D:\atguigu\temp\JDK18\test\_io\..\..\HelloIO.java  文件/目录的规范路径名：D:\atguigu\temp\HelloIO.java  文件/目录的父目录名：..\.. |

* 路径分隔符：
* 在 UNIX、Linux、Mac 系统中：正斜杠 /
* window的路径分隔符使用“\”

|  |
| --- |
| Java程序中的“\”表示转义字符，所以在Windows中表示路径，需要用“\\”;或者直接使用“/”也可以；  Java程序支持将“/”当成平台无关的路径分隔符。 |

* 直接使用File.separator常量值表示，直接忽略操作系统差异
* 路径中如果出现“..”表示上一级目录，路径名如果以“/”开头，表示从“根目录”下开始导航。
* 路径分隔符（path separator）：File.pathSeparator
* 在 UNIX、Linux、Mac 系统中：冒号:
* 在 Windows 系统中：分号;
* 是否区分大小写
* 在 Windows、Mac 系统中：文件名、目录名不区分大小写
* 在 UNIX、Linux 系统中：文件名、目录名区分大小写

### 13.1.2 获取常规信息等方法

文件和目录检测方法：

* boolean exists()：判断File对象对应的文件或目录是否存在
* canRead()：判断File对象对应的文件或目录是否可读
* canWrite()：判断File对象对应的文件或目录是否可写
* isHidden()：判断File对象对应的文件或目录是否是否隐藏
* isFile()：判断File对象对应的是否是文件
* isDirectory()：判断File对象对应的是否是目录
* isAbsolute()：判断File对象对应的文件或目录是否是绝对路径

获取常规信息：

* long lastModified()：返回File对象对应的文件或目录的最后修改时间（毫秒值）
* long length()：返回File对象对应的文件的内容的长度**（字节数）**，**如果File对象对应的是目录，则结果是不确定的**。

以上都是对应于File对象的属性信息，如果File对象对应的文件或目录不存在，返回的都是对应属性的默认值，例如：length()返回为0，isFile()和isDirectory()返回为false等。

|  |
| --- |
| @Test  **public** **void** test4(){  File file = **new** File("d:/atguigu/javase/HelloIO.java");  **long** time = file.lastModified();  SimpleDateFormat sf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");  String format = sf.format(**new** Date(time));  System.***out***.println("最后修改时间：" + format);  System.**out**.println("文件大小：" + file.length());  } |

### 13.1.3 操作文件API

#### 1、创建文件

(1)创建文件：boolean createNewFile() throws IOException

如果指定的文件不存在并成功地创建，则返回 true；如果指定的文件已经存在，则返回 false。

代码示例：

|  |
| --- |
| @Test  **public** **void** test5() **throws** IOException{  File file = **new** File("d:/atguigu/javase/HelloIO.java");  file.createNewFile();  } |

(2)创建临时文件，通常会结合deleteOnExit()使用

* public static File createTempFile(String prefix,String suffix) throws IOException在默认临时文件目录中创建一个空文件，使用给定前缀和后缀生成其名称。调用此方法等同于调用 createTempFile(prefix, suffix, null)。
  + prefix - 用于生成文件名的前缀字符串；必须至少三个字符。
  + suffix - 用于生成文件名的后缀字符串；如果为 null，默认为 ".tmp"

1. public static File createTempFile(String prefix,String suffix,File directory)throws IOException

在指定目录中创建一个新的空文件，使用给定的前缀和后缀字符串生成其名称。

* + prefix - 用于生成文件名的前缀字符串；必须至少三个字符。
  + suffix - 用于生成文件名的后缀字符串；如果为 null，默认为 ".tmp"
  + directory - 将创建的文件所在的目录；如果使用默认临时文件目录，则该参数为 null

|  |
| --- |
| @Test  **public** **void** test6() **throws** IOException{  File tempFile = File.*createTempFile*("Hello", ".tmp");  System.***out***.println(tempFile.getAbsolutePath());  //C:\Users\Irene\AppData\Local\Temp\Hello2541030191749214481.tmp  } |

#### 2、删除文件

boolean delete() 当且仅当成功删除文件时，返回 true；否则返回 false。

public void deleteOnExit()：当退出JVM时，删除文件，一般用于删除临时文件

|  |
| --- |
| @Test  **public** **void** test7() {  File file = **new** File("d:/atguigu/javase/HelloIO.java");  file.delete();  } |
| @Test  **public** **void** test8() **throws** IOException{  File tempFile = File.*createTempFile*("Hello", ".tmp");  //....省略代码  tempFile.deleteOnExit();  } |

#### 3、文件重命名

boolean renameTo(File dest)：此方法行为的许多方面都是与平台有关的：重命名操作无法将一个文件从一个文件系统移动到另一个文件系统，该操作不是不可分的，如果已经存在具有目标抽象路径名的文件，那么该操作可能无法获得成功。应该始终检查返回值，以确保重命名操作成功。

|  |
| --- |
| @Test  **public** **void** test9(){  File src = **new** File("d:/atguigu/javase/HelloIO.java");  File dest = **new** File("d:/atguigu/javase/HelloFile.java");  src.renameTo(dest);  } |
| @Test  **public** **void** test10() {  File src = **new** File("d:/atguigu/javase/HelloIO.java");  File dest = **new** File("e:/HelloFile.java");  //D盘和E盘相同的文件系统可以成功，例如都是NTFS。  src.renameTo(dest);  } |

### 13.1.4 操作目录API

#### 1、创建目录

* boolean mkdir() ：必须确保父目录存在，否则创建失败
* boolean mkdirs()：如果父目录链不存在，会一同创建父目录链

|  |
| --- |
| @Test  **public** **void** test1(){  File dir = **new** File("d:/atguigu/javase/io");  dir.mkdir();  //如果d:/atguigu/javase不存在不会报错，只是没创建成功  }  @Test  **public** **void** test2() {  File dir = **new** File("d:/atguigu/javase/io");  dir.mkdirs();  //如果d:/atguigu/javase不存在，会一并创建  } |

#### 2、列出目录内容☆

* public String[] list()列出当前目录的下级目录或文件的名称
* public File[] listFiles() 列出当前目录的下级的目录或文件对应的File对象

配合递归 可以列出下级的下级....

|  |
| --- |
| @Test  **public** **void** test3() {  File dir = **new** File("d:/atguigu");  listSubFiles(dir);  }  **public** **void** listSubFiles(File dir) {  **if** (dir != **null** && dir.isDirectory()) {  File[] listFiles = dir.listFiles();  **if** (listFiles != **null**) {  **for** (File sub : listFiles) {  listSubFiles(sub);  }  }  }  System.***out***.println(dir);  } |

代码示例：求目录总大小

|  |
| --- |
| @Test  **public** **void** test4() {  File dir = **new** File("D:/atguigu");  **long** length = getLength(dir);  System.***out***.println("大小：" + length);  }    **public** **long** getLength(File dir){  **if** (dir != **null** && dir.isDirectory()) {  File[] listFiles = dir.listFiles();  **if**(listFiles!=**null**){  **long** sum = 0;  **for** (File sub : listFiles) {  sum += getLength(sub);  }  **return** sum;  }  }**else** **if**(dir != **null** && dir.isFile()){  **return** dir.length();  }  **return** 0;  } |

* public File[] listFiles(FileFilter filter)返回所有满足指定过滤器的文件和目录。

如果给定 filter 为 null，则接受所有路径名。否则，当且仅当在路径名上调用过滤器的 FileFilter.accept(java.io.File) 方法返回 true 时，该路径名才满足过滤器。如果当前File对象不表示一个目录，或者发生 I/O 错误，则返回 null。

* public File[] listFiles(FilenameFilter filter)返回所有满足指定过滤器的文件和目录。

如果给定 filter 为 null，则接受所有路径名。否则，当且仅当在路径名上调用过滤器的 FilenameFilter.accept(java.io.File, java.lang.String) 方法返回 true 时，该路径名才满足过滤器。如果当前File对象不表示一个目录，或者发生 I/O 错误，则返回 null。

* public static File[] listRoots()列出可用的文件系统根。

代码示例：列出“D:/atguigu/javaee/JavaSE20180808”该目录以及子目录下所有的.java文件

|  |
| --- |
| @Test  **public** **void** test5() {  File dir = **new** File("D:/atguigu/javaee/JavaSE20180808");  listByFileFilter(dir);  }    **public** **void** listByFileFilter(File file) {  **if** (file != **null** && file.isDirectory()) {  File[] listFiles = file.listFiles(**new** FilenameFilter() {  @Override  **public** **boolean** accept(File dir, String name) {  **return** name.endsWith(".java") || **new** File(dir,name).isDirectory();  }  });  **if** (listFiles != **null**) {  **for** (File sub : listFiles) {  **if**(sub.isFile()){  System.***out***.println(sub);  }  listByFileFilter(sub);  }  }  }  } |

#### 3、删除目录

* boolean delete()：只能删除空目录。否则需要先将目录下的所有内容删除才能将该目录删除。

示例代码：删除"D:/atguigu/javase"目录，包括它的下一级

|  |
| --- |
| @Test  **public** **void** test6() {  File dir = **new** File("D:/atguigu/javase");  forceDeleteDir(dir);  }  **public** **void** forceDeleteDir(File dir) {  **if** (dir != **null** && dir.isDirectory()) {  File[] listFiles = dir.listFiles();  **if**(listFiles!=**null**){  **for** (File sub : listFiles) {  forceDeleteDir(sub);  }  }  }  dir.delete();  } |

#### 4、重命名目录(移动目录)

* boolean renameTo(File dest)：如果是windows目录，不能够跨盘符，只能在同一个盘下，不能从D盘移动到E盘。因此改名相当于修改文件的路径

示例代码：

|  |
| --- |
| @Test  **public** **void** test7() {  File dir = **new** File("D:/atguigu/javase");  File dest = **new** File("D:/atguigu/java代码");  dir.renameTo(dest);  } |
| @Test  **public** **void** test8() {  File dir = **new** File("D:/atguigu/javase");  File dest = **new** File("D:/temp");  dir.renameTo(dest);  } |

## 13.2 IO流

### 13.2.1 java.io包中的IO流分类

**第一：**IO流四大超级父类，抽象基类。

|  |
| --- |
| * 字节输入流：InputStream * 字节输出流：OutputStream * 字符输入流：Reader * 字符输出流：Writer |

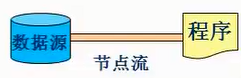
**第二：IO流类的设计选用了“装饰者”设计模式，即IO流分为两大类，“被装饰”的组件和“装饰”的组件。**

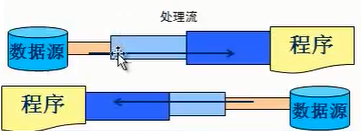
|  |
| --- |
| 例如：以InputStream为例  其中FileInputStream、ByteArrayInputStream等是“被装饰”的组件，依次用来连接和读取“文件”、“内存中的字节数组”的等。  而BufferedInputStream、DataInputStream、ObjectInputStream等是用来“装饰”的组件，依次是给其他InputStream的IO流提供装饰的辅助功能的，依次可以增加“提高效率的缓冲功能”、“按照Java数据类型读取数据的能力”、“读取并恢复Java对象的能力”等  你会发现OutputStream、Reader、Writer系列的流设计方式也是一样的。 |

* IO流的选取可以通过以下几个分类来简化选取过程。

|  |
| --- |
| * **按照IO流的方向：输入流和输出流** * I：代表Input * O：代表Output   Java的IO流是单向的，只能从输入流（Input、Reader）中读取（read）数据，也只能往输出流（Output、Writer）中写（write、print）出数据。   * **按照IO流的处理数据的基本单位分：** * 字节流（XxxStream）：以Byte字节为单位；一个字节一个字节处理，它适用于一切数据，包括纯文本、doc、xls、图片、音频、视频等等 * 字符流（XxxReader和XxxWriter）：以字符Char为单位；一个字符一个字符处理，只能处理纯文本文件。 * **按照角色分：节点流、处理流** * 节点流：连接源头、目的地，即**被**装饰者IO流 * 处理流：增强功能，提高性能，即装饰者IO流 |

**节点流处于IO操作的第一线，所有操作必须通过他们进行；处理流是通过包装节点流来完成功能的，处理流可以增加很多层。处理流必须依赖和包装节点流，而不能单独存在。**





|  |
| --- |
| 提示：装饰模式（Decorator Pattern）也称为包装模式（Wrapper Pattern），其使用一种对客户端透明的方式来动态地扩展对象的功能，它是通过继承扩展功能的替代方案之一。在现实生活中你也有很多装饰者的例子，例如：人需要各种各样的衣着，不管你穿着怎样，但是，对于你个人本质来说是不变的，充其量只是在外面加上了一些装饰，有，“遮羞的”、“保暖的”、“好看的”、“防雨的”.... |

### 13.2.2 常见IO流

|  |  |  |
| --- | --- | --- |
| **文件IO流**  **（节点流）** | 从文件读取 | FileInputStream |
| FileReader |
| 写入到文件 | FileOutputStream |
| FileWriter |
| **缓冲IO流**  （装饰流） | 从其他输入流中读取 | BufferedInputStream |
| BufferedReader |
| 写入到其他输出流中 | BufferedOutputStream |
| BufferedWriter |
| **转换流**  （装饰流） | 从其他输入流读取，并解码 | InputStreamReader |
| 写入到其他输出流中，并编码 | OutputStreamWriter |
| **数据流**  （装饰流） | 从底层输入流中读取基本 Java 数据类型 | DataInputStream |
| 将基本 Java 数据类型写入输出流中 | DataOutputStream |
| **对象流**  （节点流） | 序列化 | ObjectOutputStream |
| 反序列化 | ObjectInputStream |
| **打印流** |  | PrintStream |
|  | PrintWriter |
| 其他 | ...... |  |

### 13.2.3 抽象基类的常用方法

#### 1、InputStream

1. **int read()**

* 从输入流中读取数据的下一个字节。
* 一个字节是8bit,因此返回值是 0 到 255 范围内的 int 值。
* 如果因为已经到达流末尾而没有可用的字节，则返回值 -1。

1. **int read(byte[] b)**

* 从此输入流中将最多 b.length 个字节的数据读入一个 byte 数组中。
* 从source读取字节数据，放进byte数组b中，从b[0]开始存储读取进来的字节数据；
* 如果因为已经到达流末尾而没有可读的字节，则返回值 -1。否则以整数形式返回实际读取的字节数。

1. **int read(byte[] b, int off,int len)**

* 尝试读取 len 个字节写入byte数组，但实际上读取的字节也可能小于该值。
* 从b[off]开始存储
* 以整数形式返回实际读取的字节数。如果因为流位于文件末尾而没有可用的字节，则返回值 -1.

1. **public void close() throws IOException**

关闭此输入流并释放与该流关联的所有系统资源。

#### 2、OutputStream

1. void write(int b)

* 将指定的字节写入此输出流。
* write 的常规协定是：向输出流写入一个字节。要写入的字节是参数 b 的八个低位。b 的 24 个高位将被忽略。 即写入0~255范围的

1. void write(byte[] b)

* 将byte数组b中的所有字节从流中写到指定的sink
* write(b) 的常规协定是：应该与调用 write(b, 0, b.length) 的效果完全相同。

1. void write(byte[] b,int off,int len)

* 将byte数组b的内容，从b[0]开始写lne个字节写到指定的sink

1. public void flush()throws IOException

刷新此输出流并强制写出所有缓冲的输出字节，调用此方法指示应将这些字节立即写入它们预期的目标。

1. public void close() throws IOException

关闭此输出流并释放与该流关联的所有系统资源。

#### 3、Reader

(1)int read()

* 读取一个字符。
* 返回的是该字符对应的Unicode编码值。范围在 0 到 65535 之间 (0x00-0xffff)（2个字节的Unicode码）
* 如果已到达流的末尾，则返回 -1

(2)int read(char[] cbuf)

* 将字符读入数组。
* 返回本次读取的字符数。受限于数组长度
* 如果已到达流的末尾，则返回 -1。

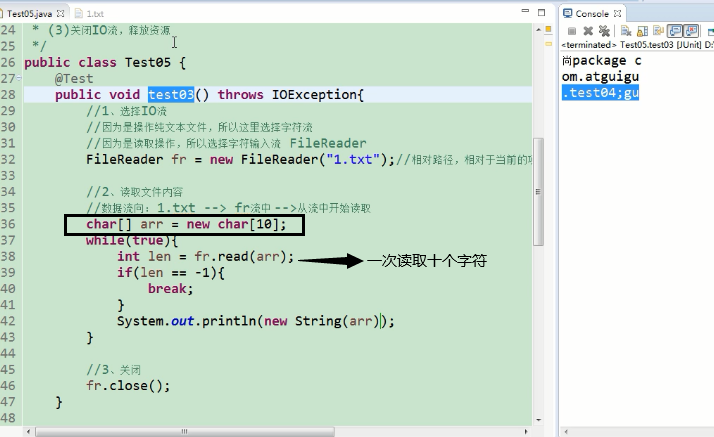
(3)int read(char[] cbuf,int off,int len)

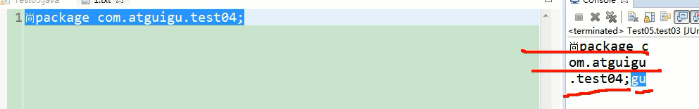
* 将字符读入数组的某一部分。存到数组cbuf中，从off处开始存储，最多读len个字符。
* 如果已到达流的末尾，则返回 -1。
* 否则返回本次读取的字符数。

(4)public void close() throws IOException

关闭此输入流并释放与该流关联的所有系统资源。

**例1：**



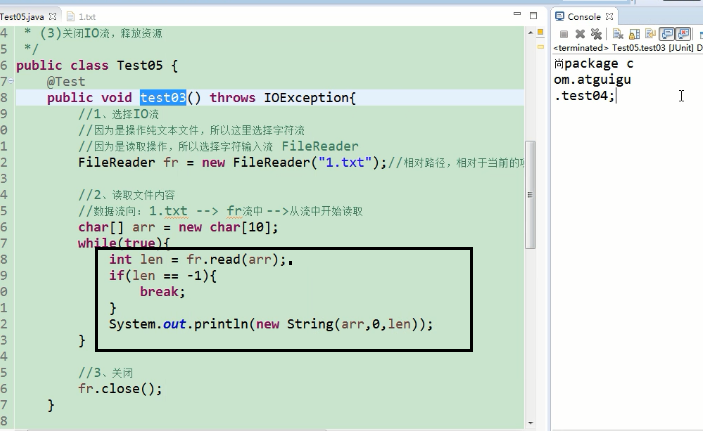


**存在的问题：**

最后一次读取到数组中，最后一次会多出上一次读取的。这是因为一直在使用一个数组，最后一次读取的并不满足十个，没有覆盖掉上次读取的

**如何改进？**

**new String(arr,0,len)**



#### 4、Writer

1. **void write(int c)、Writer append(char c)**

* 写出单个字符。
* 只能是2byte字符，要写入的字符包含在给定整数值的 16 个低位中，16 高位被忽略。 即写入0 到 65535 之间的Unicode码。

1. **void write(char[] cbuf)、Writer append(CharSequence csq)**

* 写出整个字符数组。

1. **void write(char[] cbuf,int off,int len)、Writer append(CharSequence csq, int start, int end)**

* 写入字符数组的某一部分。从off开始，写入len个字符

1. **void write(String str)**

* 写入字符串。

1. **void write(String str,int off,int len)**

写入字符串的某一部分。

1. **void flush()**

刷新该流的缓冲，则立即将它们写入预期目标。

1. **public void close() throws IOException**

关闭此输出流并释放与该流关联的所有系统资源。

#### 5、Scanner类

可以从很多source中读取数据，键盘输入，文件、流都可以。

##### 1.Scanner类构造器

|  |
| --- |
|  |

构造器参数有： File类、InputStream类 这意味着可以从file或者输入流中扫描数据

|  |
| --- |
| //构造器参数可以是InputStream  //System.in就是InputStream类型，默认从键盘输入获取数据  public Scanner(InputStream source) {  this(new InputStreamReader(source), *WHITESPACE\_PATTERN*); } |

##### 2.Scanner常用方法

1.public void close(): 关闭Scanner类

2.public boolean hasNext() : 继承自Iterator接口，用于遍历输入

3.public String next()： 遇到空白符结束

4.public boolean hasNextLine()

5.public String nextLine() 遇到换行符结束

6.public boolean hasNextBoolean()

7.public boolean hasNextInt()

8.public boolean haseNextShort()

...haseNextXXX() 对应nextXXX() XXX是基础数据类型

### 13.2.4 IO流使用步骤

#### 1、读取/接收数据的步骤

第一步：选择IO流

**1、选择节点流**

明确你要从哪里读取数据，即数据源是什么？

如果从文件中，就选择FileInputStream、FileReader；

如果从内存的数组中，就选择ByteArrayInputStream、CharArrayReader；

如果从网络中，就选择InputStream；

**2、选择字节流还是字符流**

如果是纯文本数据，并且文本的编码方式与当前平台的编码方式一致，那么选择字符流的效率更高，否则选择字节流。

**3、是否需要增加额外的辅助功能**

缓冲功能：BufferedReader、BufferedInputStream

是否需要按行读：BufferedReader、Scanner

是否需要以与机器无关方式从底层输入流中读取基本 Java 数据类型：DataInputStream

是否需要对象的反序列化：ObjectInputStream

是否需要对流中的数据进行解码的转换：InputStreamReader

第二步：循环读取数据

* read()
* readLine()
* readXxx()，例如：readInt()，readObject()，readUTF()等

第三步：关闭close()

#### 2、写/发送数据的步骤

第一步：选择IO流

1、选择节点流

明确你要把数据写到哪里去，即数据的目的地是哪里？

如果写到文件中，就选择FileOutputStream、FileWriter；

如果写到内存某数组中，就选择ByteArrayOutputStream、CharArrayWriter；

如果发送到网络中，就选择OutputStream；

2、选择字节流还是字符流

如果可选，那么看数据是否是纯文本，如果是纯文本并且文本的编码方式与当前平台的编码方式一致，那么选择字符流的效率更高，否则选择字节流。

3、是否需要增加额外的辅助功能

缓冲功能：BufferedOutputStream、BufferedWriter

是否需要按行写：PrintStream、PrintWriter

是否需要以以适当方式将基本 Java 数据类型写入输出流：DataOutputStream

是否需要将对象的序列化：ObjectOutputStream

是否需要对流中的数据进行编码的转换：OutputStreamWriter

第二步：循环写数据

* write(xx)
* print(xx)和println(xx)
* writeXxx：例如：writeInt(xx)、writeDouble(xx)、writeObject(xx)等

第三步：关闭IO流close()

## 13.3 案例

### 13.3.1 输出纯文本数据

案例：把用户的留言保存到message.txt文件中，从键盘输入留言

|  |
| --- |
| **package** com.atguigu.part2.io;  **import** java.io.FileWriter;  **import** java.io.IOException;  **import** java.util.Scanner;  **public** **class** TestFileWriter {  **public** **static** **void** main(String[] args) {  Scanner input = **new** Scanner(System.***in***);  FileWriter fw = **null**;    **try** {  fw = **new** FileWriter("message.txt",true);//true表示追加模式，默认是覆盖模式    **while**(**true**){  System.***out***.println("请输入留言：");  String message = input.nextLine();    **if**("stop".equals(message)){  **break**;  }  fw.write(message);  System.***out***.println("结束留言，请输入stop");  }  } **catch** (IOException e) {  e.printStackTrace();  } **finally**{  input.close();  **try** {  fw.close();  } **catch** (IOException e) {  }  }  }  } |

### 13.3.2 读取纯文本数据

案例：读取用户留言信息message.txt文件中的内容

|  |
| --- |
| **package** com.atguigu.part2.io;  **import** java.io.FileReader;  **import** java.io.IOException;  **public** **class** TestFileReader {  **public** **static** **void** main(String[] args) {  FileReader fr = **null**;  **try** {  fr = **new** FileReader("message.txt");    **char**[] data = **new** **char**[1024];  **int** len;  **while**((len = fr.read(data)) != -1){  System.***out***.println(**new** String(data,0,len));  }    } **catch** (IOException e) {  e.printStackTrace();  }**finally**{  **try** {  fr.close();  } **catch** (IOException e) {  }  }  }  } |

### 13.3.3 缓冲流按行读取

案例：读取用户留言信息message.txt文件中的内容

BufferedReader会将换行符作为一个标识符，readLine()只会读取一个换行符之前的内容，从而实现按行读取；

|  |
| --- |
| **package** com.atguigu.part2.io;  **import** java.io.BufferedReader;  **import** java.io.FileReader;  **import** java.io.IOException;  **public** **class** TestBufferedReader {  **public** **static** **void** main(String[] args) {  BufferedReader br = **null**;  **try** {  br = **new** BufferedReader(**new** FileReader("message.txt"));    String str;  **while**((str = br.readLine())!=**null**){  System.***out***.println(str);  }    } **catch** (IOException e) {  e.printStackTrace();  }**finally**{  **try** {  br.close();  } **catch** (IOException e) {  }  }  }  } |

BufferedReader底层默认封装了一个8192个字符容量的char数组

### 13.3.4 FileInputStream实现复制文件

文件IO流，和文件的数据传输：

（1）文件字节输入流：FileInputStream

（2）文件字节输出流：FileOutputStream

**（3）文件字符输入流：FileReader，只能按照平台默认的编码方式读取数据**

**（4）文件字符输出流：FileWriter，只能按照平台默认的编码方式写入数据**

**(FileReader和FileWriter的局限性)**

注意：

在写入一个文件时，如果目录下有同名文件将被覆盖，除非是追加模式。

在读取文件时，必须保证该文件已存在，否则出异常。

案例：将文件”atguigu/file/1.jpg”复制”atguigu/img/1.jpg”

|  |
| --- |
| **package** com.atguigu.part2.io;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **public** **class** TestCopy {    **public** **static** **void** main(String[] args) {  **try** {  *copyFile*(**new** File("atguigu/file/1.jpg"), **new** File("atguigu/img/1.jpg"));  System.***out***.println("文件复制成功");  } **catch** (IOException e) {  System.***out***.println("文件复制失败");  }  }    **public** **static** **void** copyFile(File src, File dest) **throws** IOException{  **if**(!src.isFile()){  **throw** **new** RuntimeException(src.getPath() + "不存在");  }  FileInputStream fis = **new** FileInputStream(src);  FileOutputStream fos = **new** FileOutputStream(dest);    **byte**[] data = **new** **byte**[1024];  **int** len;  **while**((len = fis.read(data))!=-1){  fos.write(data, 0, len);  }    fos.close();  fis.close();  }  } |

### 13.3.5 缓冲流实现复制文件

案例：将文件”atguigu/download/经典视频.zip”复制到”atguigu/share/值得看.zip”

使用缓冲流，可以提高效率。**缓冲流的默认缓冲区大小是8192字节/字符。**

|  |
| --- |
| **package** com.atguigu.part2.io;  **import** java.io.BufferedInputStream;  **import** java.io.BufferedOutputStream;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **public** **class** TestBufferedCopy{    **public** **static** **void** main(String[] args) {  **try** {  *copyFileInHighSpeed*(**new** File("atguigu/download/经典视频.zip"), **new** File("atguigu/share/值得看.zip"));  System.***out***.println("文件复制成功");  } **catch** (IOException e) {  System.***out***.println("文件复制失败");  }  }    **public** **static** **void** copyFileInHighSpeed(File src, File dest) **throws** IOException{  **if**(!src.isFile()){  **throw** **new** RuntimeException(src.getPath() + "不存在");  }  BufferedInputStream bis = **new** BufferedInputStream(**new** FileInputStream(src));  BufferedOutputStream bos = **new** BufferedOutputStream(**new** FileOutputStream(dest));    **byte**[] data = **new** **byte**[1024];  **int** len;  **while**((len = bis.read(data))!=-1){  bos.write(data, 0, len);  }    bis.close();  bos.close();  }  } |

* 过程如下： 在读取的时候，先按照1024 byte为单位，将字节数据写到缓冲区中，当缓冲区满了或者强制调用flush方法，从缓冲区一次性输出到目的地

|  |
| --- |
| 无缓冲区：    有缓冲区：    无缓冲区： |

* **当使用字节流和包装流的时候，先创建字节流，在创建包装流；关闭的时候，先关闭包装流，在关闭字节流；**

### 13.3.6 读写纯文本数据的同时进行编码和解码

当要读取指定编码的纯文本数据时，可以使用InputStreamReader进行解码。所谓解码，即把二进制的字节序列按照指定字符编码解码为可以被正确识别的字符内容。

案例：当前系统平台的字符编码方式是：GBK，需要读取一个UTF-8的文件内容，到当前系统中。

|  |
| --- |
| **package** com.atguigu.part2.io;  **import** java.io.BufferedReader;  **import** java.io.FileInputStream;  **import** java.io.IOException;  **import** java.io.InputStreamReader;  **public** **class** TestDecode {  **public** **static** **void** main(String[] args) {  BufferedReader br = **null**;  **try** {  br = **new** BufferedReader(**new** InputStreamReader(**new** FileInputStream("test\_utf8.txt")));    String str;  **while**((str = br.readLine())!=**null**){  System.***out***.println(str);  }  } **catch** (IOException e) {  e.printStackTrace();  } **finally**{  **try** {  br.close();  } **catch** (IOException e) {  }  }  }  } |

同理，要将纯文本数据，按照指定编码转成字节序列时，可以选择OutputStreamWriter。

案例：编写程序实现将一个编码为“GBK”的纯文本文件test.txt，复制为编码是“UTF-8”的文件other.txt。

|  |
| --- |
| **package** com.atguigu.part2.io;  **import** java.io.BufferedReader;  **import** java.io.BufferedWriter;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.InputStreamReader;  **import** java.io.OutputStreamWriter;  **public** **class** TestEncode {  **public** **static** **void** main(String[] args) **throws** IOException {  *copy*(**new** File("test.txt"),"GBK",**new** File("other.txt"),"UTF-8");  }    **public** **static** **void** copy(File src, String srcCharset, File dest, String destCharset) **throws** IOException{  **if**(!src.isFile()){  **throw** **new** RuntimeException(src.getPath() + "不存在");  }  BufferedReader br = **new** BufferedReader(**new** InputStreamReader(**new** FileInputStream(src),srcCharset));  BufferedWriter bw = **new** BufferedWriter(**new** OutputStreamWriter(**new** FileOutputStream(dest),destCharset));    String str;  **while**((str = br.readLine())!=**null**){  bw.write(str);  bw.newLine();  }    bw.close();  br.close();  }  } |

### 13.3.7 操作Java各种数据类型的数据

DataOutputStream和DataInputStream专门用于写出和写入Java基本数据类型的数据；由于基本数据类型也不属于文本文件的范畴，因此只能用字节流；

是个装饰流；

如果要存储如下一组数据，到game.dat文件中，并在后面可以重写读取。

|  |
| --- |
| String name = “巫师”;  int age = 300;  char gender = ‘男’;  int energy = 5000;  double price = 75.5;  boolean relive = true; |

* 可以使用DataOutputStream进行写，随后用DataInputStream进行读取，而且**顺序要一致。**

示例代码：

|  |
| --- |
| **package** com.atguigu.part2.io;  **import** java.io.DataInputStream;  **import** java.io.DataOutputStream;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **public** **class** TestData {  **public** **void** save() **throws** IOException{  String name = "巫师";  **int** age = 300;  **char** gender = '男';  **int** energy = 5000;  **double** price = 75.5;  **boolean** relive = **true**;    DataOutputStream dos = **new** DataOutputStream(**new** FileOutputStream("game.dat"));  dos.writeUTF(name);  dos.writeInt(age);  dos.writeChar(gender);  dos.writeInt(energy);  dos.writeDouble(price);  dos.writeBoolean(relive);  dos.close();  }  **public** **void** reload()**throws** IOException{  DataInputStream dis = **new** DataInputStream(**new** FileInputStream("game.dat"));  String name = dis.readUTF();  **int** age = dis.readInt();  **char** gender = dis.readChar();  **int** energy = dis.readInt();  **double** price = dis.readDouble();  **boolean** relive = dis.readBoolean();    System.***out***.println(name+"," + age + "," + gender + "," + energy + "," + price + "," + relive);    dis.close();  }  } |

### 13.3.8 保存对象

如果你正在编写游戏，就得有存储和恢复游戏的功能。如果程序要存储状态，你可以选择上面的方式（DataOutputStream），对每一个对象逐个地把每一项变量的值写到特定格式的文件中。但是其实，Java还提供了面向对象的处理方式——直接存储对象。不过，这种方式保存的数据，必须是由Java程序来读取。如果在程序所储存的文件数据需要给某些非Java应用程序所读取时，就不能选用这种方式了。

#### 序列化与反序列化

* 序列化：对象转换成字节数据输出，对象的输出过程叫做序列化（对象转换成字节序列 ）
* 序列化的过程就是将对象进行编码，编码的内容包括：类名、类签名、对象的字段和数据值，以及对象中引用的其他所有对象的闭包。
* 想要输出对象，必须借助ObjectOutputStream，它有一个writeObject(obj)方法可以输出对象。
* 如果想要反序列化，必须借助ObjectInputStream，它有一个readObject()方法可以读取对象。
* 不是所有对象都可以序列化，必须实现java.io.Serializable接口的类或其子类的对象；
* 如果该对象的属性也是引用数据类型，并且该属性也要序列化，那么该属性的类型或其父类也要实现java.io.Serializable接口，否则会报java.io.NotSerializableException异常。

|  |
| --- |
| **package** com.atguigu.part2.io;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.ObjectOutputStream;  **import** com.atguigu.part2.io.bean.Account;  **public** **class** TestObjectOutputStream {  **public** **static** **void** main(String[] args) {  Account.*setInterestRate*(0.0024);  Account account = **new** Account("111000111", "尚硅谷", "123456", 1000.0, 2.4);    ObjectOutputStream oos = **null**;  **try** {  oos = **new** ObjectOutputStream(**new** FileOutputStream("account.dat"));  oos.writeObject(account);  } **catch** (IOException e) {  e.printStackTrace();  } **finally**{  **try** {  oos.close();  } **catch** (IOException e) {  }  }  }  } |
| **package** com.atguigu.part2.io.bean;  **import** java.io.Serializable;  **public** **class** Account **implements** Serializable{  **private** **static** **double** *interestRate*;  **private** String number;  **private** String name;  **private** String password;  **private** **double** balance;  **private** **double** interest;  **public** Account(String number, String name, String password, **double** balance, **double** interest) {  **super**();  **this**.number = number;  **this**.name = name;  **this**.password = password;  **this**.balance = balance;  **this**.interest = interest;  }  **public** **static** **double** getInterestRate() {  **return** *interestRate*;  }  **public** **static** **void** setInterestRate(**double** interestRate) {  Account.*interestRate* = interestRate;  }  **public** String getNumber() {  **return** number;  }  **public** **void** setNumber(String number) {  **this**.number = number;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  **public** **double** getBalance() {  **return** balance;  }  **public** **void** setBalance(**double** balance) {  **this**.balance = balance;  }  **public** **double** getInterest() {  **return** interest;  }  **public** **void** setInterest(**double** interest) {  **this**.interest = interest;  }  @Override  **public** String toString() {  **return** "Account [number=" + number + ", name=" + name + ", password=" + password + ", balance=" + balance  + ", interest=" + interest + "]";  }  } |

序列化的文件是很难让人阅读的，但它比纯文本文件、或一项一项数据保存的数据更容易让程序恢复对象的状态，也比较安全，因为一般人不会知道如何动手修改数据。

#### 哪些属性不序列化

无论是transient的还是static的属性，对于对象来说还是存在的，但是不会被序列化，不会序列化意味着该属性的值不会随着对象序列化出去，但是在反序列化构建对象的时候，static属性和transient的属性仍然是有的，只不过值是按照一定规则恢复的；具体恢复方式如下：

1. static的属性不会被序列化，反序列化时恢复为当前类变量的值（即使调用了静态的set方法修改类属性，还是恢复为类变量的值）

2.标记为transient（瞬时）的实例变量不会被序列化；在反序列化时，被恢复为默认值。

**static属性虽然不会被序列化和反序列化，由于其属于类，所以在反序列化的时候仍然能得到静态属性的值。但是transient属性在对象反序列化后，值重置为默认值。**

|  |
| --- |
|  |
| **即便在写之前更改静态属性值，也无法改变、** |

#### 3、序列化版本ID

将对象序列化之后，如果想反序列化回来构建对象，那么必须要有该类才能还原和使用该对象。但如果在序列化对象后，修改了类会发生什么事？

* 严重违反Java的类型安全性和兼容性的修改：
  + 删除了某个实例变量
  + 改变实例变量的类型
  + 将类从可序列化修改为不可序列化
  + 将实例变量修改静态的等等。
* 不会影响兼容性的修改：
  + 加入新的实例变量（还原时按默认值处理）
  + 将实例变量从瞬时修改为非瞬时的（还原时按默认值处理）

现在所有对类的修改都会导致在反序列化时失败都会抛出java.io.InvalidClassException异常；

原因为流中类的序列化ID和本地类的序列化ID不同；

如何实现对类做兼容性修改的时候，反序列化能够成功并不抛出异常？

1. 将本地类的序列化ID和流中的同步

（2）在实现在实现java.io.Serializable接口时，增加一个long类型的静态常量serialVersionUID。

如果类没有显示定义这个静态变量，它的值是Java运行时环境根据类的内部细节自动生成的，若类的源代码作了修改，serialVersionUID 就会发生变化，从而导致“旧”数据反序列化失败。

自定义异常的时候，建议：

1. 继承Throwable或者其子类，一般为Exception或者RuntimeException
2. 保留两个构造器，无参和给message赋值的构造器
3. 增加序列化ID ***serialVersionUID***

|  |
| --- |
| **package** com.atguigu.part2.io.bean;  **import** java.io.Serializable;  **public** **class** Account **implements** Serializable{  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** **static** **double** *interestRate*;  **private** String number;  **private** String name;  **private** String password;  **private** **double** balance;  **transient** **private** **double** interest;  //......此处省略构造器、get/set等其他成员方法  } |

### java.io.Externalizable接口

1.如果实现Serializer接口，对象如何序列化？各个属性序列化的顺序是什么？

都是默认的，程序员无法指定也不用关心

2.java.io.Externalizable接口

此接口有两个方法，分别为序列化写出和反序列化写入

可以控制序列化写出哪些属性，反序列化写入哪些属性，无论是transient的还是static的

|  |
| --- |
| package \_13\_IO;  import java.io.Externalizable; import java.io.IOException; import java.io.ObjectInput; import java.io.ObjectOutput;  public class phone implements Externalizable {  private static final long *serialVersionUID* = 1L;   private String brand;  private static String *name* = "苹果"; *//static属性* private transient int age; *//暂时的* public phone(){}  public phone(String brand, int age) {  this.brand = brand;  this.age = age;  }   @Override  public String toString() {  return "phone{" +  "brand='" + brand + '\'' +  ", age=" + age +  '}';  }   @Override  public void writeExternal(ObjectOutput out) throws IOException {  *//out是对象输出流* out.writeUTF(brand);  out.writeUTF(*name*);  out.writeInt(age);  }   @Override  public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {  brand = in.readUTF();  *name* = in.readUTF();  age = in.readInt();  } }  *//todo day23\_02 序列化接口Externalizable* @Test public void test15() throws IOException{  phone p = new phone("苹果",1);  FileOutputStream fos = new FileOutputStream("d://Externalizable.txt");  ObjectOutputStream oos = new ObjectOutputStream(fos);  oos.writeObject(p);   oos.close();  fos.close(); } @Test public void test16() throws IOException, ClassNotFoundException {   FileInputStream fis = new FileInputStream("d://Externalizable.txt");  ObjectInputStream ois = new ObjectInputStream(fis);  Object o = ois.readObject();  System.*out*.println(o);   ois.close();  fis.close(); } |

### 13.3.9 打印流

* 打印流是输出流，有两个打印流：

1. PrintStream
2. PrintWriter

* 功能：

1.将Java基本数据类型的数据格式转化为字符串输出;

2.引用类型的数据自动调用toString()。

这两个类提供了一系列重载的print()和println()方法，用于多种数据类型的输出，用于不会抛出 I/O 异常。

* System.out.print()解析：

|  |
| --- |
| **1.out是什么？**  **System类中的out是一个static属性，为PrintStream类型**  **public final static PrintStream *out* = null;** |
| **2.print()方法打印基本属性**    对于基本数据类型的打印，其实都是调用了String的valueOf方法，所以打印出来都是字符串  public void print(long l) {  write(String.*valueOf*(l)); } |
| 3.print()打印对象  public void println(Object x) {  String s = String.*valueOf*(x);  synchronized (this) {  print(s);  newLine();  } }  public static String valueOf(Object obj) {  return (obj == null) ? "null" : obj.toString(); }  所以打印对象调用的都是toString()， null则为 "null" |

* PrintWriter

在web阶段的时候，从服务器往客户端返回消息的时候用到response,response.getWriter就是获取PrintWriter对象

Web服务器往客户端返回html网页时，用的就是PrintWriter对象

PrintStream 打印的所有字符都使用平台的默认字符编码转换为字节。在需要写入字符而不是写入字节的情况下，应该使用 PrintWriter 类，PrintWriter类还可以自定义字符编码。

与 PrintStream 类不同，如果启用了自动刷新，则只有在调用 println、printf 或 format 的其中一个方法时才可能完成此操作，而不是每当正好输出换行符时才完成。这些方法使用平台自有的行分隔符概念，而不是换行符。

案例：

从键盘输入消息，按行写入文件

|  |
| --- |
| **package** com.atguigu.part2.io;  **import** java.io.IOException;  **import** java.io.PrintWriter;  **import** java.util.Scanner;  **public** **class** TestPrintWriter {  **public** **static** **void** main(String[] args) {  Scanner input = **new** Scanner(System.***in***);  PrintWriter pw = **null**;    **try** {  pw = **new** PrintWriter("message.txt");    **while**(**true**){  System.***out***.println("请输入留言：");  String message = input.nextLine();    **if**("stop".equals(message)){  **break**;  }  pw.println(message);  System.***out***.println("结束留言，请输入stop");  }  } **catch** (IOException e) {  e.printStackTrace();  } **finally**{  input.close();  pw.close();  }  }  } |

### 13.3.10 拷贝或剪切整个目录

案例：

将”atguigu/download”文件夹，复制到”atguigu/temp”目录中去

|  |
| --- |
| **package** com.atguigu.part3.io;  **import** java.io.File;  **import** java.io.IOException;  **public** **class** TestFileUtils {  **public** **static** **void** main(String[] args) **throws** IOException {  FileUtils.*copyDirectory*(**new** File("atguigu/download"), **new** File("atguigu/temp"));  }  } |
| **package** com.atguigu.part3.io;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **public** **class** FileUtils {  //复制文件  **public** **static** **void** copyFile(File src, File dest) **throws** IOException{  **if**(!src.isFile()){  **throw** **new** RuntimeException(src.getPath() + "不存在");  }  FileInputStream fis = **new** FileInputStream(src);  FileOutputStream fos = **new** FileOutputStream(dest);    **byte**[] data = **new** **byte**[1024];  **int** len;  **while**((len = fis.read(data))!=-1){  fos.write(data, 0, len);  }    fos.close();  fis.close();  }  //复制文件夹  **public** **static** **void** copyDirectory(File srcDir,File destDir)**throws** IOException{  //父目录不能拷贝到子目录中  **if**(destDir.getAbsolutePath().contains(srcDir.getAbsolutePath())){  **throw** **new** IOException("父目录不能拷贝到子目录中");  }  **if**(!destDir.isDirectory()){  **throw** **new** IOException("目标对象不是目录");  }  File dest = **new** File(destDir,srcDir.getName());  //如果srcDir是个文件，那么直接复制文件到destDir中  **if**(srcDir.isFile()){  *copyFile*(srcDir, dest);  }**else**{  dest.mkdirs();//先在destPath目录中创建srcDir对应的文件夹  File[] list = srcDir.listFiles();//准备复制srcDir的子目录或子文件  **for**(File f: list){  *copyDirectory*(f,dest);  }  }  }  } |

思考：那么剪切呢？

|  |
| --- |
| **package** com.atguigu.part3.io;  **import** java.io.File;  **import** java.io.IOException;  **public** **class** TestFileUtils {  **public** **static** **void** main(String[] args) **throws** IOException {  FileUtils.cutDirectory(**new** File("atguigu/download"), **new** File("atguigu/temp"));  }  } |
| //剪切文件夹  **public** **static** **void** cutDirectory(File srcDir,File destDir)**throws** IOException{  **if**(destDir.getAbsolutePath().contains(srcDir.getAbsolutePath())){  **throw** **new** IOException("父目录不能剪切到子目录中");  }  **if**(!destDir.isDirectory()){  **throw** **new** IOException("目标对象不是目录");  }  File dest = **new** File(destDir,srcDir.getName());  **if**(srcDir.isFile()){  *copyFile*(srcDir, dest);  }**else**{  dest.mkdirs();  File[] list = srcDir.listFiles();  **for**(File f: list){  *cutDirectory*(f,dest);  }  }  srcDir.delete();  } |

## 13.4 System中的IO流

System.in和System.out分别代表了系统标准的输入和输出设备

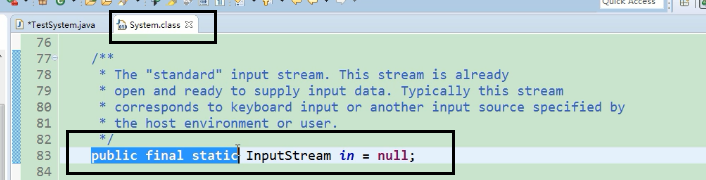
* 默认输入设备是键盘，输出设备是显示器
* System.in的类型是InputStream
* System.out的类型是PrintStream
* System.err的类型是PrintStream，按照惯例，此输出流用于显示错误消息，或者显示那些即使用户输出流（变量 out 的值）已经重定向到通常不被连续监视的某一文件或其他目标，也应该立刻引起用户注意的其他信息。

重定向：

* public static void setIn(InputStream in)
* public static void setOut(PrintStream out)
* public static void setErr(PrintStream err)

控制台是个文件，查看System类中的initializeSystemClass()方法

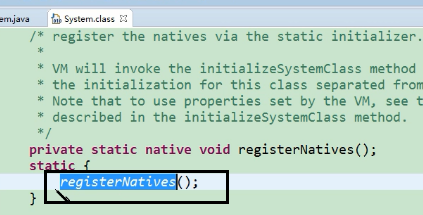
* FileInputStream fdIn = new FileInputStream(FileDescriptor.in);
* FileOutputStream fdOut = new FileOutputStream(FileDescriptor.out);
* FileOutputStream fdErr = new FileOutputStream(FileDescriptor.err);



System类中的IO流赋值为null，如何能调用流的方法？

|  |
| --- |
| 所以其实不是null |

System类中有一个静态代码块，在System类初始化时执行，此方法是一个native方法，用于注册初始化这些常量



**而且这些静态常量其实不是常量[或者说是在java层面是常量]**

|  |
| --- |
| 通过System类的setOut方法，设定out指向新的PrintStream输出流。  此测试代码 将 aaa-ddd输入到1.txt文件中，而非打印到控制台上。  **既然out在Java中显示为常量，为什么能改？**      **因为setOut方法调用的是native方法，native方法不是java语言写的。因此可以看似是用java代码改变了常量，其实使用非java方法来修改。** |

### 13.3.10.1 重定向System.in和System.out

|  |
| --- |
| **package** com.atguigu.part4.io;  **import** java.io.FileDescriptor;  **import** java.io.FileInputStream;  **import** java.util.Scanner;  **public** **class** TestSystemIn {  **public** **static** **void** main(String[] args) **throws** Exception{  //重定向从文件输入  System.*setIn*(**new** FileInputStream("myjava\\info.txt"));  Scanner input = **new** Scanner(System.***in***);  **while**(input.hasNext()){  String str = input.nextLine();  System.***out***.println(str);  }  input.close();  //重定向回键盘输入  System.*setIn*(**new** FileInputStream(FileDescriptor.***in***));  }  } |
| **package** com.atguigu.part4.io;  **import** java.io.FileDescriptor;  **import** java.io.FileOutputStream;  **import** java.io.PrintStream;  **public** **class** TestSystemOut {  **public** **static** **void** main(String[] args) **throws** Exception{  System.***out***.println("hello");  //重定向输出到文件  System.*setOut*(**new** PrintStream("myjava\\print.txt"));  System.***out***.println("world");  //重定向回控制台  System.*setOut*(**new** PrintStream(**new** FileOutputStream(FileDescriptor.***out***)));  System.***out***.println("java");  }  } |

### 13.3.10.2 为Scanner指定输入流

java.util.Scanner是一个可以使用正则表达式来解析基本类型和字符串的简单文本扫描器。

示例代码：从键盘输入数据

|  |
| --- |
| **package** com.atguigu.part4.io;  **import** java.util.Scanner;  **public** **class** TestScannerSystemIn {  **public** **static** **void** main(String[] args) {  Scanner input = **new** Scanner(System.***in***);    System.***out***.print("姓名：");  String name = input.nextLine();    System.***out***.print("性别：");  **char** gender = input.next().charAt(0);    System.***out***.print("年龄：");  **int** age = input.nextInt();    System.***out***.print("电话：");  String phone = input.next();    System.***out***.print("邮箱：");  String email = input.next();    System.***out***.println("姓名：" + name);  System.***out***.println("性别：" + gender);  System.***out***.println("年龄：" + age);  System.***out***.println("电话：" + phone);  System.***out***.println("邮箱：" + email);    input.close();  }  } |

示例代码：从文件扫描数据

|  |
| --- |
| **package** com.atguigu.part4.io;  **import** java.io.FileInputStream;  **import** java.util.Scanner;  **public** **class** TestScanner {  **public** **static** **void** main(String[] args) **throws** Exception{  Scanner input = **new** Scanner(**new** FileInputStream("myjava\\info.txt"));  **while**(input.hasNext()){  String str = input.nextLine();  System.***out***.println(str);  }  input.close();  }  } |

## 13.5 JDK1.7新特性：try...with...resources

(1)原始的方式，将异常抛出的方式：

|  |
| --- |
| package \_13\_IO;  import java.io.\*;  public class \_05\_try\_catch {  public static void main(String[] args) throws IOException {  *//从d:/1.txt(GBK)文件中，读取内容，写到项目根目录下1.txt(UTF-8)文件中* FileInputStream fis = new FileInputStream("d:/1.txt");  InputStreamReader isr = new InputStreamReader(fis,"GBK");  BufferedReader br = new BufferedReader(isr);   FileOutputStream fos = new FileOutputStream("1.txt");  OutputStreamWriter osw = new OutputStreamWriter(fos,"UTF-8");  BufferedWriter bw = new BufferedWriter(osw);   String str;  while((str = br.readLine()) != null){  bw.write(str);  bw.newLine();  }   bw.close();  osw.close();  fos.close();   br.close();  isr.close();  fis.close();   } } |

(2)用try-catch

将IO流对象的声明和初始化都写在try-catch外面，并且初始化都为null，因为不初始化只声明，在finally中要关闭这些流，要调用对象的close方法，而声明只是在方法栈中开辟一个对象引用，这样无法通过编译。

|  |
| --- |
| public void test01(){  *//从d:/1.txt(GBK)文件中，读取内容，写到项目根目录下1.txt(UTF-8)文件中*  ***//第一步：声明并初始化要用到的资源为NUll 局部变量必须初始化***  FileInputStream fis = null;  InputStreamReader isr = null;  BufferedReader br = null;  FileOutputStream fos = null;  OutputStreamWriter osw = null;  BufferedWriter bw = null;  *//第二步：将IO流对象的创建放在try块中* try {  fis = new FileInputStream("d:/1.txt");  isr = new InputStreamReader(fis,"GBK");  br = new BufferedReader(isr);   fos = new FileOutputStream("1.txt");  osw = new OutputStreamWriter(fos,"UTF-8");  bw = new BufferedWriter(osw);   String str;  while((str = br.readLine()) != null){  bw.write(str);  bw.newLine();  }  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (UnsupportedEncodingException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }finally{  ***//第三步：将资源的关闭写在finally中 由于close()也会抛出异常，因此还要进行try-catch***try {  if(bw!=null){  bw.close();  }  } catch (IOException e) {  e.printStackTrace();  }  try {  if(osw!=null){  osw.close();  }  } catch (IOException e) {  e.printStackTrace();  }  try {  if(fos!=null){  fos.close();  }  } catch (IOException e) {  e.printStackTrace();  }   try {  if(br!=null){  br.close();  }  } catch (IOException e) {  e.printStackTrace();  }  try {  if(isr!=null){  isr.close();  }  } catch (IOException e) {  e.printStackTrace();  }  try {  if(fis!=null){  fis.close();  }  } catch (IOException e) {  e.printStackTrace();  }  }  } |

(3) 减少流的关闭 关闭最外层会帮我们关闭内层流

|  |
| --- |
| public void test02() {  BufferedReader br = null;  BufferedWriter bw = null;  try {  br = new BufferedReader(new InputStreamReader(new FileInputStream("d:/1.txt"),"GBK"));  bw = new BufferedWriter(new OutputStreamWriter(new FileOutputStream("1.txt"),"UTF-8"));   String str;  while((str = br.readLine()) != null){  bw.write(str);  bw.newLine();  }  } catch (UnsupportedEncodingException e) {  e.printStackTrace();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }finally{  try {  if(bw!=null){  bw.close();  }  } catch (IOException e) {  e.printStackTrace();  }  try {  if(br!=null){  br.close();  }  } catch (IOException e) {  e.printStackTrace();  }  }  } |

(4)

|  |
| --- |
| package com.atguigu.test05;  import java.io.BufferedReader;  import java.io.BufferedWriter;  import java.io.FileInputStream;  import java.io.FileNotFoundException;  import java.io.FileOutputStream;  import java.io.IOException;  import java.io.InputStreamReader;  import java.io.OutputStreamWriter;  import java.io.UnsupportedEncodingException;  import org.junit.Test;  */\*  \* JDK1.7中新增了一种try...catch处理的方式，  \* 此语法名称称为try...with....resource，但实际上代码中是没有with和resource，****它是为资源关闭专门设计的try...catch的语法***  *\*  \*写法如下：  \* try(  \* 需要关闭的资源对象的创建 写在try的小括号里面  \* ){  \* 可能发生异常的逻辑代码  \* }catch(异常类型 e){  \* 异常处理代码  \* }catch(异常类型 e){  \* 异常处理代码  \* }  \*  \** ***凡是在try()中声明的资源对象，都会自动关闭，无论是否发生异常。***  *\*/* public class TestTryWithResource {  @Test  public void test03() {  *//从d:/1.txt(GBK)文件中，读取内容，写到项目根目录下1.txt(UTF-8)文件中* try (  FileInputStream fis = new FileInputStream("d:/1.txt");  InputStreamReader isr = new InputStreamReader(fis, "GBK");  BufferedReader br = new BufferedReader(isr);   FileOutputStream fos = new FileOutputStream("1.txt");  OutputStreamWriter osw = new OutputStreamWriter(fos, "UTF-8");  BufferedWriter bw = new BufferedWriter(osw);  ) {  String str;  while ((str = br.readLine()) != null) {  bw.write(str);  bw.newLine();  }  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  } |

不管{}部分是否发生异常，都会关闭资源对象

|  |
| --- |
| **package** com.atguigu.part5;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.FileNotFoundException;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **public** **class** TestTryWithResource {  **public** **static** **void** copy(File src, File dest) **throws** IOException{  **try**(  InputStream input = **new** FileInputStream(src);  OutputStream output = **new** FileOutputStream(dest);  ){  **byte**[] data = **new** **byte**[1024];  **int** len;  **while** ((len = input.read(data)) != -1) {  output.write(data, 0, len);  }  }  }    **public** **static** **void** copy2(File src, File dest) {  **try**(  InputStream input = **new** FileInputStream(src);  OutputStream output = **new** FileOutputStream(dest);  ){  **byte**[] data = **new** **byte**[1024];  **int** len;  **while** ((len = input.read(data)) != -1) {  output.write(data, 0, len);  }  }**catch**(FileNotFoundException e){  e.printStackTrace();  }**catch**(IOException e){  e.printStackTrace();  }  }  } |

## 13.6 本章小结

通过本章的学习，你的技能得到更大的提升了，你可以做到把数据持久化到文件中，而不是仅仅存在与内存中了，也为在网络中传输数据奠定了良好的基础。

|  |
| --- |
| 问：那JDK1.4引入的NIO相关类呢？  答：java.nio这批API带来了重大的效能提升并可以充分利用执行程序的机器上的原始容量。NIO的一项关键能力是你可以直接控制buffer。另一项能力是non-blocking的输入和输出，它能让你的输入/输出程序代码在没有东西可以读取或写入时不必等在那里。某些现有的类（包括FileInputStream和FileOutputStream）会利用到其中的一部分功能。NIO使用起来更为复杂，除非你真的很需要新功能，不然的话，使用本章介绍的功能方法会简单的多。此外，如果没有很好的设计，NIO可能会引发效能损失。非NIO的输入/输出适合九成以上的应用，特别在你还是新手的时候更是如此。 |