**3、具体内容**

Java SE的四大核心知识点：面向对象、类集框架、Java IO、JDBC。

Java IO的核心就一句话解释：**如果抽象类或接口之中的抽象方法被子类所覆写了，那么实例化这个子类的时候，所调用的方法一定是被覆写过的方法**。

所有的IO操作都在java.io包之中进行定义，而且整个java.io包实际上就是五个类和一个接口：

· 五个类：File、InputStream、OutputStream、Reader、Wirter；

· 一个接口：Serializable。

**3.1、文件操作类：File（重点）**

在整个java.io包之中，File类是唯一的一个与文件本身操作有关的类，所谓的文件本身指的是：文件的创建、删除、重命名、取得文件大小、修改日期。

如果要想使用File类操作文件的话，那么肯定要通过构造方法实例化File类对象，而实例化File类对象的过程之中主要使用以下两种构造方法：

· **在Java EE的开发之中：**public File(String pathname)；

· **在Android开发之中：**public File(File parent, String child)。

**范例：**文件的基本操作，主要有两种功能：

· **创建文件：**public boolean createNewFile() throws IOException；

· **删除文件：**public boolean delete()；

· **判断路径是否存在：**public boolean exists()；

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:\\demo.txt"); // 文件的路径  **if** (file.exists()) { // 文件存在  file.delete(); // 删除文件  } **else** { // 文件不存在  file.createNewFile(); // 创建新文件  }  }  } |

本程序操作就表示文件如果存在则删除，如果不存在，则创建一个新的文件，此时基本功能是实现了，不过这个程序此时却存在了三个问题：

**问题一：**关于路径分隔符

在windows操作系统之中，使用“\”作为路径分隔符，而在linux系统下使用“/”作为路径的分隔符，而从实际的开发而言，大部分情况下都会在windows中做开发，而后将项目部署到linux下，那么此时，路径的分隔符都需要进行修改，这样实在是过于麻烦，为此在File类之中提供了一个常量：public static final String **separator**（按照Java的命名规范来讲，对于全局常量应该使用大写字母的方式定义，而此处使用的是小写，是由Java的发展历史所带来的问题）。

|  |
| --- |
| File file = **new** File("D:" + File.*separator* + "demo.txt"); // 文件的路径 |

**问题二：**是有可能（本人没出现过，学生出现过）会出现的问题

发现当程序执行完成之后，对于文件的创建或者是删除是会存在一些操作上的延迟，如果现在假设先删除了一个文件，而后立刻判断此文件是否存在，那么可能得到的结果就是错误的（为true），因为所有的\*.class文件都要通过JVM与操作系统间接操作，这样就有可能会出现延迟的问题。

**问题三：**之前进行文件创建的时候都是在根路径下创建完成的，如果说现在要创建的文件有目录呢？例如，现在要创建一个d:\hellodemo\my\test\demo.txt文件，而此时在执行程序的时候hellodemo目录不存在，这个时候执行的话就会出现错误提示：

|  |
| --- |
| Exception in thread "main" java.io.IOException: 系统找不到指定的路径。 |

因为现在目录不存在，所以不能创建，那么这个时候必须要首先判断要创建文件的父路径是否存在，如果不存在应该创建一个目录，之后再进行文件的创建，而要想完成这样的操作，需要以下几个方法的支持：

· **找到一个指定文件的父路径：**public File getParentFile()；

· **创建目录：**public boolean mkdirs()。

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "hellodemo"  + File.*separator* + "my" + File.*separator* + "test"  + File.*separator* + "demo.txt"); // 文件的路径  **if** (!file.getParentFile().exists()) { // 父路径不存在  file.getParentFile().mkdirs(); // 创建目录  }  **if** (file.exists()) { // 文件存在  file.delete(); // 删除文件  } **else** { // 文件不存在  file.createNewFile(); // 创建新文件  }  }  } |

以后在任何的java.io.File类开发的过程之中，都一定要考虑文件目录的问题。

除了以上的常用的方法之外，在File类之中还可以通过以下的方法取得一些文件的基本信息：

· 取得文件的名称|：public String getName()；

· 给定的路径是否是文件夹：public boolean isDirectory()；

· 给定的路径是否是文件：public boolean isFile()；

· 是否是隐藏文件：public boolean isHidden()；

· 文件的最后一次修改日期：public **long** lastModified()；

· 取得文件大小：public long length()，是以字节为单位返回的。

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.math.BigDecimal;  **import** java.text.SimpleDateFormat;  **import** java.util.Date;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "eclipse.zip"); // 文件的路径  **if** (file.exists()) { // 文件存在  System.*out*.println("文件名称：" + file.getName());  System.*out*.println(file.getName()  + (file.isDirectory() ? "是一个目录。" : "不是一个目录。"));  System.*out*.println(file.getName()  + (file.isFile() ? "是一个文件。" : "不是一个文件。"));  System.*out*.println(file.getName()  + (file.isHidden() ? "是一个隐藏文件。" : "不是一个隐藏文件。"));  System.*out*.println("最后一次更改日期："  + **new** SimpleDateFormat("yyyy年MM月dd日 HH时mm分ss秒")  .format(**new** Date(file.lastModified())));  System.*out*.println("文件大小："  + **new** BigDecimal(file.length() / (**double**) 1024 / 1024)  .divide(**new** BigDecimal(1), 2,  BigDecimal.*ROUND\_HALF\_UP*).doubleValue()  + "M。");  }  }  } |

**范例：**为一个文件重命名，使用File类可以为文件执行重命名的一个操作，操作方法：public boolean renameTo(File dest)，在这个方法上需要传入一个File类对象，而这个对象就表示新的名字。

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "eclipse.zip"); // 文件的路径  **if** (file.exists()) { // 文件存在  File newFile = **new** File("D:" + File.*separator* + "世界末日的eclipse.zip");  file.renameTo(newFile);  }  }  } |

在File类之中有一个非常重要的方法，可以列出一个指定目录下的全部文件信息：

· **列出目录内容：**public File[] listFiles()，此方法将目录中的所有文件以File对象数组的方式返回；

**范例：**列出指定的目录内容

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "javademo"); // 文件的路径  **if** (file.exists()) { // 文件目录存在  File result[] = file.listFiles(); // 列出目录中的全部内容  **for** (**int** x = 0; x < result.length; x++) {  System.*out*.println(result[x]);  }  }  }} |

那么下面继续做一个稍微复杂点的应用，现在希望可以将一个目录之中的全部文件都列出来，那么这种情况下只能采用递归：因为列出一个目录下的全部文件或文件夹之后，如果发现列出的内容是文件夹，则应该向后继续列出。

**〖了解〗范例：**列出指定目录下的全部内容

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator*); // 文件的路径  *print*(file);  }  **public** **static** **void** print(File file) {  **if** (file.isDirectory()) { // 给的路径是文件夹  File result[] = file.listFiles(); // 列出此目录中的全部内容  **if** (result != **null**) {  **for** (**int** x = 0; x < result.length; x++) {  *print*(result[x]);  }  }  }  System.*out*.println(file);  }} |

当然，如果此时可以将输出操作变为删除操作（file.delete();），就表示彻底清空。

## 3.2、字节流和字符流（核心）

使用File类执行的所有操作都是针对于文件本身，但是却没有针对于文件的内容，而要进行文件内容操作就需要通过Java之中提供的两组类完成：

· **字节操作流（是在JDK 1.0的时候定义的）：**OutputStream、InputStream；

· **字符操作流（是在JDK 1.1的时候定义的）：**Writer、Reader。

但是不管是字节流还是字符流的操作，本身都表示资源操作，而执行所有的资源操作都会按照如下的几个步骤进行，下面以文件操作为例（对文件进行读、写操作）：

· 如果要操作的是文件，那么首先要通过File类对象找到一个要操作的文件路径（路径有可能存在，有可能不存在，如果不存在，则要创建路径）；

· 通过字节流或字符流的子类为字节流或字符流的对象实例化（向上转型）；

· 执行读 / 写操作；

· 最后一定要关闭操作的资源（close()），不管日后如何操作，资源永远要关闭。

### 3.2.1、字节输出流：OutputStream

java.io.OutputStream主要的功能是进行字节数据的输出的，而这个类的定义如下：

|  |
| --- |
| public abstract class OutputStream  extends Object  implements Closeable, Flushable |

发现OutputStream类定义的时候实现了两个接口：Closeable、Flushable，那么这两个接口的定义如下：

|  |  |
| --- | --- |
| **Closeable：JDK 1.5推出** | **Flushable：JDK 1.5推出** |
| **public** **interface** Closeable **extends** AutoCloseable {  **public** **void** close() **throws** IOException;  } | **public** **interface** Flushable {  **public** **void** flush() **throws** IOException;  } |

**提示：**对于Closeable继承的AutoCloseable接口

AutoCloseable是在JDK 1.7的时候又增加了一个新的接口，但是这个接口的定义和Closeable定义是完全一样的，我个人认为：有可能在一些其他的类上出现了自动的关闭功能，Closeable是手工关闭，AutoCloseable属于自动关闭。

但是对于Closeable和Flushable这两个接口实话而言用户不需要关注，因为从最早的习惯对于flush()和close()两个方法都是直接在OutputStream类之中定义的，所以很少去关心这些父接口问题。

对于OutputStream类而言发现其本身定义的是一个抽象类（abstract class），按照抽象类的使用原则来讲，需要定义抽象类的子类，而现在如果要执行的是文件操作，则可以使用FileOutputStream子类完成，如果按照面向对象的开发原则，子类要为抽象类进行对象的实例化，而后调用的方法以父类中定义的方法为主，而具体的实现找实例化这个父类的子类完成，也就是说在整个的操作之中，用户最关心的只有子类的构造方法：

· 实例化FileOutputStream（新建数据）：public FileOutputStream(File file) throws FileNotFoundException；

· 实例化FileOutputStream（追加数据）：public FileOutputStream(File file, boolean append)

throws FileNotFoundException

当取得了OutputStream类的实例化对象之后，下面肯定要进行输出操作，在OutputStream类之中定义了三个方法：

· **输出单个字节数据：**public abstract void write(**int** b) throws IOException；

· **输出一组字节数据：**public void write(**byte[] b**) throws IOException；

· **输出部分字节数据：**public void write(byte[] b, int off, int len) throws IOException；

**范例：**使用OutputStream向文件之中输出数据，输出路径：d:\hellodemo\test.txt

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileOutputStream;  **import** java.io.OutputStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "hellodemo"  + File.*separator* + "test.txt"); // 第1步：定义文件路径  **if** (!file.getParentFile().exists()) { // 父路径不存在  file.getParentFile().mkdirs(); // 创建父路径  }  OutputStream output = **new** FileOutputStream(file); // 第2步：通过子类实例化父类  String data = "Hello World .";// 要输出的数据  output.write(data.getBytes()); // 第3步：输出数据，要将数据变为字节数组输出  output.close(); // 第4步：关闭资源  }  } |

在整个的文件输出过程之中可以发现，如果现在要输出的文件不存在，那么会出现自动创建文件的情况，并且如果重复执行以上的代码，会出现新的内容覆盖掉旧内容的操作，所以下面可以使用FileOutputStream类的另外一个构造方法进行数据的追加：

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileOutputStream;  **import** java.io.OutputStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "hellodemo"  + File.*separator* + "test.txt"); // 第1步：定义文件路径  **if** (!file.getParentFile().exists()) { // 父路径不存在  file.getParentFile().mkdirs(); // 创建父路径  }  OutputStream output = **new** FileOutputStream(file, **true**);// 第2步：通过子类实例化父类  String data = "Hello World .\r\n";// 要输出的数据  output.write(data.getBytes()); // 第3步：输出数据，要将数据变为字节数组输出  output.close(); // 第4步：关闭资源  }  } |

如果说现在不想全部内容输出，也可以使用另外一个write()方法部分内容输出。

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileOutputStream;  **import** java.io.OutputStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "hellodemo"  + File.*separator* + "test.txt"); // 第1步：定义文件路径  **if** (!file.getParentFile().exists()) { // 父路径不存在  file.getParentFile().mkdirs(); // 创建父路径  }  OutputStream output = **new** FileOutputStream(file); // 第2步：通过子类实例化父类  String data = "Hello World .\r\n";// 要输出的数据  output.write(data.getBytes(), 0, 5); // 第3步：输出数据，要将数据变为字节数组输出  output.close(); // 第4步：关闭资源  }  } |

在OutputStream类之中所有的数据都是以字节数据为主的。

### 3.2.2、字节输入流：InputStream

如果现在要从指定的数据源之中读取数据，使用InputStream，而这个类的定义如下：

|  |
| --- |
| public **abstract** class InputStream  extends Object  implements Closeable |

在InputStream类之中，定义了三个读取数据的操作方法：

· **读取单个字节：**public abstract int read() throws IOException；

|- **说明：**每次执行read()方法都会读取一个数据源的指定数据，如果现在发现已经读取到了结尾返回-1；

· **读取多个字节：**public int read(byte[] b) throws IOException；

|- 说明：如果现在要读取的数据小于byte的数据，这个时候read()方法的返回值int返回的是数据个数，如果现在开辟的字节数组小于读取的长度，如果数据已经读完了，则这个时候的int返回的是-1；

· **读取指定多个字节：**public int read(byte[] b, int off, int len) throws IOException。

既然InputStream为抽象类，那么这个抽象类要使用就必须有子类，现在是通过文件读取内容，肯定使用FileInputStream子类进行操作，与OutputStream类的使用一样，对于FileInputStream也只关心构造方法：

· **FileInputStream类构造方法：**public FileInputStream(File file) throws FileNotFoundException；

**范例：**一次性全部读取数据进来

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.InputStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "hellodemo"  + File.*separator* + "test.txt"); // 定义文件路径  **if** (file.exists()) { // 文件存在则可以读取  InputStream input = **new** FileInputStream(file) ;  **byte** data[] = **new** **byte**[1024]; // 假设要读的长度是1024  **int** len = input.read(data) ; // 读取数据，返回读取个数  input.close() ; // 关闭  System.*out*.println("读取的数据是：【" + **new** String(data, 0, len) + "】");  }  }  } |

在使用手机的过程之中短信功能就是这种操作，每一个短信只能发送70个字，不管是否够70都按70算，因为接收短信的时候是开辟了一个指定长度的空间进行接收。

**范例：**单个字节读取

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.InputStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "hellodemo"  + File.*separator* + "test.txt"); // 定义文件路径  **if** (file.exists()) { // 文件存在则可以读取  InputStream input = **new** FileInputStream(file);  **byte** data[] = **new** **byte**[1024]; // 假设要读的长度是1024  **int** foot = 0; // 操作data数组的脚标  **int** temp = 0;  **do** {  temp = input.read(); // 读取了一个字节  **if** (temp != -1) {  data[foot++] = (**byte**) temp; // 保存读取进来的单个字节  }  } **while** (temp != -1);// 没有读取完，还有内容  input.close(); // 关闭  System.*out*.println("读取的数据是：【" + **new** String(data, 0, foot) + "】");  }  }  } |

很明显，对于不确定个数只知道结束条件的循环，只能采用while或者是do..while，而在讲解这两种循环的时候也强调过：几乎不会去使用do…while，主要使用while循环，那么以上的代码也可以修改为while循环。

**范例：**使用while循环修改之前的程序

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.InputStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "hellodemo"  + File.*separator* + "test.txt"); // 定义文件路径  **if** (file.exists()) { // 文件存在则可以读取  InputStream input = **new** FileInputStream(file);  **byte** data[] = **new** **byte**[1024]; // 假设要读的长度是1024  **int** foot = 0; // 操作data数组的脚标  **int** temp = 0;  // 第一步：temp = input.read()，读取一个单个字节，并且将内容给temp变量  // 第二步：temp != -1，将接收到的temp的数值判断是否为-1，如果为-1则表示退出循环，如果不是，则保存数据  **while** ((temp = input.read()) != -1) {  data[foot++] = (**byte**) temp; // 保存读取进来的单个字节  }  input.close(); // 关闭  System.*out*.println("读取的数据是：【" + **new** String(data, 0, foot) + "】");  }  }  } |

在日后的所有开发之中，都会使用以上的while循环方式进行数据的读取。

### 3.2.3、字符输出流：Writer

Writer类也是一个专门用于数据输出的操作类，这个类定义：

|  |
| --- |
| public abstract class Writer  extends Object  implements Appendable, Closeable, Flushable |

在Wirter类之中定义的write()方法都是以字符数据为主，但是在这些方法之中，只关心一个：

· **输出一个字符串：**public void write(String str) throws IOException。

在Wirter类之中比OutputStream类最为方便的一点就是其可以直接使用String型数据输出，并且不再需要将其变为字节数组了。而Writer类本身也是一个抽象类，那么如果要使用依然要依靠它的子类，尤其是现在操作的是文件，使用FileWriter子类。

**范例：**使用Wirter类进行内容的输出

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileWriter;  **import** java.io.Writer;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "hellodemo"  + File.*separator* + "test.txt"); // 定义文件路径  **if** (!file.getParentFile().exists()) {  file.getParentFile().mkdirs();// 创建父目录  }  Writer out = **new** FileWriter(file);  String data = "Hello World .";  out.write(data) ; // 直接输出字符串  out.close() ; // 关闭输出  }  } |

从输出来讲，Wirter类的输出要比OutputStream类更加的方便。

### 3.2.4、字符输入流：Reader

Reader是进行字符数据读取的操作类，其定义：

|  |
| --- |
| public abstract class Reader  extends Object  implements Readable, Closeable |

在Writer类之中存在了直接输出一个字符串数据的方法，可是在Reader类之中并没有定义这样的方法，只是定义了三个按照字符读取的方法，为什么会这样？

因为在使用OutputStream输出数据的时候，其程序可以输出的大小一定是程序可以承受的数据大小，那么如果说使用InputStream读取的时候，可能被读取的数据非常的大，那么如果一次性全读进来了，就会出现问题，所以只能一个一个的进行读取。

Reader依然是抽象类，那么如果从文件读取，依然使用FileReader类。

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileReader;  **import** java.io.Reader;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "hellodemo"  + File.*separator* + "test.txt"); // 定义文件路径  **if** (file.exists()) {  Reader in = **new** FileReader(file); // 字符输入流  **char** data[] = **new** **char**[1024]; // 开辟数组  **int** len = in.read(data); // 读取数据  System.*out*.println("读取数据内容：【" + **new** String(data, 0, len) + "】");  in.close() ;  }  }} |

字符比字节的好处就是在于字符串数据的支持上，而这个好处还只是在Writer类中体现，所以与字节流相比，字符流的操作并不是对等的关系。

### 3.2.5、字节流和字符流的区别

通过以上的代码演示，现在可以发现，对于字节流和字符流可以完成类似的功能，那么在开发之中使用那一种呢？

**两者的区别：**

字节流在进行IO操作的时候，直接针对的是操作的数据终端（例如：文件），而字符流操作的时候不是直接针对于终端，而是针对于缓存区（理解为内存）操作，而后由缓存取操作终端（例如：文件），属于间接操作，按照这样的方式，如果说在使用字节流的时候不关闭最后的输出流操作，也可以将所有的内容进行输出，而字符输出流的时候如果不关闭，则意味着缓冲区之中的内容不会被输出，当然，这个时候可以由用户自己去调用flush()方法进行强制性的手工清空：

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileWriter;  **import** java.io.Writer;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "hellodemo"  + File.*separator* + "test.txt"); // 定义文件路径  **if** (!file.getParentFile().exists()) {  file.getParentFile().mkdirs();// 创建父目录  }  Writer out = **new** FileWriter(file);  String data = "Hello World .";  out.write(data) ; // 直接输出字符串  out.flush() ; // 清空缓冲区  }  } |

对于电脑磁盘或者是网络数据传输上，使用最多的数据类型都是字节数据，包括图片、音乐、各种可执行程序也都是字节数据，很明显，字节数据要比字符数据更加的广泛，但是在进行中文处理的过程之中，字符流又要比字节流方便许多，所以如果要使用的话，首先考虑的是字节流（还有一个原因是因为字节流和字符流的代码形式类似），如果有中文处理的问题，才会考虑使用字符流。

**主要的区别：**

· 字节流没有使用到缓冲区，而字符流使用了；

· 处理各种数据都可以通过字节流完成，而在处理中文的时候使用字符流会更好。

### 3.2.6、思考题（核心）

要求模拟dos系统的文件拷贝命令，由初始化参数输入源文件和拷贝文件的路径，而后执行文件拷贝操作。

**提示：**本程序暂时不考虑类的设计，所有的代码都在主方法中完成。考虑复制一个大文件的情况，例如：500M以上。

**实现思路：**

如果要想实现文件的复制那么肯定要使用的是字节流，因为文件有可能是图片，而对于这样的操作有两种实现方式； · **方式一：**将要复制的文件内容全部读取到内存之中，而后将所有内容一次输出到目标文件之中；

· **方式二：**采用边读边写的方式一点一点的进行文件的拷贝。

很明显，使用方式二会更加的方便，因为如果要复制的文件很大，那么方式一是不可能在内存之中装如此只大的数据量，所以全部读取是不太可能的。

**范例：**代码的基础实现

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **public** **class** Copy {  **public** **static** **void** main(String[] args) **throws** Exception {  **if** (args.length != 2) { // 长度不是2，命令有错误  System.*out*.println("执行程序命令错误！");  System.*exit*(1); // 程序退出  }  File inFile = **new** File(args[0]); // 要复制的源文件  **if** (!inFile.exists()) { // 源文件不存在  System.*out*.println("源文件不存在！");  System.*exit*(1); // 程序退出  }  File outFile = **new** File(args[1]); // 目标文件  InputStream input = **new** FileInputStream(inFile);  OutputStream output = **new** FileOutputStream(outFile);  **int** temp = 0;  **while** ((temp = input.read()) != -1) { // 有内容  output.write(temp); // 输出一个内容  }  input.close() ;  output.close() ;  }  } |

但是这种代码有一个问题：以上的代码适合于很小的文件，但是文件稍微大一些了，那么肯定不适合。因为这种做法是采用一个个字节复制的方式完成的，根本就不适合于实际的开发，如果要想开发的更加实际一些，应该采用一块一块的读取和输出方式完成。

**【核心】范例：**改进的做法

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **public** **class** Copy {  **public** **static** **void** main(String[] args) **throws** Exception {  **if** (args.length != 2) { // 长度不是2，命令有错误  System.*out*.println("执行程序命令错误！");  System.*exit*(1); // 程序退出  }  **long** start = System.*currentTimeMillis*() ;  File inFile = **new** File(args[0]); // 要复制的源文件  **if** (!inFile.exists()) { // 源文件不存在  System.*out*.println("源文件不存在！");  System.*exit*(1); // 程序退出  }  File outFile = **new** File(args[1]); // 目标文件  InputStream input = **new** FileInputStream(inFile);  OutputStream output = **new** FileOutputStream(outFile);  **int** temp = 0;  **byte** data [] = **new** **byte**[1024 \* 1024] ; // 每次读取1024  **while** ((temp = input.read(data)) != -1) { // 有内容  output.write(data, 0, temp); // 输出一个内容  }  **long** end = System.*currentTimeMillis*() ;  System.*out*.println("拷贝完成，所花费的时间：" + (end - start));  input.close() ;  output.close() ;  }  } |

在日后的开发之中，如果给出的是一个InputStream，要求你通过这个InputStream读取出来的数据保存到指定的文件之中的时候，就采用以上代码，这些操作在日后学习Struts框架的时候会出现。

**3.3、转换流（了解，千万别会）**

现在对于IO操作就存在了字节流和字符流两种操作流，那么对于这两种操作流之间也是可以进行转换的，而转换的操作类有两个：

· **将字节输出流变为字符输出流（OutputStream 🡺 Writer）：**OutputStreamWriter；

· **将字节输入流变为字符输入流（InputStream 🡺 Reader）：**InputStreamReader。

以上两个类的定义结构和构造方法如下：

|  |  |
| --- | --- |
| **OutputStreamWriter：** | **InputStreamReader：** |
| public class OutputStreamWriter extends Writer | public class InputStreamReader extends Reader |
| public OutputStreamWriter(OutputStream out) | public InputStreamReader(InputStream in) |

通过以上的继承结构和构造方法可以清楚的发现，既然OutputStreamWriter是Wirter的子类，那么必然OutputStreamWriter可以通过Wirter类执行对象的向上转进行接收，而同时这个OutputStreamWriter类的构造方法可以接收OutputStream，这样就可以完成转换。

**范例：**将字节输出流变为字符输出流

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileOutputStream;  **import** java.io.OutputStream;  **import** java.io.OutputStreamWriter;  **import** java.io.Writer;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "hellodemo"  + File.*separator* + "test.txt"); // 定义文件路径  **if** (!file.getParentFile().exists()) {  file.getParentFile().mkdirs();// 创建父目录  }  OutputStream output = **new** FileOutputStream(file) ; // 字节输出流  Writer out = **new** OutputStreamWriter(output) ;  out.write("Hello World .") ;  out.close() ;  }  } |

**范例：**将字节输入流变为字符输入流

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.InputStream;  **import** java.io.InputStreamReader;  **import** java.io.Reader;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  File file = **new** File("D:" + File.*separator* + "hellodemo"  + File.*separator* + "test.txt"); // 定义文件路径  **if** (file.exists()) { // 文件存在则可以读取  InputStream input = **new** FileInputStream(file) ;  Reader in = **new** InputStreamReader(input) ; // 将字节输入流变为字符输入流  **char** data[] = **new** **char**[1024]; // 假设要读的长度是1024  **int** len = in.read(data) ; // 读取数据，返回读取个数  input.close() ; // 关闭  in.close() ;  System.*out*.println("读取的数据是：【" + **new** String(data, 0, len) + "】");  }  }  } |

本代码在实际工作的时候出现的几率不会超过1%设置更低。讲解这个类的目的主要是希望通过这个类来观察一下之前文件字符流和文件字节流的定义区别。

对于文件操作可以使用FileInputStream、FileOutputStream、FileReader、FileWriter四个类，那么下面分别观察这四个类的继承结构。

**第一组：**观察FileInputStream、FileOutputStream类的继承结构

|  |  |
| --- | --- |
| **FileInputStream：** | **FileOutputStream：** |
| java.lang.Object  |- **java.io.InputStream**  |- java.io.FileInputStream | java.lang.Object  |- **java.io.OutputStream**  |- java.io.FileOutputStream |

**第二组：**观察FileReader、FileWriter类的继承结构

|  |  |
| --- | --- |
| **FileReader：** | **FileWriter：** |
| java.lang.Object  |- **java.io.Reader**  |- java.io.InputStreamReader  |- java.io.FileReader | java.lang.Object  |- **java.io.Writer**  |- java.io.OutputStreamWriter  |- java.io.FileWriter |

通过以上的继承关系也可以发现，实际上所有的字符数据都是需要进行转换的，依靠转换流完成，以后真正保存或者是传输的数据是不可能有字符的，全部都是字节，而字符只是在电脑之中处理后的结果。

**3.4、内存操作流（次重点，现在不用，以后用的）**

在讲解之前首先来思考一个问题：就是说如果现在某个操作必须发生IO，但是又不希望有一些临时文件产生的话，那么现在肯定无法使用之前的文件操作流，所以为了解决这样的问题，提供了内存操作流，即：以内存进行操作的终端，以发生IO操作关系。

对于内存操作流也是分为两组：

· **字节内存操作流：**内存输入流（ByteArrayInputStream）、内存输出流（ByteArrayOutputStream）；

· **字符内存操作流：**内存输入流（CharArrayReader）、内存输出流（CharArrayWriter）。

以后的开发之中主要以字节的内存操作流为主，而如果要想学习这两个字节操作流之前，必须首先观察这两个类的继承结构以及构造方法的定义。

|  |  |
| --- | --- |
| **ByteArrayInputStream：** | **ByteArrayOutputStream：** |
| java.lang.Object  |- **java.io.InputStream**  |- java.io.ByteArrayInputStream | java.lang.Object  |- **java.io.OutputStream**  |- java.io.ByteArrayOutputStream |
| public ByteArrayInputStream(byte[] buf) | public ByteArrayOutputStream() |

下面使用之前的文件操作流和内存操作流做一个对比：

· 文件操作流的形式：

|- FileOutputStream：程序 🡺 OutputStream 🡺 输出到文件；

|- FileInputStream：程序 🡸 InputStream 🡸 文件；

· 内存操作流的形式：

|- ByteArrayInputStream：程序 🡺 InputStream 🡺 输出到内存；

|- ByteArrayOutputStream：程序 🡸 OutputStream 🡸 内存。

虽然从概念上是反的，但是从使用形式上而言，依然是InputStream负责输入，OutputStream负责输出。

**范例：**使用内存操作流完成一个字符串大小写字母的转换操作

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.ByteArrayInputStream;  **import** java.io.ByteArrayOutputStream;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "hello world ."; // 有非字母组成  InputStream input = **new** ByteArrayInputStream(str.getBytes()); // 将数据输出到内存之中  OutputStream output = **new** ByteArrayOutputStream(); // 准备从内存之中读取数据  **int** temp = 0;  **while** ((temp = input.read()) != -1) {  output.write((**char**) Character.*toUpperCase*(temp)); // 所有内容都在字节输出流中  }  String newStr = output.toString(); // 取数据  output.close();  input.close();  System.*out*.println(newStr);  }  } |

本程序之中发生了IO操作，但是并没有临时文件的产生，一切的操作都是以内存为主。此知识的使用是在日后学习AJAX + XML操作的时候才会用到，因为DOM解析一定需要内存操作流支持。

通过本程序也应该可以发现一个很熟悉的身影，观察一下继承结构。

同样都是输出（write()）或者是输入（read()）方法，但是根据实例化父类的子类对象不同，输入或输出的位置也不同，此处就很好的体现了面向对象的多态性。

**3.5、字符编码（了解）**

在计算机的世界之中，所有的显示文字都是按照其指定的数字编码进行保存的，在以后进行程序的开发之中，会经常见到一些的一些常见的编码：

· GBK / GBK2312：表示国标中文编码，其中GBK是包含简体中文和繁体中文，而GB2312只有简体；

· ISO 8859-1：是一种国际通用编码，可以表示任何文字，但是对于窗格文字需要进行一些转码；

· UNICODE：使用了十六进制完成的编码，可以准确的表示出任何的语言文字；

· UTF-8编码：部分编码使用UNICODE，而一些编码继续使用像ISO 8859-1类型的编码，适合于网络传输，在以后的所有的项目开发之中，都必须采用此编码。可是考虑到日后学习的方便，几乎都会使用命令行进行操作，所以命令行只支持GBK编码，UTF不支持，一旦程序设置了UTF编码，那么肯定通过命令行查看的就是乱码。

在开发之中经常会遇见乱码的问题。所谓的乱码问题的核心在于编码和解码不统一。如果要想正确的避免项目之中出现的乱码，那么首先就应该知道环境之中所支持的编码是什么。

**范例：**查看当前环境下的所有属性

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  System.*getProperties*().list(System.*out*);  }  } |

这个时候显示出来的信息是很多的，这里面有专门的编码选项“file.encoding=GBK”，也就是说如果没有任何的意外，所有的文字编码都是GBK。

**范例：**手工出现乱码

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileOutputStream;  **import** java.io.OutputStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  OutputStream output = **new** FileOutputStream(**new** File("D:"  + File.*separator* + "text.txt"));  output.write("世界，你好！".getBytes("ISO8859-1")) ;  output.close() ;  }  } |

但是以上的问题是我们自己编写代码测试的，本身不具备很强的说明意义，等日后学习到Java WEB开发的时候一定可以感受到编码所带来的问题。

**3.6、打印流（重点）**

如果说现在要想输出数据，肯定使用OutputStream或者是Writer，那么请问，这两个操作类在执行输出的时候你认为它好用吗？

如果现在要想输出字符串，使用Writer可以直接输出，而使用OutputStream还需要将字符串变为字节数组，那么如果现在要想输出数字（int型或double型），还需要将这些数据先变为字符串，之后再变为字节数组输出，所以，如果用户直接调用OutputStream或Writer输出的时候本身并不方便，所以在这个时候可以想办法将OutputStream或Wirter变得加强一些，定义一个专门的工具类：PrintUtil.java。

**范例：**编写一个输出功能类

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.OutputStream;  **class** PrintUtil {  **private** OutputStream output = **null** ;  **public** PrintUtil(OutputStream output) { // 通过构造方法设置输出的位置  **this**.output = output ;  }  **public** **void** print(String str) {  **try** {  **this**.output.write(str.getBytes()) ;  } **catch** (IOException e) {  e.printStackTrace();  }  }  **public** **void** println(String str) {  **this**.print(str.concat("\r\n")) ;  }  **public** **void** print(**int** num){  **this**.print(String.*valueOf*(num)) ;  }  **public** **void** println(**int** num) {  **this**.println(String.*valueOf*(num)) ;  }  **public** **void** print(**double** num){  **this**.print(String.*valueOf*(num)) ;  }  **public** **void** println(**double** num) {  **this**.println(String.*valueOf*(num)) ;  }  **public** **void** close() {  **try** {  **this**.output.close() ;  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  PrintUtil tools = **new** PrintUtil(**new** FileOutputStream(**new** File("D:"  + File.*separator* + "test.txt")));  tools.print("姓名：");  tools.println("张三。");  tools.print(1);  tools.print(" + ");  tools.print(1);  tools.print(" = ");  tools.println(1 + 1);  tools.close();  }  } |

如果此时不看PrintUtil类，可以清楚的发现，客户端的输出操作要比直接使用OutputStream输出更加简单，那么既然我们都可以想到的问题，Java也早就想到了，为此它专门提供了两个类：字节打印流类（PrintStream）、字符打印流类（PrintWriter），现在还是以字节打印流为主，而字符打印流主要是可以方便的输出中文。

**范例：**直接使用打印流

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileOutputStream;  **import** java.io.PrintStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  PrintStream tools = **new** PrintStream(**new** FileOutputStream(**new** File("D:"  + File.*separator* + "test.txt")));  tools.print("姓名：");  tools.println("张三。");  tools.print(1);  tools.print(" + ");  tools.print(1);  tools.print(" = ");  tools.println(1 + 1);  tools.close();  }  } |

首先来观察一下PrintStream类和PrintWriter类的继承结构和构造方法：

|  |  |
| --- | --- |
| **PrintStream：** | **PrintWriter：** |
| java.lang.Object  |- java.io.**OutputStream**  |- java.io.FilterOutputStream  |- java.io.PrintStream | java.lang.Object  |- java.io.Writer  |- java.io.PrintWriter |
| public PrintStream(**OutputStream** out) | public PrintWriter(**Writer** out) |

看见了以上的结构，可能第一反应就属于代理设计模式，但是它并不是代理，代理设计模式的特点：要在接口上完成、用户调用代理主题方法的时候依然是接口之中定义的方法。而此时的PrintStream类调用的绝对不是OutputStream类之中定义的一系列的write()方法。虽然PrintStream在外表上的操作方法产生了变化，但是实际上依然执行的是OutputStream类所定义的操作，所以本质没有发生变化，只是提供了一些更加方便的功能支持，所以这种设计思路在设计模式上讲称为装饰设计模式。

上一个代码是在文件操作流上使用了打印流，下面也可以在内存操作流上使用打印流，相当于通过打印流为内存输出流进行了一种处理载体。

**范例：**在内存流上使用打印流

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.ByteArrayInputStream;  **import** java.io.ByteArrayOutputStream;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **import** java.io.PrintStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String str = "hello world ."; // 有非字母组成  InputStream input = **new** ByteArrayInputStream(str.getBytes()); // 将数据输出到内存之中  OutputStream output = **new** ByteArrayOutputStream(); // 准备从内存之中读取数据  PrintStream out = **new** PrintStream(output) ; // 使用打印流间接调用了内存输出流  **int** temp = 0;  **while** ((temp = input.read()) != -1) {  out.print((**char**) Character.*toUpperCase*(temp)); // 所有内容都在字节输出流中  }  String newStr = output.toString(); // 取数据  output.close();  input.close();  System.*out*.println(newStr);  }  } |

而通过本程序也可以发现，如果要想决定输出的位置完全由子类决定，打印流照单全收。

【了解】但是在JDK 1.5之后，打印流也进行了更新，增加了一新的方法，格式化输出：

· **格式化输出：**public PrintStream printf(String format, Object... args)

当看到此方法名称的时候应该首先想到的是C语言中的输出，而现在Java也具备了同样的功能，而输出的时候可以使用一些标记来表示要输出的内容，例如：字符串（%s）、整数（%d）、小数（%m.nf）、字符（%c）等，之所以有这样的输出，其实目的还是在于抢夺C语言的开发人员市场。

**范例：**观察格式化输出

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileOutputStream;  **import** java.io.PrintStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  PrintStream tools = **new** PrintStream(**new** FileOutputStream(**new** File("D:"  + File.*separator* + "test.txt")));  String name = "张三";  **int** age = 20;  **double** score = 89.9876321; // 成绩，小数点太多  tools.printf("姓名：%s，年龄：%d，成绩：% 5.2f ", name, age, score);  tools.close();  }  } |

而在JDK 1.5之后增加字符串格式化操作类不光有PrintStream，还有String类，String类也提供了一个格式化字符串的操作方法：public static String format(String format, Object... args)

**范例：**格式化字符串

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  String name = "张三";  **int** age = 20;  **double** score = 89.9876321; // 成绩，小数点太多  String str = String.*format*("姓名：%s，年龄：%d，成绩：% 5.2f ", name, age, score);  System.*out*.println(str);  }  } |

虽然格式化字符串可以执行准确的四舍五入操作，但是这种处理完的数据都是String型，而实际的工作中，如果要四舍五入，肯定还是要编写BigDecimal类完成。

**以后只要是程序输出数据的操作，都使用PrintStream类。**

**3.7、System类（了解）**

在PrintStream类之中发现有许多的println()方法，实际上这些方法在之前一直在使用，下面来观察一下System类的一些定义，有三个与IO有关的常量：

· **错误输出：**public static final PrintStream err；

· **系统输出：**public static final PrintStream out；

· **系统输入：**public static final InputStream in；

**1、 错误输出（本人实在是不知道这个能做什么，现在基本上没人用了）**

**范例：**观察错误输出

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  **try** {  Integer.*parseInt*("abc");  } **catch** (Exception e) {  System.*err*.println(e);  System.*out*.println(e);  }  }  } |

从Java本身的规定是这样解释的：System.err输出的是不希望用户看见的错误，而System.out输出的是希望用户看见的错误，两者没什么区别，都用不上，知道就行了。

**2、 系统输出**

系统输出是将所有信息输出到指定的输出设备上 —— 显示器。而System.out本身是属于PrintStream对象，而PrintStream是OutputStream子类，所以现在实际上可以利用System.out为OutputStream类执行实例化操作。

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.OutputStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  OutputStream output = System.*out* ; // 具备系统输出  output.write("Hello World .".getBytes()) ;  }  } |

本程序没有任何的意义，而讲解主要目的就希望可以理解：OutputStream会根据实例化它的子类或对象不同，输出的位置也不同。

**3、 系统输入**

系统输入针对于标准的输入设备 —— 键盘，也就是俗称的键盘输入数据，但是System.in返回的是InputStream型的数据，所以下面编写一个操作由键盘输入数据。

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.InputStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  InputStream input = System.*in*;  **byte** data[] = **new** **byte**[1024];  System.*out*.print("请输入数据：");  **int** len = **input.read(data)**; 🡺 等待用户输入，程序进入到阻塞状态  System.*out*.println("输入的内容是：" + **new** String(data, 0, len));  }  } |

除了实例化InputStream类的对象不同之外，其他的地方和之前文件输入数据没有任何的区别，但是这个程序本身有问题，已经开辟的空间大小是1024，如果输入的数据超过1024呢？发现只会接收满足于指定长度的数据，程序有bug，那么最好的解决方法是不设置长度，输入一个读取一个，一直到用户不输入为止。

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.InputStream;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  InputStream input = System.*in*;  StringBuffer buf = **new** StringBuffer();  System.*out*.print("请输入数据：");  **int** temp = 0;  **while** ((temp = input.read()) != -1) { // 用户可以一直输入下去  **if** (temp == '\n') { // 输入结束  **break**; // 退出循环  }  buf.append((**char**) temp);  }  System.*out*.println("输入的内容是：" + buf);  }  } |

这个时候看似长度问题解决了，并且可以由用户随意输入任意长度的数据，但是新的问题又来了，此时的程序是按照一个一个字节的方式取的内容，所以如果输入的是中文，那么一个中文被读了两次，所以一定会出现乱码。

**3.8、缓冲区操作：BufferedReader（次重点，15%）**

如果说现在把所有的输入数据都放在一起了，一次性读取出来了，那么这个时候肯定就能够避免中文问题了，而这一操作就必须依靠缓冲区操作流完成，对于缓冲区的读取在IO包中定义了两种类：BufferedInputStream、BufferedReader，但是考虑到本次操作有中文的问题，肯定使用BufferedReader类完成操作，下面就需要来观察一下BufferedReader类的继承结构、构造方法、主要操作方法。

|  |  |
| --- | --- |
| **继承结构：** | java.lang.Object  |- java.io.Reader  |- java.io.BufferedReader |
| **构造方法：** | public BufferedReader(**Reader** in) |
| **读取操作：** | public String readLine() throws IOException |

实际上之所以不使用BufferedInputStream而使用BufferedReader类的另外一个原因在于BufferedReader类有一个方法可以读取一行数据，而这个方法BufferedInputStream类没有。

在BufferedReader类之中的readLine()方法返回的是String数据，而一旦返回的数据类型是String就非常方便：

· String类之中有大量的方法供用户操作；

· String类可以直接使用正则验证其格式；

· String类可以向基本数据类型或日期型进行转型操作。

**范例：**使用BufferedReader进行数据读取

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.BufferedReader;  **import** java.io.InputStreamReader;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  BufferedReader buf = **new** BufferedReader(  **new** InputStreamReader(System.*in*));  System.*out*.print("请输入数据：");  String str = buf.readLine() ;  System.*out*.println("输入的内容是：" + str);  }  } |

此时输入的数据不再存在长度的限制了，并且可以方便的进行中文数据的输入。

**范例：**对输入的数据进行验证，现在要求一个用户由键盘输入一个数字，而后进行数字的乘法错误，如果用户输入的不是数字，则需要提醒用户重新输入

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.BufferedReader;  **import** java.io.InputStreamReader;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  BufferedReader buf = **new** BufferedReader(  **new** InputStreamReader(System.*in*));  **boolean** flag = **true**;  **int** num = 0;  **while** (flag) { // 处于循环操作  System.*out*.print("请输入数据：");  String str = buf.readLine();  **if** (str.matches("\\d+")) { // 是数字  num = Integer.*parseInt*(str);  flag = **false**; // 退出循环  } **else** {  System.*out*.print("您输入的不是数字！");  }  }  System.*out*.println("输入的内容是：" + num \* num);  }  } |

通过本程序只是希望提醒，如果要想使用BufferedReader多次输入数据的话，重复调用readLine()方法即可。

**3.9、JDK 1.5的新支持：Scanner（重点）**

在JDK 1.5之后为了方便用户进行输入数据的操作，专门提供了一个java.util.Scanner类，这个类是作为了一个工具类出现的，在Scanner类之中定义了如下的一些主要操作方法：

**· 构造方法：public Scanner(InputStream source)；**

**· 判断是否有数据：public boolean hasNextXxx()；**

**· 取得数据：public 数据类型 nextXxx()；**

**· 修改分隔符：public Scanner useDelimiter(String pattern)；**

以后调用的时候在执行nextXxx()之前一定要首先使用hasNextXxx()判断是否有指定格式的数据出现。

**范例：**通过Scanner进行数据的输入

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Scanner;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Scanner scan = **new** Scanner(System.*in*);  scan.useDelimiter("\n"); // 只将换行作为分隔符  System.*out*.print("请输入数据：");  **if** (scan.hasNext()) { // 有内容  String str = scan.next();  System.*out*.println("输入数据：" + str);  }  }  } |

**范例：**使用Scanner判断输入数据是否是int

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Scanner;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Scanner scan = **new** Scanner(System.*in*);  System.*out*.print("请输入数据：");  **if** (scan.hasNextDouble()) { // 有内容  **double** data = scan.nextDouble() ;  System.*out*.println("输入数据：" + data);  } **else** {  System.*out*.println("输入的数据不是数字！");  }  }  } |

在Scanner类之中，useDelimiter()方法的输入针对于字符串，但是其他的数据类型并不方便使用。

**范例：**利用正则验证

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.text.SimpleDateFormat;  **import** java.util.Date;  **import** java.util.Scanner;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Scanner scan = **new** Scanner(System.*in*);  System.*out*.print("请输入您的生日：");  **if** (scan.hasNext("\\d{4}-\\d{2}-\\d{2}")) { // 有内容  String str = scan.next("\\d{4}-\\d{2}-\\d{2}") ;  Date date = **new** SimpleDateFormat("yyyy-MM-dd").parse(str) ;  System.*out*.println(date);  } **else** {  System.*out*.println("输入的数据不是数字！");  }  }  } |

如果说现在由程序向文件输出内容使用PrintStream可以方便完成，但是反过来，如果说现在在程序之中读取文件内容，如果使用InputStream类读取并不方便，而使用Scanner就可以方便完成了。

**范例：**使用Scanner读取文件

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.util.Scanner;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Scanner scan = **new** Scanner(**new** FileInputStream(**new** File("D:"  + File.*separator* + "test.txt")));  scan.useDelimiter("\n") ;  **while** (scan.hasNext()) {  System.*out*.println(scan.next());  }  scan.close() ;  }  } |

这个时候的操作发现明显很简单，所以这样就得出一个结论：**以后程序输出数据使用打印流，程序输入数据使用Scanner。**

## 3.10、对象序列化（核心中的战斗机）

### 3.10.1、对象序列化的概念（核心）

所谓的对象序列化指的是可以将在内存之中保存的对象数据（对象里面包含属性），进行二进制数据传输的一种操作，而如果要想完成这样的二进制操作，那么对象所在的类就必须实现java.io.Serializable接口。这个接口和Cloneable接口道理一样，都属于一种标识接口，表示一种能力。

**范例：**定义可以被序列化的类

|  |
| --- |
| @SuppressWarnings("serial")  **class** Person **implements** **Serializable**{  **private** String name ;  **private** **int** age ;  **public** Person(String name,**int** age) {  **this**.name = name ;  **this**.age = age ;  }  @Override  **public** String toString() {  **return** "姓名：" + **this**.name + "，年龄：" + **this**.age;  }  } |

以后Person类的对象（主要是里面的属性）就可以进行二进制数据的传输了。

### 3.10.2、实现序列化和反序列化（了解）

如果要想进行对象的序列化和反序列化的手工操作，在Java之中提供了两个操作类：ObjectOutputStream、ObjectInputStream，而这两个类的继承结构、构造方法、操作方法定义如下：

|  |  |
| --- | --- |
| **ObjectOutputStream：** | **ObjectInputStream：** |
| java.lang.Object  |- java.io.**OutputStream**  |- java.io.ObjectOutputStream | java.lang.Object  |- java.io.**InputStream**  |- java.io.ObjectInputStream |
| public ObjectOutputStream(**OutputStream** out) throws IOException | public ObjectInputStream(**InputStream** in) throws IOException |
| public final void writeObject(**Object** obj) throws IOException | public final **Object** readObject() throws IOException, ClassNotFoundException |

**范例：**实现的序列化操作

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Person per = **new** Person("张三", 20);  ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream(  **new** File("D:" + File.*separator* + "person.ser")));  oos.writeObject(per); // 输出对象  oos.close();  }  } |

**范例：**执行反序列化操作

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream(  **new** File("D:" + File.*separator* + "person.ser")));  Person per = (Person) ois.readObject(); // 接收对象  ois.close();  System.*out*.println(per);  }  } |

在日后的开发之中，会由容器帮助用户自动的完成这种序列化和反序列化的操作，所以本次的功能只是作为了解，不过如果从事的是Android开发，并且要使用java独有的数据格式（序列化格式）就必须自己去手工编写对象序列化的输入和输出操作了。

### 3.10.3、transient关键字

默认的情况下，当一个类的对象被序列化的时候，这个类之中的所有属性都被保存下来，如果现在某些属性不希望被保存的话，那么可以使用transient进行声明。

**范例：**观察transient定义

|  |
| --- |
| **private** **transient** String name; // 不能被序列化  **private** **int** age; // 可以被序列化 |

**面试题：**请解释Java序列化的作用以及实现？

序列化操作的主要目的是为保证对象可以以二进制数据的方式进行传输，如果要想实现对象序列化，对象所在的类必须实现java.io.Serializable接口，默认情况下一个对象的所有属性都会被序列化下来，也可以使用transient关键字定义不被序列化的属性。

# 4、总结

1、 File类操作文件的几个主要方法；

2、 文件拷贝操作的思路；

3、 输出使用打印流，输入使用Scanner（如果Scanner不好使使用BufferedReader）；

4、 对象序列化的概念。

# 5、预习任务

1、 类集框架：Collection、List、Set、Map、Iterator、ListIterator、Enumeration、ArrayList、Vector、HashSet、TreeSet、HashMap、Hashtable、Properties、Stack；

2、 JDBC：连接数据库、调用SQL完成数据表的CRUD、分页操作；