**1 Java IO流的概念，分类**

**1.1 Java IO流的概念**

java的IO是实现输入和输出的基础，可以方便的实现数据的输入和输出操作。在java中把不同的输入/输出源（键盘，文件，网络连接等）抽象表述为“流”(stream)。**流的本质是数据传输，根据数据传输特性将流抽象为各种类，方便更直观的进行数据操作。**

Java的IO模型设计非常优秀，它使用Decorator(装饰者)模式，按功能划分Stream，您可以动态装配这些Stream，以便获得您需要的功能。 例如，您需要一个具有缓冲的文件输入流，则应当组合使用FileInputStream和BufferedInputStream。

**1.2.1按照流的流向分，可以分为输入流和输出流。**

* 输入流：只能从中读入数据，而不能向其写出数据。
* 输出流：只能向其写出数据，而不能向其读入数据。

**注：为什么叫写出和读入，这里涉及到一个方向问题，因为划分输入/输出流时是从程序运行所在的内存的角度来考虑的；将内存中的数据写出到硬盘中的文件，或者是将硬盘中文件的信息读入到内存。**

注：java的输入流主要是InputStream和Reader作为基类，而输出流则是主要由OutputStream和Writer作为基类。它们都是一些抽象基类，无法直接创建实例。

**Java io 中常见的类：**



**1.2.2 按照操作单元划分，可以划分为字节流和字符流。**

字符流的由来： 因为数据编码的不同，而有了对字符进行高效操作的流对象。本质其实就是基于字节流读取时，去查了指定的码表。  
字节流和字符流的区别：

1. 读写单位不同：字节流以字节（8bit）为单位，字符流以字符为单位，根据码表映射字符，一次可能读多个字节。
2. 处理对象不同：字节流能处理所有类型的数据（如图片、avi等），而字符流只能处理字符类型的数据。

* 字节流：一次读入或读出是8位二进制。
* 字符流：一次读入或读出是16位二进制。

**1.2.3 按照流的角色划分为节点流和处理流。**

* 节点流(低级流)：直接与数据源相连，读入或写出。直接使用节点流，读写不方便，为了更快的读写文件，才有了处理流。
* 处理流(高级流)和节点流一块使用，在节点流的基础上，再套接一层，套接在节点流上的就是处理流。如BufferedReader处理流的构造方法总是要带一个其他的流对象做参数。一个流对象经过其他流的多次包装，称为流的链接。

**1.3.1 流的原理浅析:**

java IO流共涉及40多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存在非常紧密的联系， Java IO流的40多个类都是从如下4个抽象类基类中派生出来的。

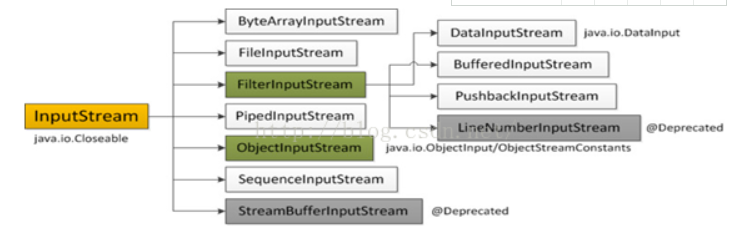
* **InputStream/Reader: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。**
* **OutputStream/Writer: 所有输出流的基类，前者是字节输出流，后者是字符输出流。**

1. 对于InputStream和Reader而言，它们把输入设备抽象成为一个”水管“，这个水管的每个“水滴”依次排列。
2. 字节流和字符流的处理方式其实很相似，只是它们处理的输入/输出单位不同而已。输入流使用隐式的记录指针来表示当前正准备从哪个“水滴”开始读取，每当程序从InputStream或者Reader里面取出一个或者多个“水滴”后，记录指针自定向后移动；除此之外，InputStream和Reader里面都提供了一些方法来控制记录指针的移动。
3. 对于OutputStream和Writer而言，它们同样把输出设备抽象成一个”水管“，只是这个水管里面没有任何水滴。
4. 当执行输出时，程序相当于依次把“水滴”放入到输出流的水管中，输出流同样采用隐示指针来标识当前水滴即将放入的位置，每当程序向 OutputStream 或者 Writer 里面输出一个或者多个水滴后，记录指针自动向后移动。

注：Java的处理流模型则体现了Java输入和输出流设计的灵活性。处理流的功能主要体现在以下两个方面。

* **性能的提高：主要以增加缓冲的方式来提供输入和输出的效率。**
* **操作的便捷：处理流可能提供了一系列便捷的方法来一次输入和输出大批量的内容，而不是输入/输出一个或者多个“水滴”。**

框架图：





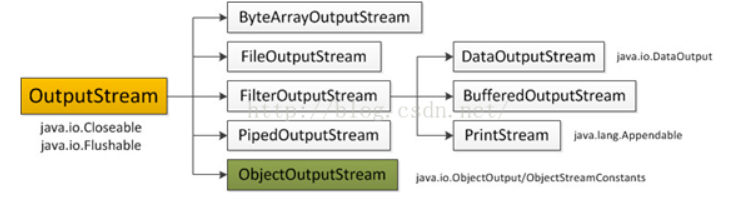
IO 中输入字节流的继承图可见上图，可以看出：

1.InputStream是所有的输入字节流的父类，它是一个抽象类。

2.ByteArrayInputStream、StringBufferInputStream(上图的StreamBufferInputStream)、FileInputStream是三种基本的介质流，它们分别从Byte数组、StringBuffer、和本地文件中读取数据。

3.PipedInputStream是从与其它线程共用的管道中读取数据.

4.ObjectInputStream和所有FilterInputStream的子类都是装饰流（装饰器模式的主角）



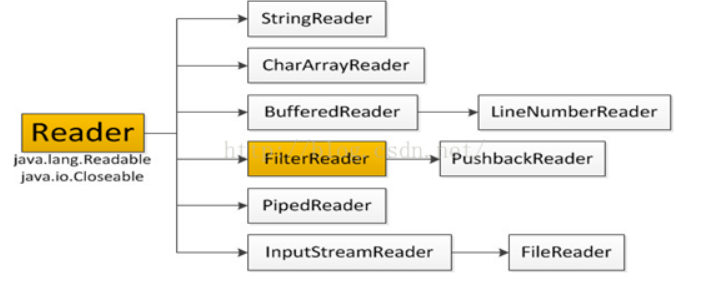


IO 中输出字节流的继承图可见上图，可以看出：

1.OutputStream是所有的输出字节流的父类，它是一个抽象类。

2.ByteArrayOutputStream、FileOutputStream是两种基本的介质流，它们分别向Byte数组、和本地文件中写入数据。PipedOutputStream是向与其它线程共用的管道中写入数据。

3.ObjectOutputStream和所有FilterOutputStream的子类都是装饰流。

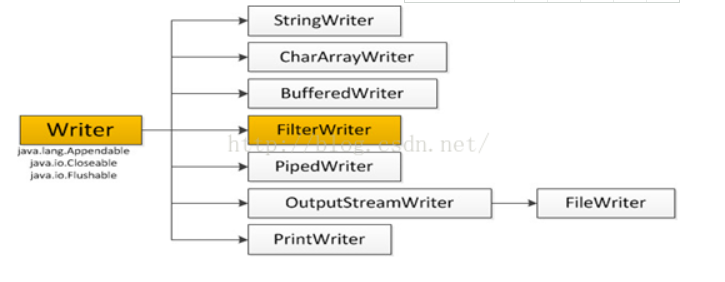


1.Reader是所有的输入字符流的父类，它是一个抽象类。

2.charreader、StringReader是两种基本的介质流，它们分别将Char数组、String中读取数据。PipedReader是从与其它线程共用的管道中读取数据。

3.BufferedReader很明显就是一个装饰器，它和其子类负责装饰其它Reader对象。

4.FilterReader是所有自定义具体装饰流的父类，其子类PushbackReader对Reader对象进行装饰，会增加一个行号。

5.InputStreamReader是一个连接字节流和字符流的桥梁，它将字节流转变为字符流。FileReader可以说是一个达到此功能、常用的工具类，在其源代码中明显使用了将FileInputStream转变为Reader的方法。我们可以从这个类中得到一定的技巧。Reader中各个类的用途和使用方法基本和InputStream中的类使用一致。后面会有Reader与InputStream的对应关系。

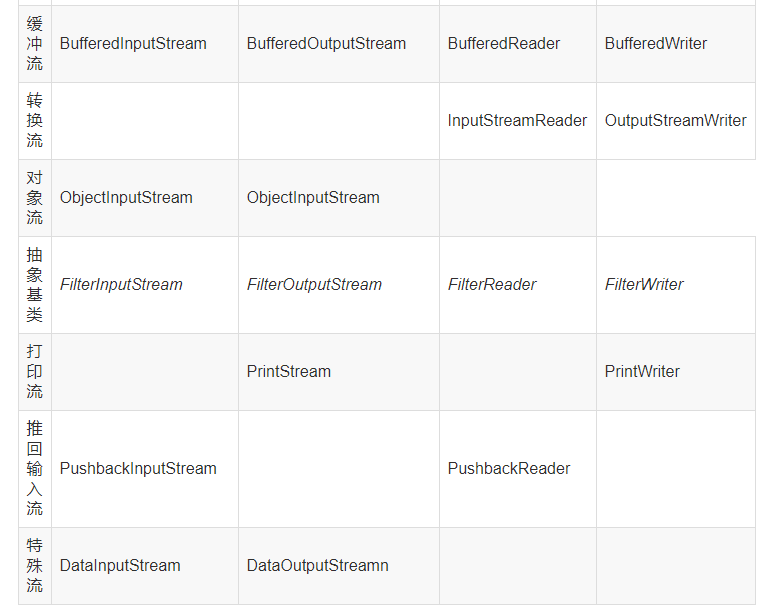
1.Writer是所有的输出字符流的父类，它是一个抽象类。

2.CharArrayWriter、StringWriter是两种基本的介质流，它们分别向Char数组、String中写入数据。PipedWriter是向与其它线程共用的管道中写入数据，

3.BufferedWriter是一个装饰器为Writer提供缓冲功能。

4.PrintWriter和PrintStream极其类似，功能和使用也非常相似。

5.OutputStreamWriter是OutputStream到Writer转换的桥梁，它的子类FileWriter其实就是一个实现此功能的具体类（具体可以研究一SourceCode）。功能和使用和OutputStream极其类似.



**转换流**的特点：

1. 其是字符流和字节流之间的桥梁

2. 可对读取到的字节数据经过指定编码转换成字符

3. 可对读取到的字符数据经过指定编码转换成字节

何时使用转换流？

1. 当字节和字符之间有转换动作时；

2. 流操作的数据需要编码或解码时；

具体的对象体现：

**转换流：**在IO中还存在一类是转换流，将字节流转换为字符流，同时可以将字符流转化为字节流。

1. InputStreamReader:字节到字符的桥梁

2. OutputStreamWriter:字节到字符的桥梁

OutputStreamWriter(OutStream out):将字节流以字符流输出。

InputStreamReader(InputStream in)：将字节流以字符流输入。

这两个流对象是字符体系中的成员，它们有转换作用，本身又是字符流，所以在构造的时候需要传入字节流对象进来。

**2.1 IO体系的基类（InputStream/Reader，OutputStream/Writer）**

字节流和字符流的操作方式基本一致，只是操作的数据单元不同——字节流的操作单元是字节，字符流的操作单元是字符。

InputStream和Reader是所有输入流的抽象基类，本身并不能创建实例来执行输入，但它们将成为所有输入流的模板，所以它们的方法是所有输入流都可使用的方法。  
在**InputStream**里面包含如下3个方法。

* abstract int read(); 从输入流中读取单个字节（相当于从水管中取出一滴水），返回所读取的字节数据（字节数据可直接转换为int类型）。
* int read(byte[] b)从输入流中最多读取b.length个字节的数据，并将其存储在字节数组b中，返回实际读取的字节数。
* int read(byte[] b,int off,int len); 从输入流中最多读取len个字节的数据，并将其存储在数组b中，放入数组b中时，并不是从数组起点开始，而是从off位置开始，返回实际读取的字节数。

这个抽象类还重载了其它的read方法，但都是在底层调用了上面这个读取单字节的抽象的read()方法。该抽象类还有如下方法：

　　①、abstract int read();

　　②、int read(byte[] b)，最大读取b.length个字节数据；

　　③、int read(byte[] b, int off, int len)，最大读取len个字节数据到b字节数组中，从off位置开始存放；

　　④、long skip(long n)，在输入流中跳过n个字节，返回实际跳过的字节数。当遇到末尾的时候实际跳过的数据可能小于n；

　　⑤、int available()，返回在不阻塞的情况下流中的可以读取的字节数；

　　⑥、void close()，关闭流；

　　⑦、void mark(int readlimit)，在输入流的当前位置打一个标记（注：不是所有的流都支持这一特性）；

　　⑧、void reset()，返回到最后一个标记处。随后调用read方法会从最后一个标记处重新读取字节数据。如果当前没有标记，则不会有任何变化；

　　⑨、boolean markSupported()，判断当前流是否支持标记操作；

在**Reader**中包含如下3个方法。

* int read(); 从输入流中读取单个字符（相当于从水管中取出一滴水），返回所读取的字符数据（字节数据可直接转换为int类型）。
* int read(char[] b)从输入流中最多读取b.length个字符的数据，并将其存储在字节数组b中，返回实际读取的字符数。
* abstract int read(char[] b,int off,int len); 从输入流中最多读取len个字符的数据，并将其存储在数组b中，放入数组b中时，并不是从数组起点开始，而是从off位置开始，返回实际读取的字符数。

关于inputstream返回值的说明：

Read()：

1) public abstract int read() throws IOException ; 从输入流中读取数据的下一个字节，返回0到255范围内的int字节值。如果因为已经到达流末尾而没有可用的字节，则返回-1。在输入数据可用、检测到流末尾或者抛出异常前，此方法一直阻塞。

Read(byte[] b):

2) public int read(byte[] b) throws IOException; 从输入流中读取一定数量的字节，并将其存储在缓冲区数组 b 中。以整数形式返回实际读取的字节数。在输入数据可用、检测到文件末尾或者抛出异常前，此方法一直阻塞。如果 b 的长度为 0，则不读取任何字节并返回 0；否则，尝试读取至少一个字节。如果因为流位于文件末尾而没有可用的字节，则返回值 -1；否则，至少读取一个字节并将其存储在 b 中。将读取的第一个字节存储在元素 b[0] 中，下一个存储在 b[1] 中，依次类推。读取的字节数最多等于 b 的长度。设 k 为实际读取的字节数；这些字节将存储在 b[0] 到 b[k-1] 的元素中，不影响 b[k] 到 b[b.length-1] 的元素。

Read( byte[] b , int off , int len):

3) public int read(byte[] b, int off, int len) throws [IOException](https://blog.csdn.net/qq_37835596/article/details/76559572)；读取len字节的数据从输入流到一个字节数组;试图读取多达len字节,但可能读取到少于len字节。返回实际读取的字节数为整数。第一个字节存储读入元素b[off],下一个b[off+1],等等。读取的字节数是最多等于len。read(byte[] b)就是相当于read(byte[] b, 0, b.length).所以两者差不多.性质一样;

**que1:** 为什么read()无参方法读取一个字节,返回的是一个int类型,而不是一个byte类型?

**ans1:** 因为字节输入流可以操作任意类型的文件,比如图片音频等,这些文件底层都是以二进制形式的存储的,如果每次读取都返回byte,有可能在读到中间的时候遇到111111111(文件的底层按补码来存储的);那么这11111111是byte类型的-1,我们的程序是遇到-1就会停止不读了,后面的数据就读不到了,所以在读取的时候用int类型接收,如果11111111会在其前面补上24个0凑足4个字节,那么byte类型的-1就变成int类型的255了这样可以保证整个数据读完,而结束标记的-1就是int类型;

**InputStream和Reader**提供的一些移动指针的方法：

* void mark(int readAheadLimit); 在记录指针当前位置记录一个标记（mark）。

注释：（mark就像书签一样，在这个BufferedReader对应的buffer里作个标记，以后再调用reset时就可以再回到这个mark过的地方。mark方法有个参数，通过这个整型参数，你告诉系统，希望在读出这么多个字符之前，这个mark保持有效。读过这么多字符之后，系统可以使mark不再有效，而你不能觉得奇怪或怪罪它。这跟buffer有关，如果你需要很长的距离，那么系统就必须分配很大的buffer来保持你的mark。       
reader.mark(50);//要求在50个字符之内，这个mark应该保持有效，系统会保证buffer至少可以存储50个字符；int a = reader.read();//读了一个字符；int b = reader.read();//又读了一个字符，做了某些处理，发现需要再读一次；reader.reset(); reader.read();//读到的字符和a相同；reader.read();//读到的字符和b相同）

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

**OutputStream和Writer**的用法也非常相似，两个流都提供了如下三个方法：

(outputstream: abstract void write(int c);writer: abstract void write(char[] buf, int off,int len ) );

* void write(int c); 将指定的字节/字符输出到输出流中，其中c即可以代表字节，也可以代表字符。
* void write(byte[]/char[] buf); 将字节数组/字符数组中的数据输出到指定输出流中。
* void write(byte[]/char[] buf, int off,int len ); 将字节数组/字符数组中从off位置开始，长度为len的字节/字符输出到输出流中。

因为字符流直接以字符作为操作单位，所以Writer可以用字符串来代替字符数组，即以String对象作为参数。Writer里面还包含如下两个方法。

* void write(String str); 将str字符串里包含的字符输出到指定输出流中。
* void write (String str, int off, int len); 将str字符串里面从off位置开始，长度为len的字符输出到指定输出流中。

**2.2 IO体系的基类文件流的使用(FileInputStream/FileReader ，FileOutputStream/FileWriter)**

1. 使用FileInputStream读取文件：

@Test

public void fileInputStreamTest() throws IOException {

FileInputStream fis = null;

try {

// 创建字节输入流

fis = new FileInputStream(

"D:/test/test.txt");

// 创建一个长度为1024的竹筒

byte[] b = new byte[1024];

// 用于保存的实际字节数

int hasRead = 0;

// 使用循环来重复取水的过程

while ((hasRead = fis.read(b)) > 0) {

// 取出竹筒中的水滴（字节），将字节数组转换成字符串进行输出, 后面的gbk是为了处理中文乱码问题

System.out.print(new String(b, 0, hasRead, "gbk"));

}

} catch (IOException e) {

e.printStackTrace();

} finally {

fis.close();

}

}

注：上面程序最后使用了fis.close()来关闭该文件的输入流，与JDBC编程一样，程序里面打开的文件IO资源不属于内存的资源，垃圾回收机制无法回收该资源，所以应该显示的关闭打开的IO资源。Java 7改写了所有的IO资源类，它们都实现了AntoCloseable接口，因此都可以通过自动关闭资源的try语句来关闭这些IO流。

1. 使用FileReader读取文件：

@Test

public void fileReaderTest() throws IOException{

FileReader fis = null;

try {

// 创建字节输入流

fis = new FileReader("D:/test/test.txt");

// 创建一个长度为1024的竹筒

char[] b = new char[1024];

// 用于保存的实际字节数

int hasRead = 0;

// 使用循环来重复取水的过程

while ((hasRead = fis.read(b)) > 0) {

// 取出竹筒中的水滴（字节），将字节数组转换成字符串进行输出

System.out.print(new String(b, 0, hasRead));

}

} catch (IOException e) {

e.printStackTrace();

} finally {

fis.close();

}

}

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

1. FileOutputStream的用法：

@Test

public void fileOutputStreamTest() throws IOException {

FileInputStream fis = null;

FileOutputStream fos = null;

try {

// 创建字节输入流

fis = new FileInputStream("D:/test/test.txt");

// 创建字节输出流

fos = new FileOutputStream("D:/test/test2.txt");

byte[] b = new byte[1024];

int hasRead = 0;

// 循环从输入流中取出数据

while ((hasRead = fis.read(b)) > 0) {

// 每读取一次，即写入文件输入流，读了多少，就写多少。

fos.write(b, 0, hasRead);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

fis.close();

fos.close();

}

}

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

4.缓冲流的使用（BufferedInputStream/BufferedReader, BufferedOutputStream/BufferedWriter）：  
字节缓冲流

@Test

public void bufferedTest() throws IOException {

FileInputStream fis = null;

FileOutputStream fos = null;

BufferedInputStream bis = null;

BufferedOutputStream bos = null;

try {

// 创建字节输入流

fis = new FileInputStream("D:/test/test.txt");

// 创建字节输出流

fos = new FileOutputStream("D:/test/test3.txt");

// 创建字节缓存输入流

bis = new BufferedInputStream(fis);

// 创建字节缓存输出流

bos = new BufferedOutputStream(fos);

byte[] b = new byte[1024];

int hasRead = 0;

// 循环从缓存流中读取数据

while ((hasRead = bis.read(b)) > 0) {

// 向缓存流中写入数据，读取多少写入多少

bos.write(b, 0, hasRead);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

bis.close();

bos.close();

}

}

可以看到使用字节缓存流读取和写入数据的方式和文件流（FileInputStream,FileOutputStream）并没有什么不同，只是把处理流套接到文件流上进行读写。

上面代码中我们使用了缓存流和文件流，但是我们只关闭了缓存流。这个需要注意一下，当我们使用处理流套接到节点流上的使用的时候，只需要关闭最上层的处理就可以了。java会自动帮我们关闭下层的节点流。

**2.3 转换流的使用（InputStreamReader/OutputStreamWriter）:**

下面以获取键盘输入为例来介绍转换流的用法。java使用System.in代表输入。即键盘输入，但这个标准输入流是InputStream类的实例，使用不太方便，而且键盘输入内容都是文本内容，所以可以使用InputStreamReader将其包装成BufferedReader,利用BufferedReader的readLine()方法可以一次读取一行内容，如下代码所示:

1. 转换流的使用

@Test

public void changeStreamTest(){

try {

// 将System.in对象转化为Reader对象

InputStreamReader reader=new InputStreamReader(System.in);

//将普通的Reader包装成BufferedReader

BufferedReader bufferedReader=new BufferedReader(reader);

String buffer=null;

while ((buffer=bufferedReader.readLine())!=null){

// 如果读取到的字符串为“exit”,则程序退出

if(buffer.equals("exit")){

System.exit(1);

}

//打印读取的内容

System.out.print("输入内容："+buffer);

}

}catch (IOException e){

e.printStackTrace();

}finally {

}

}

上面程序将System.in包装成BufferedReader,BufferedReader流具有缓存功能，它可以一次读取一行文本——以换行符为标志，如果它没有读到换行符，则程序堵塞。等到读到换行符为止。运行上面程序可以发现这个特征，当我们在控制台执行输入时，只有按下回车键，程序才会打印出刚刚输入的内容。

**2.4 对象流的使用（ObjectInputStream/ObjectOutputStream）的使用：**

对象类：

package com.design.prompt;

import java.io.Serializable;

public class UserTest implements Serializable {

/\*\*

\*

\*/

private static final long serialVersionUID = 6522988290199824802L;

private String username;

transient String password;

public UserTest(){

}

public UserTest(String username, String password) {

super();

this.username = username;

this.password = password;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

@Override

public String toString() {

return "UserTest [username=" + username + ", password=" + password

+ "]";

}

}

1. 写出对象：

@Test

public void objectWriterTest() {

OutputStream outputStream = null;

BufferedOutputStream buf = null;

ObjectOutputStream obj = null;

try {

// 序列化文件輸出流

outputStream = new FileOutputStream("D:/test/test.txt");

// 构建缓冲流

buf = new BufferedOutputStream(outputStream);

// 构建字符输出的对象流

obj = new ObjectOutputStream(buf);

// 序列化数据写入

obj.writeObject(new UserTest("A", "123"));// Person对象

// 关闭流

obj.close();

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

1. 读入对象

@Test

public void objectReaderTest() {

try {

InputStream inputStream = new FileInputStream("D:/test/test.txt");

// 构建缓冲流

BufferedInputStream buf = new BufferedInputStream(inputStream);

// 构建字符输入的对象流

ObjectInputStream obj = new ObjectInputStream(buf);

UserTest tempPerson = (UserTest) obj.readObject();

System.out.println("UserTest对象为：" + tempPerson);

// 关闭流

obj.close();

buf.close();

inputStream.close();

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

注：使用对象流的一些注意事项  
1.读取顺序和写入顺序一定要一致，不然会读取出错。  
2.在对象属性前面加transient关键字，则该对象的属性不会被序列化。

**File：**

File类是对文件系统中文件以及文件夹进行封装的对象，可以通过对象的思想来操作文件和文件夹。File类保存文件或目录的各种元数据信息，包括文件名、文件长度、最后修改时间、是否可读、获取当前文件的路径名，判断指定文件是否存在、获得当前目录中的文件列表，创建、删除文件和目录等方法。

File类共提供了三个不同的构造函数，以不同的参数形式灵活地接收文件和目录名信息。

构造函数：

1）File (String pathname) 例:File f1=new File("FileTest1.txt"); //创建文件对象f1，f1所指的文件是在当前目录下创建的FileTest1.txt

2）File (String parent , String child) 例:File f2=new File(“D:\\dir1","FileTest2.txt") ;// 注意：D:\\dir1目录事先必须存在，否则异常；

3）File (File parent, String child) 例:File f4=new File("\\dir3"); File f5=new File(f4,"FileTest5.txt"); //在如果 \\dir3目录不存在使用f4.mkdir()先创建

一个对应于某磁盘文件或目录的File对象一经创建， 就可以通过调用它的方法来获得文件或目录的属性。

1）public boolean exists( ) 判断文件或目录是否存在

2）public boolean isFile( ) 判断是文件还是目录

3）public boolean isDirectory( ) 判断是文件还是目录

4）public String getName( ) 返回文件名或目录名

5）public String getPath( ) 返回文件或目录的路径。

6）public long length( ) 获取文件的长度

7）public String[ ] list ( ) 将目录中所有文件名保存在字符串数组中返回。

File类中还定义了一些对文件或目录进行管理、操作的方法，常用的方法有：

1） public boolean renameTo( File newFile ); 重命名文件

2） public void delete( ); 删除文件

3） public boolean mkdir( ); 创建目录

File类的全部方法：

boolean canRead() //接受文件名作为字符串

boolean canWrite() //检查能否写入指定文件

boolean equals(Object obj) //将指定对象与调用函数的对象进行比较

boolean exists() //测试文件是否存在

String getAbsolutePath() //返回此对象表示的文件的绝对路径名

String getName() //返回此对象表示的文件的名称

String getParent() //返回此File对象的路径名的上一级，如果路径名没有上一级，则返回null

boolean isAbsolute() //测试此File对象表示的文件是否是绝对路径名

boolean isDirectory() //测试此File对象表示的文件是否是目录

boolean isFile() //测试此File对象表示的文件是否是普通文件

boolean delete() //删除此对象指定的文件

boolean createNewFile() //创建空文件

boolean mkdir() //创建由该File对象表示的目录

boolean mkdirs() //创建包括父目录的目录

例子：

1. public class FileDemo1 {

2. public static void main(String[] args) {

3. File file = new File("D:" + File.separator + "test.txt");

4. if (file.exists()) {

5. file.delete();

6. } else {

7. try {

8. file.createNewFile();

9. } catch (IOException e) {

10. // TODO Auto-generated catch block

11. e.printStackTrace();

12. }

13. }

14. }

附注：System类对IO的支持

针对一些频繁的设备交互，Java语言系统预定了3个可以直接使用的流对象，分别是：

System.in（标准输入），通常代表键盘输入。

· System.out（标准输出）：通常写往显示器。

· System.err（标准错误输出）：通常写往显示器。

标准I/O：

Java程序可通过命令行参数与外界进行简短的信息交换，同时，也规定了与标准输入、输出设备，如键盘、显示器进行信息交换的方式。而通过文件可以与外界进行任意数据形式的信息交换。

1. 命令行参数：

public class TestArgs {

public static void main(String[] args) {

for (int i = 0; i < args.length; i++) {

System.out.println("args[" + i + "] is <" + args[i] + ">");

}

}

}

运行命令：java Java C VB

运行结果：

args[0] is <Java>

args[1] is <C>

args[2] is <VB>

2. 标准输入，输出数据流：

java系统自带的标准数据流：java.lang.System:

java.lang.System

public final class System extends Object{

static PrintStream err;//标准错误流（输出）

static InputStream in;//标准输入(键盘输入流)

static PrintStream out;//标准输出流(显示器输出流)

}

注意：

（1）System类不能创建对象，只能直接使用它的三个静态成员。

（2）每当main方法被执行时,就自动生成上述三个对象。

1) 标准输出流 System.out

System.out向标准输出设备输出数据，其数据类型为PrintStream。方法：

Void print(参数)

Void println(参数)

2)标准输入流 System.in

System.in读取标准输入设备数据（从标准输入获取数据，一般是键盘），其数据类型为InputStream。方法：

int read() //返回ASCII码。若,返回值=-1，说明没有读取到任何字节读取工作结束。

int read(byte[] b)//读入多个字节到缓冲区b中返回值是读入的字节数

例：

import java.io.\*;

public class StandardInputOutput {

public static void main(String args[]) {

int b;

try {

System.out.println("please Input:");

while ((b = System.in.read()) != -1) {

System.out.print((char) b);

}

} catch (IOException e) {

System.out.println(e.toString());

}

}

}

等待键盘输入，键盘输入什么，就打印出什么：

3)标准错误流

System.err输出标准错误，其数据类型为PrintStream。可查阅API获得详细说明。

标准输出通过System.out调用println方法输出参数并换行，而print方法输出参数但不换行。println或print方法都通过重载实现了输出基本数据类型的多个方法，包括输出参数类型为boolean、char、int、long、float和double。同时，也重载实现 了输出参数类型为char[]、String和Object的方法。其中，print（Object）和println（Object）方法在运行时将调用参数Object的toString方法。

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

public class StandardInputOutput {

public static void main(String args[]) {

String s;

// 创建缓冲区阅读器从键盘逐行读入数据

InputStreamReader ir = new InputStreamReader(System.in);

BufferedReader in = new BufferedReader(ir);

System.out.println("Unix系统: ctrl-d 或 ctrl-c 退出"

+ "\nWindows系统: ctrl-z 退出");

try {

// 读一行数据，并标准输出至显示器

s = in.readLine();

// readLine()方法运行时若发生I/O错误，将抛出IOException异常

while (s != null) {

System.out.println("Read: " + s);

s = in.readLine();

}

// 关闭缓冲阅读器

in.close();

} catch (IOException e) { // Catch any IO exceptions.

e.printStackTrace();

}

}

}

附加：

IOException异常类的子类

1.public class EOFException ：非正常到达文件尾或输入流尾时，抛出这种类型的异常。

2.public class FileNotFoundException： 当文件找不到时，抛出的异常。

3.public class InterruptedIOException： 当I/O操作被中断时，抛出这种类型的异常。

**附注：Transient的使用：**

我们都知道一个对象只要实现了Serilizable接口，这个对象就可以被序列化，java的这种序列化模式为开发者提供了很多便利，我们可以不必关系具体序列化的过程，只要这个类实现了Serilizable接口，这个类的所有属性和方法都会自动序列化。

      然而在实际开发过程中，我们常常会遇到这样的问题，这个类的有些属性需要序列化，而其他属性不需要被序列化，打个比方，如果一个用户有一些敏感信息（如密码，银行卡号等），为了安全起见，不希望在网络操作（主要涉及到序列化操作，本地序列化缓存也适用）中被传输，这些信息对应的变量就可以加上transient关键字。换句话说，这个字段的生命周期仅存于调用者的内存中而不会写到磁盘里持久化。

总之，java 的transient关键字为我们提供了便利，你只需要实现Serilizable接口，将不需要序列化的属性前添加关键字transient，序列化对象的时候，这个属性就不会序列化到指定的目的地中。

运行结果：

**read before Serializable:**

**password:123456**

**username:lipanpan**

**read after serialiable:**

**username:lipanpan**

**password:null**

1）一旦变量被transient修饰，变量将不再是对象持久化的一部分，该变量内容在序列化后无法获得访问。

2）transient关键字只能修饰变量，而不能修饰方法和类。注意，本地变量是不能被transient关键字修饰的。变量如果是用户自定义类变量，则该类需要实现Serializable接口。

3）被transient关键字修饰的变量不再能被序列化，一个静态变量不管是否被transient修饰，均不能被序列化。

第三点可能有些人很迷惑，因为发现在User类中的username字段前加上static关键字后，程序运行结果依然不变，即static类型的username也读出来为“Alexia”了，这不与第三点说的矛盾吗？实际上是这样的：第三点确实没错（一个静态变量不管是否被transient修饰，均不能被序列化），反序列化后类中static型变量username的值为当前JVM中对应static变量的值，这个值是JVM中的不是反序列化得出的