# Java IO之InputStream和OutputStream

2017.01.15

InputStream和OutStream是java.io封包中面向位元組操作的兩個頂層抽象類別，所有關於java同步IO位元組串流的操作都是基於這兩個的。

## 什麼是串流？

《O'Reilly-Java Io》中是這麼解釋的：

A stream is an ordered sequence of bytes of undetermined length. Input streams move bytes

of data into a Java program from some generally external source. Output streams move bytes

of data from Java to some generally external target. (In special cases streams can also move

bytes from one part of a Java program to another.)

串流是一個不確定長度的有序位元組序列。輸入串流從外部資源將資料位元組移動到Java程式中。輸出串流從Java程式中將資料位元組移動到外部目標。(特殊的情況也可以將位元組從java程式中一部分移動到另一部分)

## 串流從哪裡來？

通常串流來自於：

1. 網路
2. 文件
3. java內部程式

## InputStream(輸入串流)

InputStream作為java中用於讀取串流中位元組的頂層抽象類別，定義一些方法：

public abstract int read() throws IOException

public int read(byte b[], int off, int len) throws IOException

public int read(byte b[]) throws IOException{

return read(b, 0, b.length);

}

有三個read方法用來讀取位元組：

1. 第一個抽象方法交由子類別實現，讀取一個無符號位元組(unsigned byte)，由於java本身沒有無符號位元組的基本類型，所以用 int 作為返回值。當返回 -1 時表示到串流的結尾，這是需要返回int的原因之一(因為帶符號的byte有可能是-1)。
2. **作用：**從串流中讀取位元組陣列，通常一個一個位元組的讀取效率相當低，可指定陣列中開始的偏移位置off和長度len。

**參數：**保存位元組的位元組陣列、偏移量、長度。

**返回值：**實際讀到的位元組數(-1為末尾)

預設的實現依靠第一個抽象方法，迴圈呼叫讀取一個無符號位元組。所以效率不高，通常會用子類別更高效的方式重寫。

1. **作用：**從串流中讀取位元組陣列，通常一個一個位元組的讀取效率相當低。

**參數：**保存位元組的位元組陣列。

**返回值：**實際讀到的位元組數(-1為末尾)。

預設的實現依賴於第二個方法，僅僅是用read(b,0,b.length)實現，重寫這個方法的子類別相對較少。

public int available() throws IOException

**作用：**從串流中立刻能夠獲取到的位元組數。

**返回值：**能夠讀取到的位元組數(沒有會返回0，到串流的末尾也會返回0)

public long skip(long bytesToSkip) throws IOException

**作用：**從串流中跳過一定位元組不讀，通常跳過比讀取後不處理快(比如檔案串流，只是指標的移動)。

**參數：**期望跳過的位元組數

**返回值：**實際跳過的位元組數(遇到末尾返回-1)

public void close() throws IOException

**作用**：用完串流後，關閉串流，但並不是所有的串流都需要關閉，比如說System.in。

關閉串流的最佳實踐：

1. 在finally中關閉。

InputStream in = null;

try{

in = new ...;

...

} catch(IOException e){

...

} finally{

try{

if (in != null){

in.close();

}

} catch(IOException e){

}

}

1. java7支援try-with-resources方式關閉串流，實現了java.lang.AutoCloseable介面的物件支援用這種方式。

try (InputStream in = new ...){

...

} catch(IOException e){

...

}

標記和重置：

public synchronized void mark(int readLimit)

public synchronized void reset() throws IOException

public boolean markSupported()

有時讀取一些位元組後希望返回到之前的位置重新讀取，這幾個方法就是用來做這件事。

public boolean markSupported()

**作用：**判斷當前串流是否支援標記

**返回值：**當前串流是否支援標記

如果當前串流不支援標記，執行reset()方法將拋出一個IOException異常，而mark()方法不會做任何操作。

public synchronized void mark(int readLimit)

**作用：**將讀取的位置標記在當前位置。

**參數：**最大可閱讀超過標記位置的位元組數(只要沒有閱讀超過readLimit位元組數，就可以重置回去)。

在同一時刻，只能有一個標記，再設定會覆蓋

java.io中只有BufferedInputStream和ByteArrayInputStream支援標記。但其他的過濾串流連接到這兩個也支援標記。

## OutputStream(輸出串流)

OutputStream作為java中用於向串流中寫入位元組的頂層抽象類別，定義一些方法：

public abstract void write(int b) throws IOException

public void write(byte[] data, int offset, int length) throws IOException

public void write(byte[] data) throws IOException

有三個write方法用來向串流寫入位元組：

1. 第一個是抽象方法交由子類別實現，向串流中寫入一個無符號位元組(0-255)，如果超過255只會取低八位元的位元組。
2. 作用：向串流中寫入位元組陣列，可以指定陣列中起始的偏移位置和長度。

**參數：**寫入的字元陣列、起始的偏移位置、長度預設的實現是迴圈呼叫第一個方法一個一個寫入，但效率極其低下，子類別一般會有更高效的方式。

1. **作用：**向串流中寫入位元組陣列。

**參數：**寫入的字元陣列。

預設的實現是呼叫第二個write()方法write(b, 0, b.length);

public void flush() throws IOException

**作用：**許多輸出串流增加一個緩衝區來提升性能，當緩衝區寫滿後才會發送資料，而flush()方法可以強制清空緩衝區發送資料，如果不刷新則有可能導致資料丟失。通常PrintStream的println()方法會自動flush()。

public void close() throws IOException

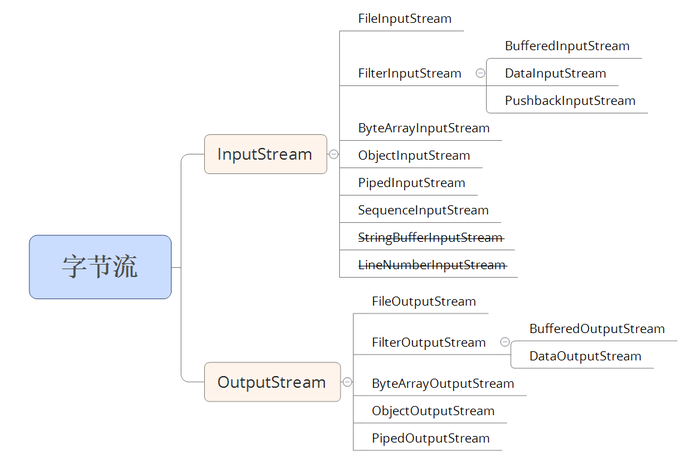
作用：當用完輸出串流之後，將串流關閉，關閉的同時也會flush()。

## 無處不在的IOException

IOException是受檢異常，在程式中必須被宣告。IOException存在的意義是因為作業系統中的"輸入"和"輸出"不可靠，作業系統層級發生的異常不受程式控制。

## 子類別

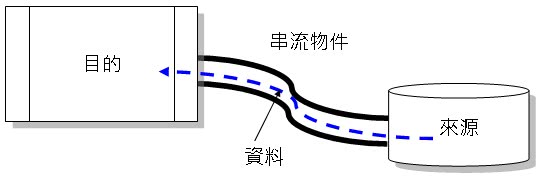
java.io封包中常用的子類別：



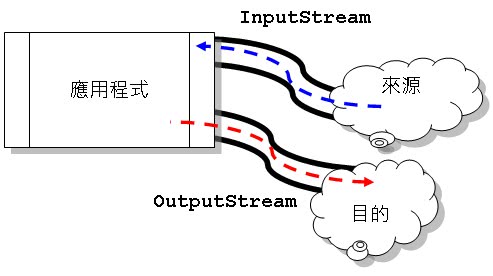
其它封包中還有一些類別。

# InputStream、OutputStream

Java將輸入/輸出(Input/Output)抽象化串流的概念，資料有其來源及目的地，銜接兩者的是串流物件。以比喻的方式來說，資料就好比水，藉由水管的銜接，由一端流向另一端。



從應用程式的角度來看，如果要將資料從來源取出，可以使用**輸入串流**，如果要將資料寫入目的地，則可以使用**輸出串流**。在Java中，輸入串流的代表物件為**InputStream**，輸出串流的代表物件為**OutputStream**。無論資料來源或目的地為何，只要設法取得InputStream或OutputStream的實例，接下來操作輸入輸出的方式都是一致，而無需理會來源或目的地的真正形式。



舉個例，可以設計一個通用的dump()方法：

public static void dump(InputStream input,

OutputStream output,

int dataLength) throws Exception{

byte[] data = new byte[dataLength];

int length = -1;

while((length = input.read(data))!= -1){

output.write(data, 0, length);

}

input.close();

output.close();

}

這個方法沒有限定來源或目的地真實形式，是依賴抽象的InputStream、OutputStream。如果要將某個檔案dump為另一個檔案，則可以這麼使用它：

dump(new FileInputStream(args[0]), new FileOutputStream(args[1]), 1024);

如果要指定下載某個網路上的檔案，則可以這麼使用它：

dump((new URL(args[0])).openStream(), new FileOutputStream(args[1]), 1024);

無論來源或目的地實體形式為何，只要想辦法取得InputStream或OutputStream，接下來其實都是操作InputStream或OutputStream的形式。例如以下是個使用ServerSocket接受客戶端連線的例子：

ServerSocket server = null;

Socket client = null;

try{

server = new ServerSocket(port);

while(true){

client = server.accept();

InputStream input = client.getInputStream();

OutputStream output = client.getOutputStream();

//接下來就是操作 InputStream、OutputStream 實例了...

...

}

}

catch(IOException ex){

...

}

一個使用Servlet讀取一個檔案並輸出至瀏覽器的例子則是如下：

response.setContentType("application/pdf");

InputStream in = this.getServletContext()

.getResourceAsStream("/WEB-INF/jdbc.pdf");

OutputStream out = response.getOutputStream();

byte[] data = new byte[1024];

int length = -1;

while((length = in.read(data))!= -1){

out.write(data, 0, length);

}

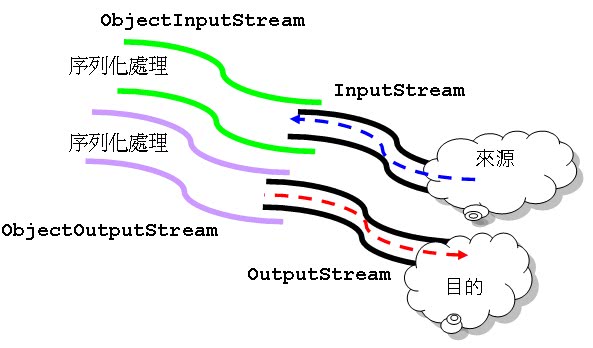
in.close();

out.close();

InputStream、OutputStream所提供的是基本的操作，如果想要為輸入輸出的資料作加工處理，則可以使用其一些子類別，這些子類別本身在建構時，都可以接受InputStream、OutputStream實例，例如具備緩衝區作用的 [BufferedInputStream、 BufferedOutputStream](https://openhome.cc/Gossip/JavaGossip-V2/BufferedInOutStream.htm)，具備資料轉換處理作用的 [DataInputStream、 DataOutputStream](https://openhome.cc/Gossip/JavaGossip-V2/DataInOutStream.htm)，具備物件序列化能力的 [ObjectInputStream、 ObjectOutputStream](https://openhome.cc/Gossip/JavaGossip-V2/ObjectInOutStream.htm) 等。

InputStream、OutputStream 與以上所提及的一些子類別(當然還有別的)，實現了設計模式中的 [Decorator 模式](https://openhome.cc/Gossip/DesignPattern/DecoratorPattern.htm)。無論是 [BufferedInputStream、 BufferedOutputStream](https://openhome.cc/Gossip/JavaGossip-V2/BufferedInOutStream.htm)、 [DataInputStream、 DataOutputStream](https://openhome.cc/Gossip/JavaGossip-V2/DataInOutStream.htm)、 [ObjectInputStream、 ObjectOutputStream](https://openhome.cc/Gossip/JavaGossip-V2/ObjectInOutStream.htm) 等，其本身都沒有改變 InputStream、OutputStream 的行為，只不過在 InputStream 取得資料之後，再作一些加工處理，或者是要輸出時作一些加工處理，再交由 OutputStream 真正進行輸出。

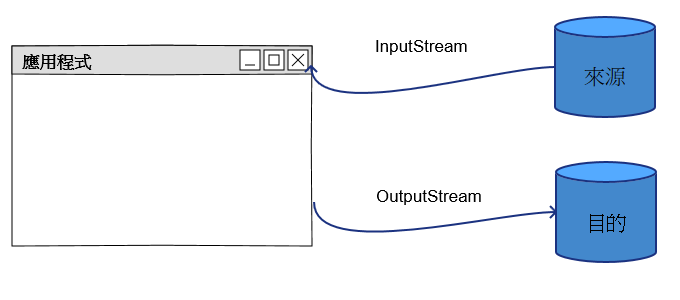
這有點像是小水管接上大水管：



只要瞭解InputStream、OutputStream抽象了資料來源與目的地的概念，以及瞭解InputStream、OutputStream及其子類別實現了[Decorator模式](https://openhome.cc/Gossip/DesignPattern/DecoratorPattern.htm)，無論實體來源時目的地為何(網路？資料庫？檔案？)，無論打算對資料作何種加工處理(套上哪個水管？甚至套接多個加工處理的水管？)，就不再會被java.io套件中眾多的API給搞混。

# Java Inputstream與Outputstream

1. 首先要瞭解Java如何以串流來抽象化輸入與輸出的概念，及Inputstream、Outputstream的架構，這樣一來無論是標準的輸出入、文檔的輸出入、網路的輸出入都可以用一致的操作來處理。
2. 串流設計概念從應用程式的角度來看如果要將來源資料讀出，可用輸入串流將資料導到程式中，而若要將資料寫入目的端則可以使用輸出串流，就如下圖所示：

[](http://3.bp.blogspot.com/-B7uuc20zWh4/VKUeM1GmAXI/AAAAAAAAAlE/f6cVjs7Mz9Q/s1600/IO.png)

1. 有需求來源與目的都不知道，可以設計一個通用的writeStreamingTo方法，這個方法接收InputStream與OutputStream，將資料以Inputstream流入應用程式，再以Outputstream寫出檔案。

/\*\*

\* 將來源資料利用輸入串流取出到應用程式，再利用輸出串流將檔案寫到目的，其中利用buffersize來控制緩衝區大小，並掌握進度。

\* @param in 輸入串流

\* @param out 輸出串流

\* @param bufferSize 緩衝區大小

\*/

private void writeStreamingTo(InputStream in,OutputStream out,int bufferSize){//待修改

try{

byte[] buffer = new byte[bufferSize];

int len = 0;

int totalBytes = in.available();

int writedBytes = 0;

System.out.println("Start Download...");

while ((len = in.read(buffer))> 0){

writedBytes += len;

out.write(buffer,0,len);

System.out.println(String.format("%.2f%s",writedBytes / (double)totalBytes \* 100,"%"));

}

in.close();

out.close();

} catch (IOException e){

e.printStackTrace();

}

}

1. Inputstream、Outputstream所提供的都是基本操作，如果想要為輸入輸出作加工處理，則可使用一些子類別，接下來將會介紹幾個常用的類別作為例，但本質沒有改變，最後還是透過Inputstream、Outputstream進行真正的輸出入，故瞭解這兩個比較重要。
2. 假設將某個檔案讀取並存到另一個檔案，則可以這麼使用。

FileInputStream in = new FileInputStream("來源路徑");

FileOutputStream out = new FileOutputStream ("目的路徑");

writeStreamingTo(in,out,1024);

1. 假設要從HTTP伺服器讀取某個網頁並另存新檔，那麼則可以考慮以下做法。

URL url = new URL("路徑");

InputStream in = url.openStream();

OutputStream out= new FileOutputStream("目的路徑");

writeStreamingTo(in,out,1024);

1. 關於第5點使用FileInputStream與FileOutputStream，但檔案存取時，磁碟存取的速度遠低於記憶體，因此為了減少對磁碟的存取，可以一次讀取一定長度的資料，寫入時也是一次寫入一定長度的資料，這樣可以增加資料存取的效率，緩衝區的概念就應運而生，因此可以將第5點改寫，使用BufferedInputStream與BufferedOutputStream，因此將檔案資料流入應用程式時，是向緩衝區拿取資料，而非對來源直接作存取，當緩衝區沒資料時會自動去來源獲取，輸出時也是相同概念，並未直接將資料寫入磁碟，而是先寫入緩衝區，在由緩衝區放滿一定大小後再寫至磁碟。

byte[] data = new byte[1024];

BufferedInputStream in = new BufferedInputStream(new FileInputStream("myTest.txt"));

BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream("myTest1.txt"));

while(in .read(data)!= -1){

bufferedOutputStream.write(data);

}

//將緩衝區中的資料全部寫出

bufferedOutputStream.flush();

# java的InputStream和OutputStream的理解

1. 在java中stream代表一種資料流程(源)，javaio的底層資料元，－(想像成水龍頭)。
2. 任何有能力產生資料流程(源)的javaio物件就可以看作是一個InputStream物件。既然它能產生出資料，就可以將資料取出，java對封裝的通用方法就read()方法－(出水龍頭)。
3. 任何有能力接收資料來源(串流)的javaio物件可以看作是一個OutputStream物件。同樣，它能接收資料，就可以呼叫它的write方法，讓它接收資料－－(進水龍頭了，呵呵)。
4. 可以在Inputstream和OutputStream資料來源的基礎上，從實際需要觸發，來重新封裝出不同性能機制的輸入、輸出串流，java.io封包中提供很豐富的輸入、輸出串流物件，如：
   1. 基於位元組串流的stream：

DataOutputStream----DataInputStream:

FileOutputStream-----FileInputStream:

...等，可以用InputStream和OutputStream從JDK文檔查閱。

* 1. 基於字元串流的stream(典型的以write和reader來標識的)：

FileWriter---FileReader:

StringWriter---StringReader:

...等，可以用Writer和Reader從JDK文檔裡頭查看說明

stream應該是水龍頭裡的水資源。

InputStream:是一個出水龍頭(把水封裝在裡頭)的一個實物物件，該物件的read方法呢，就想成這個出水龍頭這一機制物件的開關鈕，read或openStream(其他物件包容InputStream物件的物件方法)一下呢，就等於打開了出水龍頭的按鈕，水就出來，裡頭封裝的水是什麼性質的呢，就用相應的容器來裝，如string或byte[]...

OutputStream:在InputStream基礎上反著想就ok。

### OutputStream

1. 輸出資料

void write(int b)往串流中寫一個位元組b。

void write(byte b[])往串流中寫一個位元組陣列b。

void write(byte b[],int off,int len)把位元組陣列b中從下標off開始，長度為len的位元組寫入串流中。

1. flush()刷空輸出串流，並輸出所有被緩衝區的位元組由於某些串流支持緩衝區功能，該方法將把緩衝區中所有內容強制輸出到串流中。
2. 關閉串流

close()串流操作完畢後必須關閉。

### InputStream

1. 從串流中讀取資料

int read()讀取一個位元組，返回值為所讀得位元組

int read(byte b[])讀取多個位元組，放置到位元組陣列b中，通常讀取的位元組數量為b的長度，返回值為實際獨取的位元組的數量。

int read(byte b[] ,int off,int len)讀取len個位元組，放置到以下標off開始位元組陣列b中，返回值為實際讀取的位元組的數量。

int available()返回值為串流中尚未讀取的位元組的數量。

long skip(long n);讀取指標跳過n個位元組不讀，返回值為實際跳過的位元組數量

1. 關閉串流

close()串流操作完畢後必須關閉

1. 使用輸入串流中的標記。

void mark(int readlimit)記錄當前指標的所在位置，readlimit表示讀取指標讀出的readlimit個位元組後所標記的指標位置才實效。

void reset()把讀取指標重新指向用mark方法所記錄的位置boolean markSupported()當前的串流是否支援讀取指標的記錄功能。

# Java IO通過Stream(串流)來實現

關於串流，可以理解為是一種"資料的管道"。管道中流動的東西可以是基於位元組，也可以是基於字元的等。就好像管道裡面可以流動水，也可以流動石油一樣。

而對應於串流還有一個概念：輸入、輸出設備。這些設備可以是磁片檔、鍵碟(輸入裝置)、顯示器(輸出設備)、印表機(輸出設備)、網路通訊端等等。

## 何謂"串流"

Java中定義了兩種類型的串流：位元組型，和字元型。

位元組串流：處理位元組的輸入和輸出。包括讀寫二進位資料等方面的內容。

字元串流：處理字元的輸入和輸出。他採用的是Unicode編碼，可以實現國際化。使用字元串流的另外一個好處就是：字元串流比位元組串流更有效率。

## 位元組串流

位元組串流包含兩個頂層抽象類別：InputStream和OutputStream。

1. 位元組串流的兩個頂層類別是抽象類別，分別是：InputStream和OutputStream。
2. 每個抽象類別都有子類別來實現具體的功能，處理不同的設備的輸入和輸出。

下面簡單介紹位元組串流的幾個常用子類別：

|  |  |
| --- | --- |
| 位元組串流類別 | 功能簡單介紹 |
| DataInputStream | 包含讀取Java標準資料類型的輸入串流 |
| DataOutputStream | 包含寫入Java標準資料類型的輸出串流 |
| ByteArrayInputStream | 從位元組陣列讀取的輸入串流 |
| ByteArrayOutputStream | 寫入位元組陣列的輸出串流 |
| FileInputStream | 從檔案讀取的輸入串流 |
| FileOutputStream | 寫入檔案的輸出串流 |
| PrintStream | 包含最常見的Print()和Println()的輸出串流 |
| PushbackInputStream | 返回一個位元組到輸入串流，主要用於編譯器的實現 |
| PipedInputStream | 輸出管道 |
| PipedOutputStream | 輸入管道 |
| SequenceInputStream | 將n個輸入串流聯合起來，一個接一個按一定順序讀取 |
| RandomAccessFile | 隨機訪問檔案 |
| BufferInputStream | 緩衝輸入串流 |
| BufferOutputStream | 緩衝輸出串流 |
| FilterInputStream | 實現**InputStream** Interface |
| FilterOutputStream | 實現**OutputStream** Interface |
| InputStream | 抽象類別，描述串流的輸入 |
| OutputStream | 抽象類別，描述串流的輸出 |

抽象類別**InputStream**和OutpurStream定義實用的方法，其中最主要的是read()和write()。這兩個方法在**InputStream**和**OutputStream**中宣告為抽象方法，由子串流類別override實現。

每個抽象類別都有子類別實現具體的功能，處理不同的設備的輸入和輸出。

下面簡單介紹位元組串流的幾個常用子類別：

|  |  |
| --- | --- |
| 位元組串流類別 | 功能簡單介紹 |
| DataInputStream | 包含讀取Java標準資料類型的輸入串流 |
| DataOutputStream | 包含寫Java標準資料類型的輸出串流 |
| ByteArrayInputStream | 從位元組陣列讀取的輸入串流 |
| ByteArrayOutputStream | 寫入位元組陣列的輸出串流 |
| FileInputStream | 從檔讀取的輸入串流 |
| FileOutputStream | 寫入檔的輸出串流 |
| PrintStream | 包含最常見的Print()和Println()的輸出串流 |
| PushbackInputStream | 返回一個位元組到輸入串流，主要用於編譯器的實現 |
| PipedInputStream | 輸出管道 |
| PipedOutputStream | 輸入管道 |
| SequenceInputStream | 將n個輸入串流聯合起來，一個接一個按一定順序讀取 |
| RandomAccessFile | 隨機訪問檔 |
| BufferInputStream | 緩衝輸入串流 |
| BufferOutputStream | 緩衝輸出串流 |
| FilterInputStream | 實現**InputStream** Interface |
| FilterOutputStream | 實現**OutputStream** Interface |
| InputStream | 抽象類別，描述串流的輸入 |
| OutputStream | 抽象類別，描述串流的輸入 |

抽象類別**InputStream**和**OutpurStream**定義實用的方法，其中最主要的是**read()**和**write()**。這兩個方法在**InputStream**和**OutputStream**宣告為抽象方法，由子串流類別override實現。

# Java - Files and I/O

The java.io package contains nearly every class you might ever need to perform input and output (I/O)in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

## Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

**InPutStream** − The InputStream is used to read data from a source.

**OutPutStream** − The OutputStream is used for writing data to a destination.



Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

## Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream**and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −

## Example

import java.io.\*;

public class CopyFile{

public static void main(String args[]) throws IOException{

FileInputStream in = null;

FileOutputStream out = null;

try{

in = new FileInputStream("input.txt");

out = new FileOutputStream("output.txt");

int c;

while ((c = in.read())!= -1){

out.write(c);

}

}finally{

if (in != null){

in.close();

}

if (out != null){

out.close();

}

}

}

}

Now let's have a file **input.txt** with the following content − This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

$javac CopyFile.java

$java CopyFile

## Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters)into an output file −

### Example

import java.io.\*;

public class CopyFile{

public static void main(String args[]) throws IOException{

FileReader in = null;

FileWriter out = null;

try{

in = new FileReader("input.txt");

out = new FileWriter("output.txt");

int c;

while ((c = in.read())!= -1){

out.write(c);

}

}finally{

if (in != null){

in.close();

}

if (out != null){

out.close();

}

}

}

}

Now let's have a file **input.txt** with the following content − This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

$javac CopyFile.java

$java CopyFile

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

**Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.

**Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.

**Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "q" −

### Example

import java.io.\*;

public class ReadConsole{

public static void main(String args[]) throws IOException{

InputStreamReader cin = null;

try{

cin = new InputStreamReader(System.in);

System.out.println("Enter characters, 'q' to quit.");

char c;

do{

c = (char)cin.read();

System.out.print(c);

} while(c != 'q');

}finally{

if (cin != null){

cin.close();

}

}

}

}

Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q' −

$javac ReadConsole.java

$java ReadConsole

Enter characters, 'q' to quit.

1

1

e

e

q

q

## Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial.

## FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file −

InputStream f = new FileInputStream("C:/java/hello");

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File()method as follows −

File f = new File("C:/java/hello");

InputStream f = new FileInputStream(f);

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

|  |  |
| --- | --- |
| Sr.No. | Method & Description |
| 1 | public void close() throws IOException{}  This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException. |
| 2 | protected void finalize() throws IOException{}  This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException. |
| 3 | public int read(int r) throws IOException{}  This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file. |
| 4 | public int read(byte[] r) throws IOException{}  This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned. |
| 5 | public int available() throws IOException{}  Gives the number of bytes that can be read from this file input stream. Returns an int. |

There are other important input streams available, for more detail you can refer to the following links −

## FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file −

OutputStream f = new FileOutputStream("C:/java/hello")

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File()method as follows −

File f = new File("C:/java/hello");

OutputStream f = new FileOutputStream(f);

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

|  |  |
| --- | --- |
| Sr.No. | Method & Description |
| 1 | public void close() throws IOException{}  This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException. |
| 2 | protected void finalize() throws IOException{}  This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException. |
| 3 | public void write(int w) throws IOException{}  This methods writes the specified byte to the output stream. |
| 4 | public void write(byte[] w)  Writes w.length bytes from the mentioned byte array to the OutputStream. |

There are other important output streams available, for more detail you can refer to the following links −

### Example

Following is the example to demonstrate InputStream and OutputStream −

import java.io.\*;

public class fileStreamTest{

public static void main(String args[]){

try{

byte bWrite [] ={11,21,3,40,5};

OutputStream os = new FileOutputStream("test.txt");

for(int x = 0; x < bWrite.length; x++){

os.write(bWrite[x]);//writes the bytes

}

os.close();

InputStream is = new FileInputStream("test.txt");

int size = is.available();

for(int i = 0; i < size; i++){

System.out.print((char)is.read()+ " ");

}

is.close();

} catch (IOException e){

System.out.print("Exception");

}

}

}

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

## Directories in Java

A directory is a File which can contain a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on File object and what are related to directories.

## Creating Directories

There are two useful **File** utility methods, which can be used to create directories −

The **mkdir()**method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.

The **mkdirs()**method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory −

### Example

import java.io.File;

public class CreateDir{

public static void main(String args[]){

String dirname = "/tmp/user/java/bin";

File d = new File(dirname);

//Create directory now.

d.mkdirs();

}

}

Compile and execute the above code to create "/tmp/user/java/bin".

**Note** − Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/)on a Windows version of Java, the path will still resolve correctly.

## Listing Directories

You can use **list()**method provided by **File** object to list down all the files and directories available in a directory as follows −

### Example

import java.io.File;

public class ReadDir{

public static void main(String[] args){

File file = null;

String[] paths;

try{

//create new file object

file = new File("/tmp");

//array of files and directory

paths = file.list();

//for each name in the path array

for(String path:paths){

//prints filename and directory name

System.out.println(path);

}

} catch (Exception e){

//if any error occurs

e.printStackTrace();

}

}

}

This will produce the following result based on the directories and files available in your **/tmp** directory −

Output

test1.txt

test2.txt

ReadDir.java

ReadDir.class

# [java中outputStream與inputStream的相互轉換](http://www.cnblogs.com/vigarbuaa/archive/2013/01/13/2858859.html)

/\*做一個功能驗證，要用到inputStream與outputStream的轉換，本以為很簡單的東東

搞了蠻久，從"程式師 閆帆"處取得一段程式碼\*/

package com.boco.test;

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import java.io.InputStream;

import java.io.OutputStream;

public class ConvertUtil{

//inputStream轉outputStream

public ByteArrayOutputStream parse(InputStream in) throws Exception{

ByteArrayOutputStream swapStream = new ByteArrayOutputStream();

int ch;

while ((ch = in.read())!= -1){

swapStream.write(ch);

}

return swapStream;

}

//outputStream轉inputStream

public ByteArrayInputStream parse(OutputStream out) throws Exception{

ByteArrayOutputStream baos=new ByteArrayOutputStream();

baos=(ByteArrayOutputStream)out;

ByteArrayInputStream swapStream = new ByteArrayInputStream(baos.toByteArray());

return swapStream;

}

//inputStream轉String

public String parse\_String(InputStream in) throws Exception{

ByteArrayOutputStream swapStream = new ByteArrayOutputStream();

int ch;

while ((ch = in.read())!= -1){

swapStream.write(ch);

}

return swapStream.toString();

}

//OutputStream 轉String

public String parse\_String(OutputStream out) throws Exception{

ByteArrayOutputStream baos=new ByteArrayOutputStream();

baos=(ByteArrayOutputStream)out;

ByteArrayInputStream swapStream = new ByteArrayInputStream(baos.toByteArray());

return swapStream.toString();

}

//String轉inputStream

public ByteArrayInputStream parse\_inputStream(String in) throws Exception{

ByteArrayInputStream input=new ByteArrayInputStream(in.getBytes());

return input;

}

//String 轉outputStream

public ByteArrayOutputStream parse\_outputStream(String in) throws Exception{

return parse(parse\_inputStream(in));

}

}

Byte 傳輸 stream

String 傳輸 Reader, Writer

FileInputStream是做檔案位元輸入的類別

InputStreamReader是做字串輸入的類別

傳輸是以節點(Node)做為起點或終點，做出Stream這樣的動作

串流就會開啟，直到close()被呼叫為止

串流會有保護同時只開放一個程式使用，因此必須記得close()串流

要讓流過的串流再回頭看一次，要使用Buffer方式，把串流存起來，就像是蓄水槽一樣

串流使用方法

開檔

建立Stream物件

使用讀寫的方法

public class Test{

public static void main(String[] args){

FileInputStream fin;

try{

fin = new FileInputStream("c:\\file.txt");

while (fin.available()> 0)

System.out.print(fin.read());

} catch (IOException e){

e.printStackTrace();

}

}

}

FileInputStream以Byte為傳輸單位

中文佔用2個bytes，所以要用Byte[] 來做buffer，而且陣列大小必須是2的倍數

public class Test{

public static void main(String[] args){

FileInputStream fin;

try{

fin = new FileInputStream("c:\\file.txt");

byte buf[] = new byte[8];//建立一個大小8Byte的緩衝區

int bufSize;//用來計算讀取的資料大小

while (fin.available()> 0)

bufSize = fin.read(buf);//讀取資料到buf內

System.out.print(new String(buf, 0, bufSize));

} catch (IOException e){

e.printStackTrace();

}

}

}

中文字中間夾雜了1Byte大小的字元，讀取就會錯亂，這時候就沒有辦法再堅持使用FileInputStream

public class Test{

public static void main(String[] args){

FileReader fin;

try{

fin = new FileReader("c:\\file.txt");

int word;

while (fin.ready()){

word = fin.read();

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

}

} catch (IOException e){

e.printStackTrace();

}

}

}

資料型態與串流做個連結，稱作chain

為了達到串流連結，要用到一些的類別，也就是轉接頭，稱作Filter

Filter除了可以達到轉換型態的功能外，還有些具有緩衝的功能，也就是讓流去的串流可以被回溯

來看看JAVA提供那些Filter的類別

|  |  |
| --- | --- |
| 串流緩衝區 | BufferedInputStream  BufferedOutputStream  BufferedReader  BufferedWriter |
| 物件串流 | ObjectInputStream  ObjectOutputStream |
| 基本資料型態串流 | InputStreamReader  InputStreamWriter  DataInputStream  DataOutputStream |
| 計算行數 | LineNumberInputStream  LineNumberReader |
| 堆疊串流 | PushbackInputStream  PushbackReader |
| 列印串流 | PrintStream  PrintWriter |

public class Test{

public static void main(String[] args){

FileOutputStream fin;

try{

fin = new FileOutputStream("c:\\file.txt");

DataOutputStream dataWrite = new DataOutputStream(fin);

short sh[] ={1,2,3,4,5};

for(short i : sh){

dataWrite.writeShort(i);//這裡的方法要搭配資料型態使用

}

} catch (IOException e){

e.printStackTrace();

}

}

}

範例

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

/\*\*

 \* 測試類別

 \* @author hanlw

 \* 2012 -07 - 04

 \*/

public class Test\_two{

public static void main(String[] args) throws Exception{

/\*\*

\* InputStream與OutputStream的使用例子

\*

\* (緩衝檔案輸入串流)BufferedInputStream → (檔案輸入串流)FileInputStream → (輸入串流)java.io.InputStream

\*

\* (緩衝檔案輸出串流)BufferedOuputStream → (檔案輸出串流)FileOuputStream → (輸出串流)java.io.OutputStream

\*/

/\*\*

\* 1.通過串流複製一個圖片的例子

\*/

/\* File file = new File("c:/images/1.png");

File outfile = new File("C:/temp.png");

FileInputStream inputStream = new FileInputStream(file);

FileOutputStream outputStream = new FileOutputStream(outfile);

int i = 0;

while(i != -1){

i = inputStream.read();

outputStream.write(i);

}

//注意串流的關閉(★必須的)

inputStream.close();

outputStream.close();

\*/

/\*\*

\* 2.如果想提高要提高複製的速度，可以採用緩衝檔案輸入\輸出串流，如下：

\*/

/\* File file = new File("C:/images/1.png");

File outfile = new File("C:/temp1.jpg");

//檔案輸入串流

FileInputStream inputStream = new FileInputStream(file);

//檔案輸出串流

FileOutputStream outputStream = new FileOutputStream(outfile);

//緩衝檔案輸入串流

BufferedInputStream bufferedInputStream = new BufferedInputStream(inputStream);

//緩衝檔案輸出串流

BufferedOutputStream bufferedOutputStream = new BufferedOutputStream(outputStream);

int i = 0;

while(i != -1){

i = bufferedInputStream.read();

bufferedOutputStream.write(i);

}

//串流的關閉

bufferedOutputStream.flush();//強制清除緩衝區的內容

bufferedInputStream.close();

bufferedOutputStream.close();

\*/

/\*\*

\* 3.當檔案很大,要做一個緩衝處理來提高速度。如下：當檔案的大小大於512個位元組時，每次讀取512個位元組後再做處理

\*

\*/

/\* File file = new File("C:/images/1.png");

File outfile = new File("C:/temp2.png");

//檔案輸入串流

FileInputStream inputStream = new FileInputStream(file);

//檔案輸出串流

FileOutputStream outputStream = new FileOutputStream(outfile);

int i = 0;

//緩衝大小為512位元組

byte[] buffer = new byte[512];

while(true){

if(inputStream.available()< 512){

while(i != -1){

i = inputStream.read();

outputStream.write(i);

}

break;//注意此處不能忘記哦

}else{

//當檔案的大小大於512位元組時

inputStream.read(buffer);

outputStream.write(buffer);

}

}

//串流的關閉

//注意串流的關閉(★必須的)

inputStream.close();

outputStream.close();

\*/

/\*\*

\* 4.根據上面的例子，可以知道：可以做一個雙緩衝的檔案複製

\*/

File file = new File("C:/images/1.png");

File outfile = new File("C:/temp3.png");

//檔案輸入串流

FileInputStream inputStream = new FileInputStream(file);

//檔案輸出串流

FileOutputStream outputStream = new FileOutputStream(outfile);

//緩衝檔案輸入串流

BufferedInputStream bufferedInputStream = new BufferedInputStream(inputStream);

//緩衝檔案輸出串流

BufferedOutputStream bufferedOutputStream = new BufferedOutputStream(outputStream);

int i = 0;

//緩衝區的大小

byte[] buffer = new byte[512];

while(true){

if(bufferedInputStream.available()< 512){

while(i != -1){

i = bufferedInputStream.read();

bufferedOutputStream.write(i);

}

break;

}else{

//當檔案的大小還大於512位元組時

bufferedInputStream.read(buffer);

bufferedOutputStream.write(buffer);

}

}

//強制清空緩衝區的內容

bufferedOutputStream.flush();

//串流的關閉

bufferedInputStream.close();

bufferedOutputStream.close();

}

}