## JAVA流式输入/输出原理

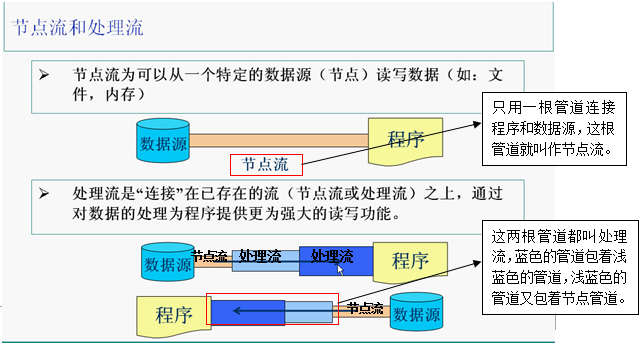
**流是用来读写数据的**，java有一个类叫File，它封装的是文件的文件名，只是内存里面的一个对象，**真正的文件是在硬盘上的一块空间**，在这个文件里面存放着各种各样的数据，我们想读文件里面的数据怎么办呢？是通过一个流的方式来读，咱们要想从程序读数据，对于计算机来说，**无论读什么类型的数据都是以010101101010**这样的形式读取的。

有的时候，**一根管道不够用**，比方说这根管道流过来的水有一些杂质，我们就可以在这个根管道的外面再包一层管道，把杂质给过滤掉。

从程序的角度来讲，从计算机读取到的**原始数据肯定都是010101**这种形式的，一个字节一个字节地往外读，你再在这根管道的外面**再包一层比较强大的管道**，这个管道可以把**010101帮你转换成字符串**。

JAVA里面的**流式输入/输出跟水流的原理**一模一样

**节点流和处理流**

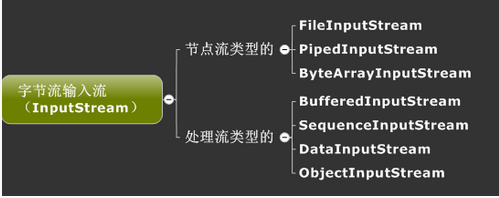


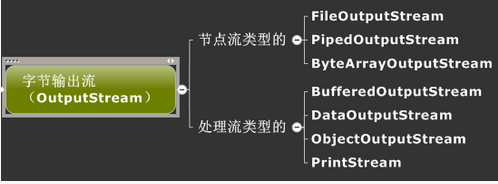
 　　你要是对原始的流不满意，你可以**在这根管道外面再套其它的管道**，**套在其它管道之上的流叫处理流。**为什么需要处理流呢？这就跟水流里面有杂质，你要过滤它，你可以再套一层管道过滤这些杂质一样。

# 节点流

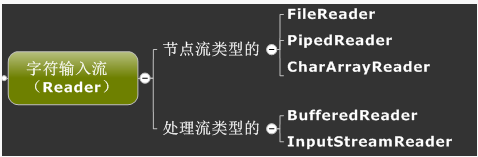
### 基本

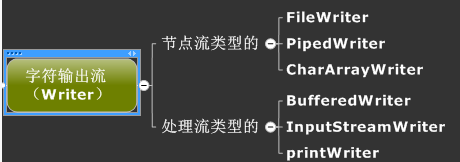
**1）字节流有两个抽象类：InputStream OutputStream其对应子类有FileInputStream和FileOutputStream实现文件读写。而BufferedInputStream和BufferedOutputStream提供缓冲区功能。**





2）字符流有两个抽象类：Writer Reader其对应子类FileWriter和FileReader可实现文件的读写操作.BufferedWriter和BufferedReader能够提供缓冲区功能，用以提高效率。





**节点流：直接读；处理流：外套读写。**

**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。  
（3）字节流在操作的时候本身是不会用到缓冲区的，是文件本身的直接操作的；而字符流在操作的时候下后是会用到缓冲区的，是通过缓冲区来操作文件，我们将在下面验证这一点。**

**结论：优先选用字节流。首先因为硬盘上的所有文件都是以字节的形式进行传输或者保存的，包括图片等内容。但是字符只是在内存中才会形成的，所以在开发中，字节流使用广泛。**

### 按字节流读文件:InputStream

此抽象类是表示字节输入流的所有类的超类。需要定义 InputStream 的子类的应用程序必须始终提供返回下一个输入字节的方法。

|  |  |
| --- | --- |
| 1 | **public void close() throws IOException{}** 关闭此文件输入流并释放与此流有关的所有系统资源。抛出IOException异常。 |
| 2 | **protected void finalize()throws IOException {}** 这个方法清除与该文件的连接。确保在不再引用文件输入流时调用其 close 方法。抛出IOException异常。 |
| 3 | **public int read(int r)throws IOException{}** 这个方法从 InputStream 对象读取指定字节的数据。返回为整数值()ASCii。返回下一字节数据，如果已经到结尾则返回-1。（要把数字准换为字符（char）temp） |
| 4 | **public int read(byte[]** buffer**) throws IOException{}**  **先把读取到的数据填满这个byte[]类型的数组buffer(buffer是内存里面的一块缓冲区)，然后再处理数组里面的数据。** |
| 5 | **public int available() throws IOException{}** 返回下一次对此输入流调用的方法可以不受阻塞地从此输入流读取的字节数。返回一个整数值。 |

**File f=new File（ "D:" + File.separator + "test.txt"）；//File找到文件**

**InputStream is =  new FileInputStream(f);**

byte b[] = new byte[1024];  byte b[] = new byte[(int)f.length()] ; // 字节数组大小由文件决定

int len = input.read(b) ; // 读取内容,read()返回值为写入多少个数

循环读取//for(int i=0;i<b.length;i++){

b[i] = (byte)input.read() ;

判断读取//

byte[] bys = new byte[1024];

int len = 0;

while ((len = fis.read(bys)) != -1) {

System.out.print(new String(bys, 0, len)); }

input.close() ; // 关闭输出流

System.out.println("内容为：" + new String(b,0,len)) ; // 把byte数组变为字符串输出

分别为read()，read(byte[] b),read(byte[] b, int off, int len)。其中read()方法是一次读取一个字节，**鬼都知道效率是非常低的。所以最好是使用后面两个方法。**

有两个构造方法可以用来创建 FileOutputStream 对象。

使用字符串类型的文件名来创建一个输出流对象：

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

也可以使用一个文件对象来创建一个输出流来写文件。我们首先得使用File()方法来创建一个文件对象：

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

**这样我们就完成了把a.txt复制了一份并命名为b.txt的文件了**

public class CopyFileDemo {

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

// 封装数据源

InputStream fis = new FileInputStream("a.txt");

OutputStream fos = new FileOutputStream("b.txt");

// 复制数据

byte[] bys = new byte[1024];

int len = 0;

while ((len = fis.read(bys)) != -1) {

fos.write(bys, 0, len);}

// 释放资源

fos.close();

fis.close();

}

}

### 按字节流写文件:OutputStream

|  |  |
| --- | --- |
| 1 | **public void close() throws IOException{}** 关闭此文件输入流并释放与此流有关的所有系统资源。抛出IOException异常。 |
| 2 | **protected void finalize()throws IOException {}** 这个方法清除与该文件的连接。确保在不再引用文件输入流时调用其 close 方法。抛出IOException异常。 |

**输出流接受输出字节并将这些字节发送到某个接收器。**

|  |  |
| --- | --- |
| void | **write**(byte[] b)           将 b.length 个字节从指定的 byte 数组写入此输出流。 |
| void | **write**(byte[] b, int off, int len)           将指定 byte 数组中从偏移量 off 开始的 len 个字节写入此输出流。 |
| abstract void | **write**(int b)           将指定的字节写入此输出流。 |

public class TestDemo {

public static void main(String[] args) throws Exception {// 此处直接抛出

// 1、定义输出的文件的路径

File file = new File("F:" + File.separator + "demo.text");

// 2、此时由于目录不存在，所以文件不能够输出，那么要先创建文件的目录

if (!file.getParentFile().exists()) { // 文件目录不存在

file.getParentFile().mkdirs(); // 创建目录}

// 3、应该使用OutputStream和其子类进行对象的实例化,此时目录存在，文件还不存在

OutputStream output = new FileOutputStream(file);

// 4、进行文件内容的输出

String str = "今天开始，感谢时间 ！";\r\n增加换行

byte data[] = str.getBytes(); // 将字符串变为字节数组

output.write(data); //将内容输出

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

output.write(data[i]); //将内容输出、

**byte**[] buf = **new** **byte**[1024];

**int** len = 0;

**while**((len=fis.read(buf))!=-1)  {

                        fos.write(buf,0,len); } }

//5、资源操作的最后一定要进行关闭

output.close();

* 让流对象变成垃圾，这样就可以被垃圾回收器回收了
* 通知系统去释放跟该文件相关的资源

[**FileOutputStream**](https://blog.csdn.net/u010913699/article/details/9227803)**(**[**String**](https://blog.csdn.net/u010913699/article/details/9227803)**name, boolean append)**  
          创建一个向具有指定 name 的文件中写入数据的输出文件流。

# 处理流：

### 转换流

**OutputStreamWriter（Writer子类）**的方法： 输出字符流变为字节流,写入文件：

**lnputStreamReader（Reader子类）**的方法： 文件的**字节流变为字符流：**

Reader reader=new InputStreamReader(new FileInputStream("StringDemo.java"));

* public void write(int c):写一个字符
* public void write(char[] cbuf):写一个字符数组
* public void write(char[] cbuf,int off,int len):写一个字符数组的一部分
* public void write(String str):写一个字符串
* public void write(String str,int off,int len):写一个字符串的一部分

OutputStreamWriter osw =newOutputStreamWriter(new FileOutputStream("osw.txt"), "UTF-8");

### 缓冲流

缓冲流也包含了四个类：**BufferedInputStream、BufferedOutputStream、BufferedReader和BufferedWriter。**流都是成对的，没有流是是不成对的，肯定是一个in，一个out。

BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("bos.txt"));

带有缓冲区的，缓冲区(Buffer)就是内存里面的一小块区域，读写数据时都是先把数据放到这块缓冲区域里面，减少io对硬盘的访问次数，保护我们的硬盘。可以把缓冲区想象成一个小桶，把要读写的数据想象成水，每次读取数据或者是写入数据之前，都是先把数据装到这个桶里面，装满了以后再做处理。这就是所谓的缓冲。先把数据放置到缓冲区上，等到缓冲区满了以后，再一次把缓冲区里面的数据写入到硬盘上或者读取出来，这样可以**有效地减少对硬盘的访问次数**，有利于保护我们的硬盘。

### **内存操作流**

**(内存操作流一般用于处理临时信息，因为临时信息不需要保存，使用后就可以删除)**

* 操作字节数组 (英文)
  + ByteArrayInputStream
  + ByteArrayOutputStream

ByteArrayInputStream bais = new ByteArrayInputStream(str.getBytes());

* 操作字符数组 （中文）
  + CharArrayReader
  + CharArrayWrite
* 操作字符串
  + StringReader
  + StringWriter

### 打印流

打印流分为两类

* 字节流打印流 PrintStream
* 字符打印流 PrintWriter

**打印流的特点：**

只有写数据的，没有读取数据。**只能操作目的地**，不能操作数据源。

可以操作任意类型的数据。

如果启动了自动刷新，能够自动刷新。

该流是可以直接操作文本文件的。

我们用打印流来复制一个文件,我们先来分析

* 需求：1.txt复制到Copy.txt中(首先你要确认有1.txt，并有内容)
* 数据源：1.txt – 读取数据 –> FileReader –高效，所以用–> BufferedReader
* 目的地：Copy.txt – 写出数据 –> FileWriter –> BufferedWriter –> PrintWriter

public class CopyFileDemo {

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

// 封装数据源

BufferedReader br = new BufferedReader(new FileReader(

"DataStreamDemo.java"));

// 封装目的地

PrintWriter pw = new PrintWriter(new FileWriter("Copy.java"), true);

String line = null;

while((line=br.readLine())!=null){

pw.println(line);// println()方法，其实等价于： bw.write(); bw.newLine(); bw.flush();三个方法

}

pw.close();

br.close();

}

}

### 随机访问流

**(RandomAccessFile类不属于流，是Object类的子类。但它融合了InputStream和OutputStream的功能。支持对随机访问文件的读取和写入。)**

public RandomAccessFile(String name,String mode)：   
第一个参数是文件路径，第二个参数是操作文件的模式。   
模式有四种，我们最常用的一种叫”rw”,这种方式表示我既可以写数据，也可以读取数据 public class RandomAccessFileDemo {

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

// 创建随机访问流对象

RandomAccessFile raf = new RandomAccessFile("raf.txt", "rw");

int i = raf.readInt();

System.out.println(i);

// 该文件指针可以通过 getFilePointer方法读取，并通过 seek 方法设置。

System.out.println("当前文件的指针位置是：" + raf.getFilePointer());

char ch = raf.readChar();

System.out.println(ch);

System.out.println("当前文件的指针位置是：" + raf.getFilePointer());

String s = raf.readUTF();

System.out.println(s);

System.out.println("当前文件的指针位置是：" + raf.getFilePointer());

// 我就要读取a，怎么办呢?

raf.seek(4);

ch = raf.readChar();

System.out.println(ch);

}

### 合并流

以前我们只是将一个文件的内容复制到另外一个文件中，现在这个合并流可以实现将两个及多个文件的内容复制到一个文件中了。

public class SequenceInputStreamDemo {

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

InputStream s1 = new FileInputStream("ByteArrayStreamDemo.java");

InputStream s2 = new FileInputStream("DataStreamDemo.java");

**SequenceInputStream** sis = new SequenceInputStream(s1, s2);

BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("Copy.java"));

// 读写操作

byte[] bys = new byte[1024];

int len = 0;

while ((len = sis.read(bys)) != -1) {

bos.write(bys, 0, len);

}

bos.close();

sis.close();

}

}

那么怎么合并3个或者三个以上的文件操作呢？

// SequenceInputStream(Enumeration e)

// 通过简单的回顾我们知道了Enumeration是Vector中的一个方法的返回值类型。

// Enumeration<E> elements()

Vector<InputStream> v = new Vector<InputStream>();

InputStream s1 = new FileInputStream("ByteArrayStreamDemo.java");

InputStream s2 = new FileInputStream("CopyFileDemo.java");

InputStream s3 = new FileInputStream("DataStreamDemo.java");

v.add(s1);

v.add(s2);

v.add(s3);

Enumeration<InputStream> en = v.elements();

SequenceInputStream sis = new SequenceInputStream(en);

BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("Copy.java"));

### 序列化流

**序列化流：**把对象按照流一样的方式存入文本文件或者在网络中传输。对象 –> 流数据**(ObjectOutputStream)**

**反序列化流:**把文本文件中的流对象数据或者网络中的流对象数据还原成对象。流数据 –>对象**(ObjectInputStream)**

//我们先写一个实体类，并实现序列化接口

public class Person implements Serializable {

private String name;

private int age;

public Person() {

super();

}

public Person(String name, int age) {

super();

this.name = name;

this.age = age;

}

。。。。。。。。

//写入数据

public class ObjectStreamDemo {

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

// 创建序列化流对象

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("oos.txt"));

// 创建对象

Person p = new Person("阿杜", 25);

// public final void writeObject(Object obj)

oos.writeObject(p);

// 释放资源

oos.close();

//读取数据

public class ObjectStreamDemo {

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

// 创建反序列化对象

ObjectInputStream ois = new ObjectInputStream(new FileInputStream(

"oos.txt"));

// 还原对象

Object obj = ois.readObject();

// 释放资源

ois.close();

// 输出对象

System.out.println(obj);

}

}

### Scanner类

由于任何数据都必须通过同一模式的捕获组检索或通过使用一个索引来检索文本的各个部分。于是可以结合使用正则表达式和从输入流中检索特定类型数据项的方法。这样，除了能使用正则表达式之外，Scanner类还可以任意地对字符串和基本类型(如int和double)的数据进行分析。借助于Scanner，可以针对任何要处理的文本内容编写自定义的语法分析器。

**next() 与 nextLine() 区别**

next():

* 1、一定要读取到有效字符后才可以结束输入。
* 2、对输入有效字符之前遇到的空白，next() 方法会自动将其去掉。
* 3、只有输入有效字符后才将其后面输入的空白作为分隔符或者结束符。
* next() 不能得到带有空格的字符串。

nextLine()：

* 1、以Enter为结束符,也就是说 nextLine()方法返回的是输入回车之前的所有字符。
* 2、可以获得空白。

注意：如果要输入 int 或 float 类型的数据，在 Scanner 类中也有支持，但是在输入之前最好先使用 hasNextXxx()方法进行验证，再使用 nextXxx() 来读取

class ScannerDemo {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

double sum = 0;

int m = 0;

while (scan.hasNextDouble()) {

double x = scan.nextDouble();

m = m + 1;

sum = sum + x;

}

System.out.println(m + "个数的和为" + sum);

System.out.println(m + "个数的平均值是" + (sum / m));

scan.close();

}

}

自己写了一个简单的小程序用来剪辑特定长度的音频，并将它们混剪在一起，大体思路是这样的：

1. 使用 FileInputStream 输入两个音频

2. 使用 FileInputStream的skip(long n) 方法跳过特定字节长度的音频文件，比如说：输入 skip(1024\*1024\*3)，这样就能丢弃掉音频文件前面的 3MB 的内容。

3. 截取中间特定长度的音频文件：每次输入 8KB 的内容，使用 count 记录输入次数，达到设置的次数就终止音频输入。比如说要截取 2MB 的音频，每次往输入流中输入 8KB 的内容，就要输入 1024\*2/8 次。

4. 往同一个输出流 FileOutputStream 中输出音频，并生成文件，实现音频混合。

下面就给出相关代码：

public class MusicCompound

{

    public static void main(String args[])

    {

        FileOutputStream fileOutputStream = null;

        FileInputStream fileInputStream = null;

        String fileNames[] = {"E:/星月神话.mp3","E:/我只在乎你.mp3"};

        //设置byte数组，每次往输出流中传入8K的内容

        byte by[] = new byte[1024\*8];

        try

        {

            fileOutputStream = new FileOutputStream("E:/合并.mp3");

            for(int i=0;i<2;i++)

            {

                int count = 0;

                fileInputStream = new FileInputStream(fileNames[i]);

                //跳过前面3M的歌曲内容

                fileInputStream.skip(1024\*1024\*3);

                while(fileInputStream.read(by) != -1)

                {

                    fileOutputStream.write(by);

                    count++;

                    System.out.println(count);

                    //要截取中间2MB的内容，每次输入8k的内容，所以输入的次数是1024\*2/8

                    if(count == (1024\*2/8))

                    {

                        break;

                    }

                }

            }

        }

        catch(FileNotFoundException e)

        {

            e.printStackTrace();

        }

        catch(IOException e)

        {

            e.printStackTrace();

        }

        finally

        {

            try

            {

                //输出完成后关闭输入输出流

                fileInputStream.close();

                fileOutputStream.close();

            }

            catch(IOException e)

            {

                e.printStackTrace();

            }

        }

    }

# socket编程

**socket在应用层之下，传输层之上的一个接口**

public class SocketServer {

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

// 监听指定的端口

int port = 55533;**（服务器）**

**ServerSocket** server = new ServerSocket(port);

// server将一直等待连接的到来

System.out.println("server将一直等待连接的到来");

Socket socket = **server.accept();**

// 建立好连接后，从socket中获取输入流，并建立缓冲区进行读取

InputStream inputStream = **socket.getInputStream();**

byte[] bytes = new byte[1024];

int len;

StringBuilder sb = new StringBuilder();

while ((len = inputStream.read(bytes)) != -1) {

//注意指定编码格式，发送方和接收方一定要统一，建议使用UTF-8

sb.append(new String(bytes, 0, len,"UTF-8"));

}

System.out.println("get message from client: " + sb);

inputStream.close();

socket.close();

server.close();

}

}

public class SocketClient {

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

**// 要连接的服务端IP地址和端口**

String host = "127.0.0.1";

int port = 55533;

// 与服务端建立连接

Socket socket = new Socket(host, port);

// 建立连接后获得输出流

OutputStream outputStream = **socket.getOutputStream();**

String message="你好 yiwangzhibujian";

socket.getOutputStream().write(message.getBytes("UTF-8"));

outputStream.close();

socket.close();

}

}

# AIO NIO(同步非阻塞)

同步和异步的概念：实际的I/O操作

**同步是用户线程发起I/O请求后需要等待或者轮询内核I/O操作完成后才能继续执行**

**异步是用户线程发起I/O请求后仍需要继续执行，当内核I/O操作完成后会通知用户线程，**或者调用用户线程注册的回调函数

阻塞和非阻塞的概念：发起I/O请求

阻塞是指I/O操作需要彻底完成后才能**返回用户空间**

非阻塞是指I/O操作被调用后**立即返回一个状态值**，无需等I/O操作彻底完成。

**同步阻塞 IO** ：一般的IO

同步非阻塞 IO :NIO

多路复用

异步非阻塞 IO : AIO; 异步 IO 操作基于**事件和回调机制**，可以简单理解为，应用操作直接返回，而不会阻塞在那里，当后台处理完成，操作系统会通知相应线程进行后续工作。

## NIO

NIO本身是基于事件驱动思想来完成的，其主要想解决的是BIO的大并发问题。

NIO的最重要的地方是当一个连接创建后，会被注册到**多路复用器**上面，所以所有的连接只需要一个线程就可以搞定，当这个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理，也就是一个请求一个线程模式。

* 可简单认为：**IO是面向流的处理，NIO是面向块(缓冲区)的处理**
  + 面向流的I/O 系统**一次一个字节地处理数据**。
  + 一个面向块(缓冲区)的I/O系统**以块的形式处理数据**。

NIO主要有**三个核心部分组成**：

* **buffer缓冲区**
* **Channel管道**
* **Selector选择器**
* Channel不与数据打交道，它只负责运输数据。与数据打交道的是Buffer缓冲区
  + **Channel-->运输**
  + **Buffer-->数据**

Channel（通道）：Channel是一个对象，可以通过它读取和写入数据。可以把它看做是IO中的流;

**Buffer的属性**

容量（capacity）：缓冲区能够容纳的数据元素的最大数量。这一容量在缓冲区创建时被设定，**并且永远不能被改变**

上界（limit）：在**写模式**下，Buffer的limit表示你最多能往Buffer里写多少数据。写模式下，limit等于Buffer的capacity。当切换Buffer到**读模式**时，limit表示你最多能读到多少数据。

位置（position）：**当前游标的位置。**下一个要被读或写的元素的索引。位置会自动由相应的 get( )和 put( )函数更新

标记（mark）： 通过mark（）标记position位置，在进行其他操作后可通过reset（） 方法恢复到标记的position

这四个属性之间总是遵循以下关系：0 <= mark <= position <= limit <= capacity

1.写入数据到 Buffer；getChannel(),ByteBuffer.allocate(1024); inChannel.read(buffer);

2.调用 flip() 方法；flip方法将Buffer从**写模式切换到读模式**。调用flip()方法会首先将limit的值设为当前position的值

3.从 Buffer 中读取数据；outChannel.write(buffer);

4.调用 clear() 方法或者 compact() 方法。

Selector是一个对象，它可以注册到很多个Channel上，监听各个Channel上发生的事件，并且能够根据事件情况决定Channel读写。这样，通过一个线程管理多个Channel，就可以处理大量网络连接了。

**I/O复用模型**

在Linux下它是这样子实现I/O复用模型的：

* 调用**select/poll/epoll/pselect**其中一个函数，**传入多个文件描述符**，如果有一个文件描述符**就绪，则返回**，否则阻塞直到超时。
* I/O 多路复用的特点是**通过一种机制一个进程能同时等待多个文件描述符**，而这些文件描述符**其中的任意一个进入读就绪状态**，select()函数**就可以返回**。
* 描述符就是一个数字，**指向内核中的一个结构体**（文件路径，数据区等一些属性）。
* 并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。

我们在网络中使用NIO往往是I/O模型的**多路复用模型**！

* Selector选择器就可以比喻成麦当劳的**广播**。
* **一个线程能够管理多个Channel的状态**

**另外多路复用IO为何比非阻塞IO模型的效率高是因为在非阻塞IO中，不断地询问socket状态是通过用户线程去进行的，而在多路复用IO中，轮询每个socket状态是内核在进行的，这个效率要比用户线程要高的多。**

**NIO流程：**

首先，通过 Selector.open() 创建一个 Selector，作为类似**调度员**的角色。

然后，创建一个 **ServerSocketChannel**，并且向 Selector 注册

**Selector 阻塞在 select 操作，当有 Channel 发生接入请求，就会被唤醒。**

* 将**Socket通道**注册到Selector中，监听感兴趣的事件
* 当感兴趣的事件就绪时，则会进去我们处理的方法进行处理
* 每处理完一次就绪事件，删除该选择键(因为我们已经处理完了)

**AIO方式**使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，再通知服务器应用去启动线程进行处理。

异步 IO 的操作基于事件和回调机制。

# Netty

**这是神一样存在的java nio框架**

用一句简单的话来说就是：**Netty封装了JDK的NIO**，让你用得更爽，你不用再写一大堆复杂的代码了。  
 用官方正式的话来说就是：**Netty是一个异步事件驱动的网络应用框架，**用于快速开发可维护的高性能服务器和客户端。