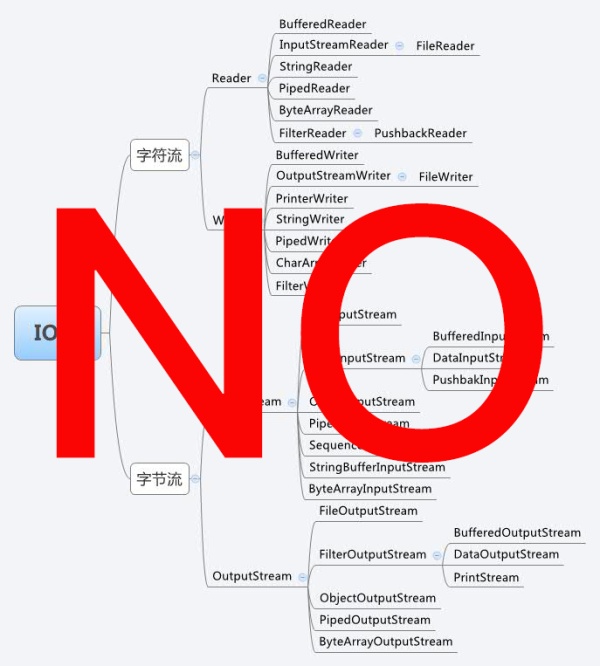
## 文章: [Java IO，硬骨头也能变软](https://zhuanlan.zhihu.com/p/28286559)

作者：小明  
链接：https://zhuanlan.zhihu.com/p/28286559  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

已私訊作者獲得授權做教學用，原文是簡體字，為尊重作者我就不轉繁體了。

## 开胃菜

先看一张网上流传的[http://java.io](https://link.zhihu.com/?target=http%3A//java.io)包的类结构图：



当你看到这幅图的时候，我相信，你跟我一样内心是崩溃的。

有些人不怕枯燥，不怕寂寞，硬着头皮看源码，但是，能坚持下去全部看完的又有几个呢！

然而，就算源码全部看完看懂，过不了几天，脑子里也会变成一团浆糊。

因为这里的类实在太多了。可能我们反复看，反复记，也很难做到清晰明白。

他就像是一块超级硬的骨头，怎么啃都啃不烂。

面对这样的做法，要坚决对他说，**NO**。

## 记不住，怎么办？

我的做法是找出他们的共性，给他们分类，只记典型，触类旁通。

上面的图虽然有分类，但是还不够细，而且没有总结出方便记忆的规律，所以我们要重新整理和归类。

这篇文章中，使用了两种分时给他们分组，目的是更全面的了解共性，帮助记忆。

## 分类一：按操作方式（类结构）

* **字节流和字符流：**

(台湾叫 字节和字符，我个人比较喜欢台湾的翻译)

* + 字节流：以字节为单位，每次次读入或读出是8位数据。可以读任何类型数据。
  + 字符流：以字符为单位，每次次读入或读出是16位数据。其只能读取字符类型数据。
* **输出流和输入流：**
  + 输出流：从内存读出到文件。只能进行写操作。
  + 输入流：从文件读入到内存。只能进行读操作。

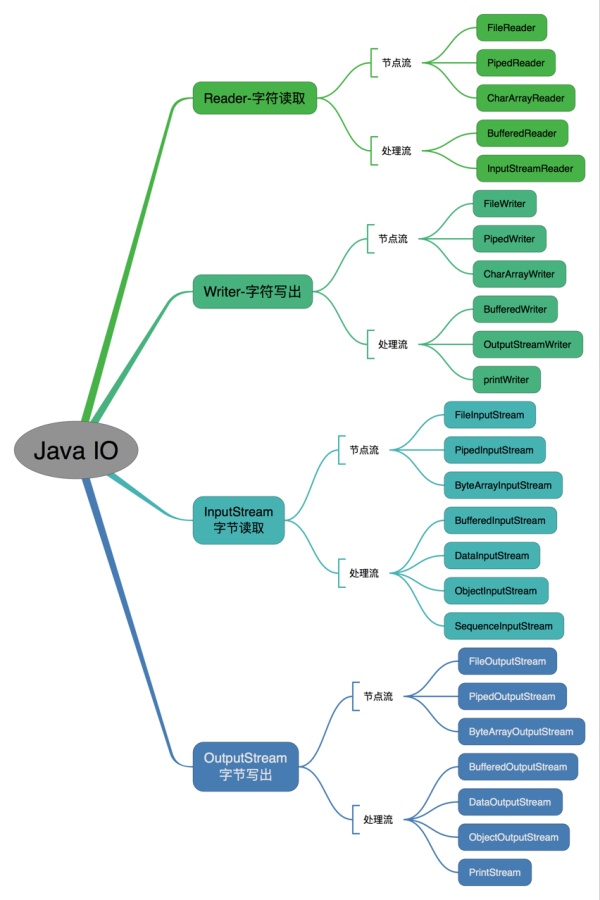
注意：这里的出和入，都是相对于系统内存而言的。

* **节点流和处理流：**
  + 节点流：直接与数据源相连，读入或读出。
  + 处理流：与节点流一块使用，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。

为什么要有处理流？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。

**按操作方式分类结构图：**

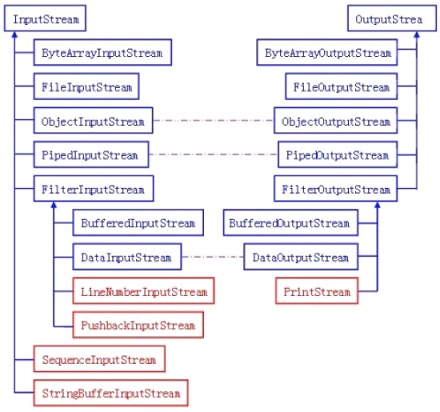
根据以上分类，以及jdk的说明，我们可以画出更详细的类结构图，如下：



**分类说明**

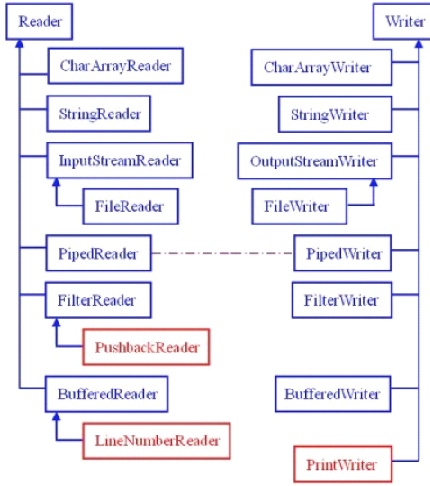
* **1. 输入字节流InputStream**：  
  输入字节流的继承图可见上图，可以看出：
  + FileInputStream： 是三种基本的介质流，它们分别从Byte 数组、StringBuffer、和本地文件中读取数据。
  + ByteArrayInputStream：
  + PipedInputStream： 是从与其它线程共享的管道中读取数据。PipedInputStream的一个实例要和PipedOutputStream的一个实例共同使用，共同完成管道的读取写入操作。主要用于线程操作。
  + ObjectInputStream 和所有FilterInputStream 的子类都是装饰流（装饰器模式的主角）
* **2. 输出字节流OutputStream：**输出字节流的继承图可见上图，可以看出：
  + FIleOutputStream：是两种基本的介质流
  + ByteArrayOutputStream： 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：是向与其它线程共享的管道中写入数据。
  + ObjectOutputStream 和所有FilterOutputStream 的子类都是装饰流。

字节流的输入和输出对照图：



* **3. 字符输入流Reader：**在上面的继承关系图中可以看出：
  + FileReader：
  + PipedReader：是从与其它线程共享的管道中读取数据
  + CharArrayReader：
  + 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。
  + 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。
  + 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。
  + 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。
* **4. 字符输出流Writer：**在上面的关系图中可以看出：
  + FileWriter:
  + PipedWriter:是向与其它线程共享的管道中写入数据
  + CharArrayWriter:
  + 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。
  + BufferedWriter 是一个装饰器，为Writer 提供缓冲功能。
  + PrintWriter 和PrintStream 极其类似，功能和使用也非常相似。
  + 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，后面会有它们的对应图。

字符流的输入和输出对照图：



* **5. 字符流与字节流转换**
  + 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。

* 具体的实现：

InputStreamReader:输入*流转到读流；*

String fileName= "d:"+File.separator+"hello.txt";

File file=new File(fileName);

Writer out=new OutputStreamWriter(new FileOutputStream(file));

out.write("hello");

out.close();

OutputStreamWriter:*输出流转到写流；*

String fileName= "d:"+File.separator+"hello.txt";

File file=new File(fileName);

Reader read=new InputStreamReader(new FileInputStream(file));

char[] b=new char[100];

int len=read.read(b);

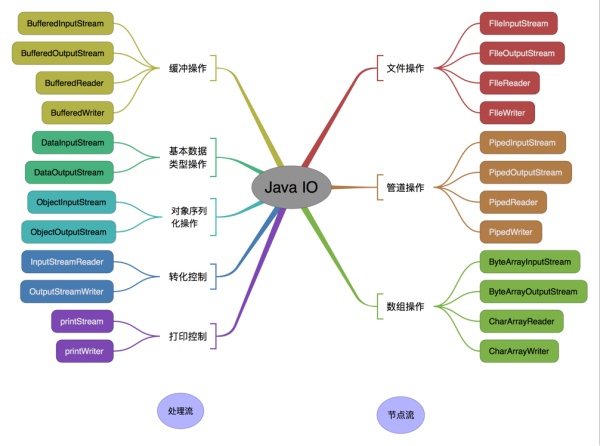
System.out.println(new String(b,0,len));

read.close();

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。

## 分类二：按操作对象

**按操作对象分类结构图：**



**分类说明：**

* 对文件进行操作（节点流）：
  + FileInputStream（字节输入流），
  + FileOutputStream（字节输出流），
  + FileReader（字符输入流），
  + FileWriter（字符输出流）
* 对管道进行操作（节点流）：
  + PipedInputStream（字节输入流）,
  + PipedOutStream（字节输出流），
  + PipedReader（字符输入流），
  + PipedWriter（字符输出流）。  
    PipedInputStream的一个实例要和PipedOutputStream的一个实例共同使用，共同完成管道的读取写入操作。主要用于线程操作。
* 字节/字符数组流（节点流）：
  + ByteArrayInputStream，
  + ByteArrayOutputStream，
  + CharArrayReader，
  + CharArrayWriter；  
    是在内存中开辟了一个字节或字符数组。

除了上述三种是节点流，其他都是处理流，需要跟节点流配合使用。

* Buffered缓冲流（处理流）：
  + BufferedInputStream，
  + BufferedOutputStream，
  + BufferedReader,
  + BufferedWriter,  
    是带缓冲区的处理流，缓冲区的作用的主要目的是：避免每次和硬盘打交道，提高数据访问的效率。
* 转化流（处理流）：
  + InputStreamReader：把字节转化成字符；
  + OutputStreamWriter：把字节转化成字符。
* 基本类型数据流（处理流）：用于操作基本数据类型值。
  + DataInputStream，
  + DataOutputStream。  
    因为平时若是我们输出一个8个字节的long类型或4个字节的float类型，那怎么办呢？可以一个字节一个字节输出，也可以把转换成字符串输出，但是这样转换费时间，若是直接输出该多好啊，因此这个数据流就解决了我们输出数据类型的困难。数据流可以直接输出float类型或long类型，提高了数据读写的效率。
* 打印流（处理流）：
  + PrintStream，
  + PrintWriter，  
    一般是打印到控制台，可以进行控制打印的地方。
* 对象流（处理流）：
  + ObjectInputStream，对象反串行化；
  + ObjectOutputStream，对象串行化；  
    把封装的对象直接输出，而不是一个个在转换成字符串再输出。
* 合并流（处理流）：
  + SequenceInputStream：可以认为是一个工具类，将两个或者多个输入流当成一个输入流依次读取。

## 典型使用案例

* **1. 复制文件：**

/\*\*

\* 复制文件:一边读，一边写

\*/

class hello {

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

if (args.length != 2) {

System.out.println("命令行参数输入有误，请检查");

System.exit(1);

}

File file1 = new File(args[0]);

File file2 = new File(args[1]);

if (!file1.exists()) {

System.out.println("被复制的文件不存在");

System.exit(1);

}

InputStream input = new FileInputStream(file1);

OutputStream output = new FileOutputStream(file2);

if ((input != null) && (output != null)) {

int temp = 0;

while ((temp = input.read()) != (-1)) {

output.write(temp);

}

}

input.close();

output.close();

}

}

* 说明：
  + 流在使用结束后，一定要执行关闭操作，即调用close( )方法。
  + FileInputStream.read()：  
    这个方法是对这个流一个一个字节的读，返回的结果就是这个字节的int表示方式；  
    当已经没有内容时，返回的结果为-1；
  + FileOutputStream.write()：  
    将内容写到文件。
* **2. 不使用FIle，将流中的字符转换大写小：**

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

String str = "ROLLENHOLT";

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

ByteArrayOutputStream output = new ByteArrayOutputStream();

int temp = 0;

while ((temp = input.read()) != -1) {

char ch = (char) temp;

output.write(Character.toLowerCase(ch));

}

String outStr = output.toString();

input.close();

output.close();

System.out.println(outStr);

}

* 说明：
  + 流在使用结束后，一定要执行关闭操作，即调用close( )方法。
* **3. 使用管道流在多个线程间通信**：

/\*\*

\* 消息发送类

\* \*/

class Send implements Runnable {

private PipedOutputStream out = null;

public Send() {

out = new PipedOutputStream();

}

public PipedOutputStream getOut() {

return this.out;

}

public void run() {

String message = "hello , Rollen";

try {

out.write(message.getBytes());

} catch (Exception e) {

e.printStackTrace();

}

try {

out.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

/\*\*

\* 接受消息类

\*/

class Recive implements Runnable {

private PipedInputStream input = null;

public Recive() {

this.input = new PipedInputStream();

}

public PipedInputStream getInput() {

return this.input;

}

public void run() {

byte[] b = new byte[1000];

int len = 0;

try {

len = this.input.read(b);

} catch (Exception e) {

e.printStackTrace();

}

try {

input.close();

} catch (Exception e) {

e.printStackTrace();

}

System.out.println("接受的内容为 " + (new String(b, 0, len)));

}

}

/\*\*

\* 测试类

\*/

class hello {

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

Send send = new Send();

Recive recive = new Recive();

try {

//管道连接

send.getOut().connect(recive.getInput());

} catch (Exception e) {

e.printStackTrace();

}

new Thread(send).start();

new Thread(recive).start();

}

}

* **4. 使用缓冲区从键盘上读入内容：**

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

BufferedReader buf = new BufferedReader(

new InputStreamReader(System.in));

String str = null;

System.out.println("请输入内容");

try{

str = buf.readLine();

}catch(IOException e){

e.printStackTrace();

}

System.out.println("你输入的内容是：" + str);

}

* **5. 将系统输出定向到文件：**

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

File file = new File("/Users/liuluming/Documents/hello.txt");

// 此刻直接输出到屏幕

System.out.println("hello");

try {

System.setOut(new PrintStream(new FileOutputStream(file)));

} catch (FileNotFoundException e) {

e.printStackTrace();

}

System.out.println("这些内容在文件中才能看到哦！");

}

## 其他类：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。

## 其他类：RandomAccessFile

该对象并不是流体系中的一员，其封装了字节流，同时还封装了一个缓冲区（字符数组），通过内部的指针来操作字符数组中的数据。 该对象特点：

1. 该对象只能操作文件，所以构造函数接收两种类型的参数：a.字符串文件路径；b.File对象。
2. 该对象既可以对文件进行读操作，也能进行写操作，在进行对象实例化时可指定操作模式(r,rw)。

**注意：**  
该对象在实例化时，如果要操作的文件不存在，会自动创建；如果文件存在，写数据未指定位置，会从头开始写，即覆盖原有的内容。 可以用于多线程下载或多个线程同时写数据到文件

补充:

補充資料沒有獲得作者同意，請大家不要外流

字节与字符(台湾叫 字节和字符，我个人比较喜欢台湾的翻译)

作者：胖胖  
链接：https://www.zhihu.com/question/39262026/answer/127103286  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

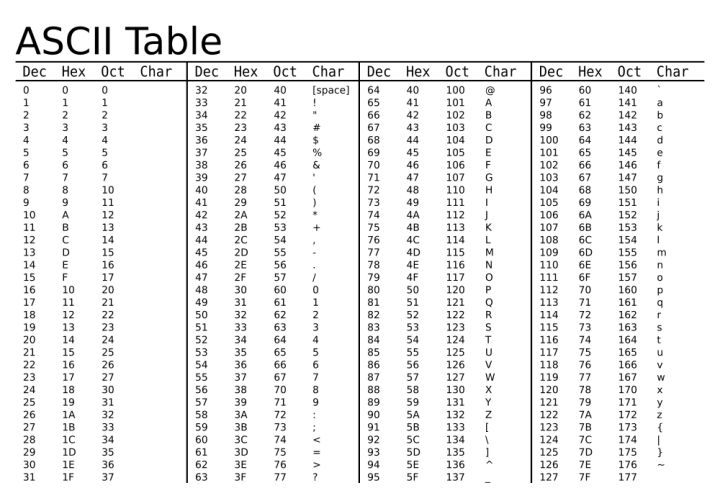
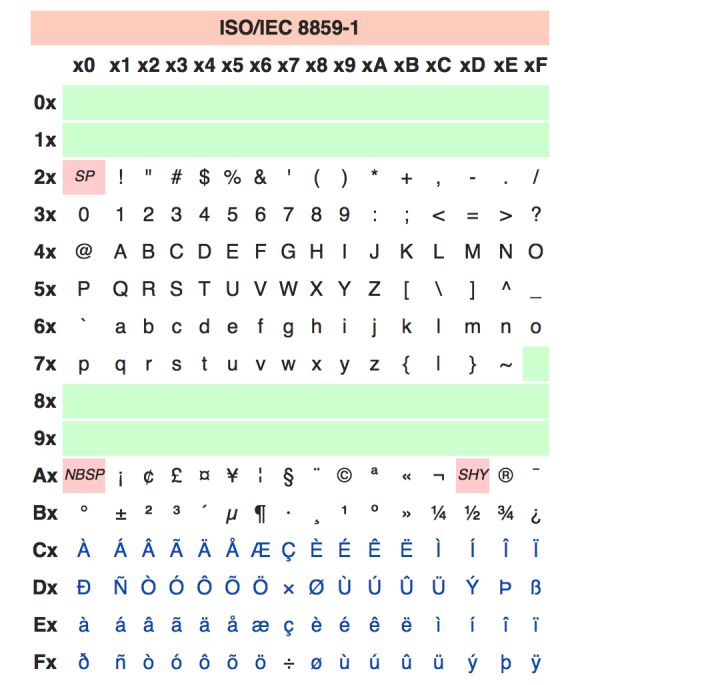
首先明确**“字节（Byte）”**和**“字符（Character）”**的大小：

* **1 byte = 8 bit**
* **1 char = 2 byte = 16 bit** (Java默认UTF-16编码)

虽然1 bit才是数据真正的最小单位，但1 bit 的信息量太少了。要表示一个有用的信息，需要好几个bit一起表示。所以除了硬件层面存在1个比特位的寄存器，大多数情况下，**字节是数据最小的基本单位**。我们熟知的基本型的大小都是8 bit（也就是1字节）的整数倍：

* boolean: 1 byte
* short: 2 byte
* int: 4 byte
* float: 4 byte
* long: 8 byte
* double: 8 byte

**ASCII**  
原本对于西方世界来讲，可能根本用不到“字符”这个东西。1个字节就解决全部问题了。因为一个字节8 bit，最多为个符号编码。英语26个字母，再加几个常用符号，标点，256个码位足够了。这就熟悉的\*\*ASCII码\*\*。如下图，ASCII码一共收录了空格及94个“可印刷字符”。每个字母或标点占一个字节。简简单单一个表，就把所有编码都解决了。

  
**ISO/IEC 8859-1**  
但西方世界不光只有英语一门语言。什么德语，法语，西班牙语都有自己的特殊字母。但这也没什么大不了的。每个国家都可以定义属于自己语言的特殊编码标准，而且大小照样不超过256。因为ASCII码中本身就有很多空码位没有使用。比如\*\*ISO/IEC 8859-n\*\*系就是国际标准化组织定义的一系列8位字符集。其中最常见的\*\*ISO/IEC 8859-1\*\*就是法语，芬兰语所用的西欧字符集。也是每个字母或符号用1个字节表示，下面这张表解决战斗。  
  
**中文编码**  
但26个字母到了中文，日语，韩语为代表的东亚象形文字面前就太小儿科了。汉字少说也有十万个。别说是8 bit，就算是16 bit也不一定够。如果说一个汉字代表一个**“字符”**，从这个时候开始，“字符”的概念开始凌驾于“字节”之上了。收到的每一个字节不能简单地译码成一个字母了，而是需要好几个字节组合成一个“汉字”。我国的汉字编码现行标准是**GB 18030**，每个字可以由1个、2个或4个字节组成，编码空间有161万个字符。另一个中国常用编码集是Big5。

**Unicode**

在出现Unicode之前，几乎每一种文字都有一套自己的编码方式。同一段**“字节流”**，在美帝可能是"hello world"，到我们天朝就变成“锟斤拷” ，“烫烫烫”了。所以**“Unicode”**可谓大势所趋。它的理念非常简单：**全世界每个不同语言的不同字符都统一编码，全球通行**。最初，每个字符占用2个字节，总共16个字符空间。比如，下面是“中国”两个字的Unicode代码。从第四版开始加入的“扩展字符集”开始使用4个字节（32 bit）编码。目前Unicode收录的字符规模大概在12万左右。

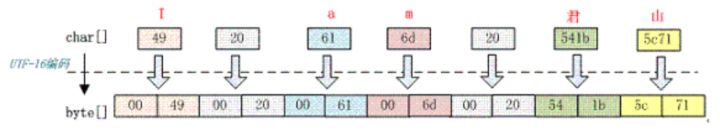
**中 4e2d 000d  
国 000a 56fd**

**UTF-16**  
编码里最容易搞混的一件事就是：**Unicode只是一套符号的编码。但计算机具体怎么读取这套编码，又是另外一件事。**

比如既然Unicode常规字符集占用2个字节，系统可以每次老老实实读取两个字节。然后用一个特殊符号告诉系统某个字符属于附加字符集，需要再往后读2个字节。比如说Java系统默认的UTF-16就是就是这样编码解码的：

考虑下面这句话：（图源：[深入分析 Java 中的中文编码问题](https://link.zhihu.com/?target=https%3A//www.ibm.com/developerworks/cn/java/j-lo-chinesecoding/" \t "_blank)）

**I am 君山**

Unicode字符集中：

**君=541b**。拆开存在两个字节里：“**54**”和“**1b**”。  
**山=5c71**。拆开存在两个字节里：“**5c**”和“**71**”。

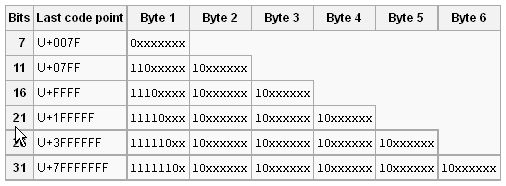
是不是很朴素？

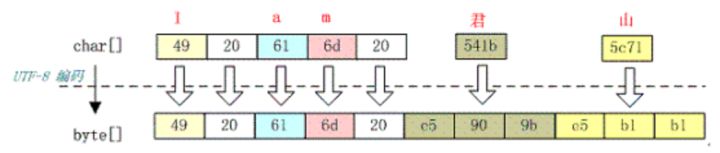
**UTF-8**   
但上面UTF-16的缺点也很明显：**就是所有英语字符“I am”也被迫用2个字节来编码。**比如，

**I=49**。在前面补零变成：“**00**”和“**49**”。

考虑到英语是使用最广泛的语言，用2个字节为1字节信息编码，浪费了内存空间。最好是让英语保持ASCII的编码，用1个字节，汉字等其他字符才用2个或更长的字节表示。这里就涉及到一个技术问题：**怎么让系统知道一个字符是用1个还是2个还是3个字节编码的呢？**这就是UTF-8做的事。

如下图所示，这里UTF-8可变长编码用到了一个小技巧：**用几位冗余信息告诉系统，当前字符有没有结束，是不是还需要继续往下读下一个字节。**

可以看到如果一个字节是**以“0”开头**的，说明是一个ASCII字符，只占一个字节。如果是**“11”开头**的，说明这个字符占用多个字节。后续每个“10”打头的字节都是这个字符的一部分。

下图演示了字符串**“I am 君山”**用 UTF-8 编码的结果：  


**君 = 541b = 0101 0100 0001 1011 (Unicode)**

需要用3个字节编码，把0101010000011011切成3部分变成**：**

0101 010000 011011

分别套上UTF-8字符头：

**1110** 0101 **10** 010000 **10** 011011 = e5 90 9b

所以如上图所示，**“君”**字用UTF-8编码就成了：**e5909b**。

**总而言之，一切都是字节流，其实没有字符流这个东西。字符只是根据编码集对字节流翻译之后的产物。**

**Java I/O的编码系统**  
Java IO库有两个支系，

* 面向字节流的**InputStream**和**OutputStream**
* 面向字符的**Reader**和**Writer**

之前说了，字节流的InputStream和OutputStream是一切的基础。实际总线中流动的只有字节流。需要对字节流做特殊解码才能得到字符流。Java中负责从字节流向字符流解码的桥梁是：

**InputStreamReader  
InputStreamWriter**

根据上面的编码规则，Reader返回的是一个解码后的Unicode码元。包装在一个int整型里返回。

abstract int read();

也就是收到3个字节后，去掉UTF-8报头，拼装起来得到“君”字的Unicode码元。

**1110** 0101 **10** 010000 **10** 011011 0101 0100 0001 1011

然后包装在4个字节的int整型里返回：

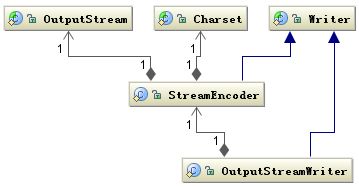
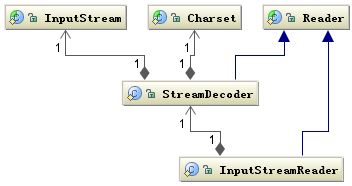
0101 0100 0001 1011 0000 0000 0000 0000 0101 0100 0001 1011

write()方法是一个相反的编码过程：

abstract void write(int c);

输入一个Unicode码元的int型，如果设定编码是UTF-8，内部会自动切割并添加报头。

以下是InputStreamReader和OutputStreamWriter的结构图，（图源：[深入分析 Java 中的中文编码问题](https://link.zhihu.com/?target=https%3A//www.ibm.com/developerworks/cn/java/j-lo-chinesecoding/" \t "_blank)）



从图中可以猜到，实际负责编码和解码的是**StreamDecoder**类和**StreamEncoder**类。过程中必须指定使用的字符编码集**Charset**。所以InputStreamReader和OutputStreamWriter的构造器都带有Charset类型的参数。

[**InputStreamReader**](https://link.zhihu.com/?target=http%3A//docs.oracle.com/javase/7/docs/api/java/io/InputStreamReader.html%23InputStreamReader%28java.io.InputStream%2C%2520java.nio.charset.Charset%29)([InputStream](https://link.zhihu.com/?target=http%3A//docs.oracle.com/javase/7/docs/api/java/io/InputStream.html) in, [Charset](https://link.zhihu.com/?target=http%3A//docs.oracle.com/javase/7/docs/api/java/nio/charset/Charset.html) cs)  
[**OutputStreamWriter**](https://link.zhihu.com/?target=https%3A//docs.oracle.com/javase/7/docs/api/java/io/OutputStreamWriter.html%23OutputStreamWriter%28java.io.OutputStream%2C%2520java.nio.charset.Charset%29)([OutputStream](https://link.zhihu.com/?target=https%3A//docs.oracle.com/javase/7/docs/api/java/io/OutputStream.html) out, [Charset](https://link.zhihu.com/?target=https%3A//docs.oracle.com/javase/7/docs/api/java/nio/charset/Charset.html) cs)

如果没有指定编码集，将使用系统默认编码集。而我们经常使用的FileInputReader和FileOutputWriter就是InputStreamReader和OutputStreamWriter的派生类。

**内存String编码**  
另外一个要使用到Charset编码集的地方，是String的构造器和getBytes()方法。也可以通过参数控制具体使用的编码集。

String s = "这是一段中文字符串";

byte[] b = s.getBytes("UTF-8");

String n = new String(b,"UTF-8");

**nio的字符编码**  
另外nio包里的ByteBuffer的asCharBuffer()方法也可以实现字节流和字符流之间的转换。

FileChannel fc=new FileInputStream(f).getChannel();

ByteBuffer bf=ByteBuffer.allocate(1024);

fc.read(bf);

bf.flip();

CharBuffer cf=bf.asCharBuffer();

但这里有个坑需要注意，asCharBuffer()方法，默认以**UTF-16BE**来解码byteBuffer里的字节。每个字符2字节。而String # getBytes()使用系统默认编码方式，大多数情况都不是UTF-16BE。所以经常CharBuffer里读取出来的会是乱码。