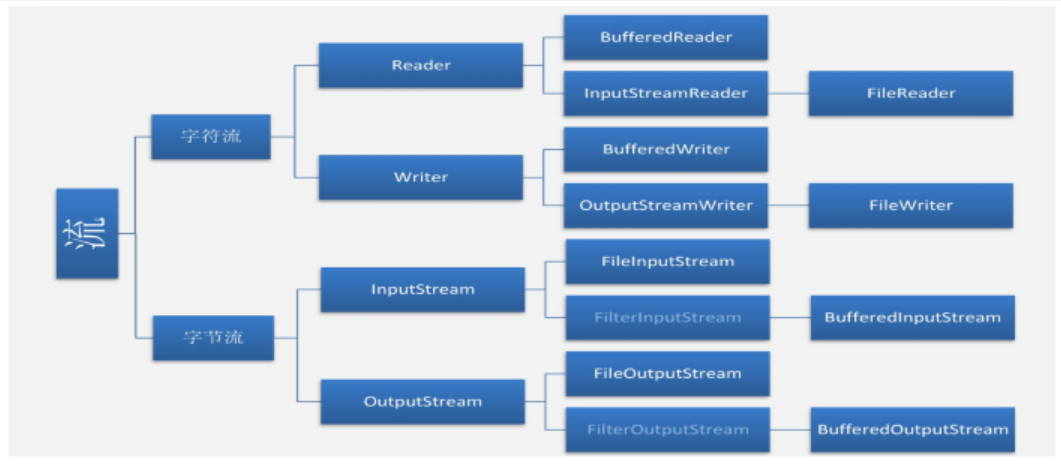
# IO流

## 流的结构图



流：可以理解为数据的流动，就是一个数据流，IO流最终要以对象来体现，对象都存在于java.io包中。

## 流的分类

（1）输入流（读）和输出流（写）

输入流：把文件中的东西进行输入到内存中。

输出流：把内存中的东西输出到文件中。

我们可以使用内存来做分别，输入到内存就是输入流，从内存中输出就是输出流。

（2）处理数据的不同，分为字节流和字符流。

## 字节和字符的理解

字节（byte）是计算机中用于计量存储容量的一种计量单位，也表示计算机编程语言中的数据类型和语言字符。byte即字节的意思，是java中的基本数据类型，用来声明字节型的变量，一个字节包含8个位，所以，byte类型的取值范围是-128到127。通常在读取非文本文件时（如图片，声音，可执行文件）需要用字节数组来保存文件的内容，在下载文件时，也是用byte数组作临时的缓冲器接收文件内容，所以说byte在文件操作时是必不可少的，不管是对文件写入还是读取都要用到。

计算机只知道字节，而字符是语义上的单位，它是有编码的，一个字符可能编码成1个2个甚至3个4个字节。这跟字符集编码有关系，英文字母和数字是单字节，但汉字这些自然语言中的字符是多字节的。一个字节只能表示

255个字符，不可能用于全球那么多种自然语言的处理，因此肯定需要多字节的存储方式。

## 流的基本知识

Java语言提供了4个非常重要的输入输出类，分别是：InputStream、OutputStream、Reader、Writer，它们都是抽象类。

# 编码

1、gbk编码中中文占用2个字节，英文占用1个字节。

2、**utf-8编码**中中文占用3个字节，英文占用1个字节。

3、utf-16be中文占用2个字节，英文也是占用2个字节。

4、当你的字节序列是某种编码时，这个时候想把字节序列变成字符串时，也需要用这种编码方式，否则会出现乱码。

5、文本文件就是字节序列，可以是任意编码的字节序列，如果我们在中文机器上直接创建文本文件，那么该文本文件只认识ansi编码。

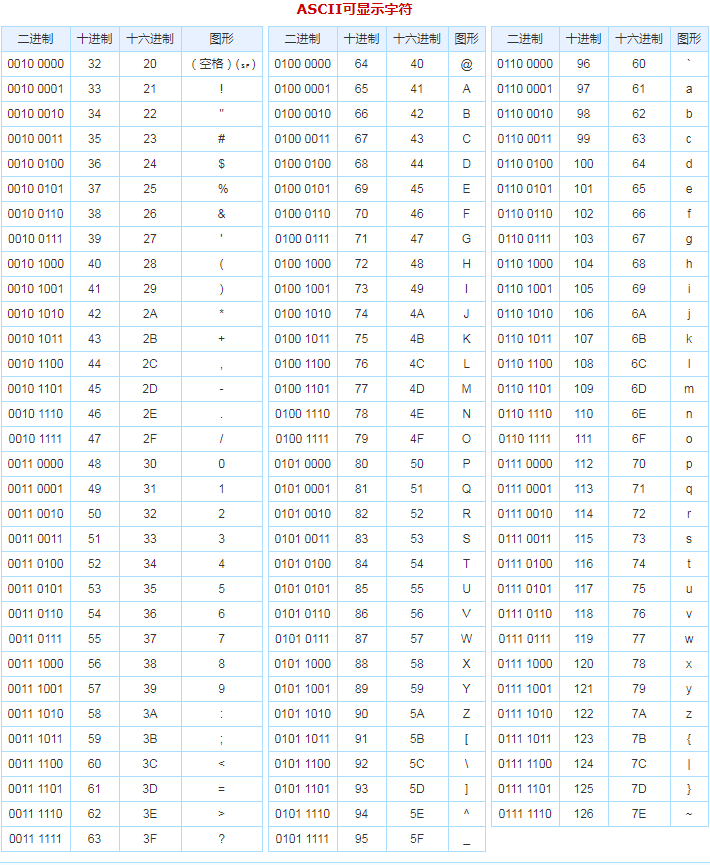
6、我们的java项目的编码方式如何查看：在eclipse中点击项目右键，选择属性，然后查看资源，资源中显示了项目的编码方式。

# ASCII编码

ASCII（American Standard Code for Information Interchange，美国信息互换标准代码，ASCⅡ）是基于拉丁字母的一套电脑编码系统。它主要用于显示现代英语和其他西欧语言。它是现今最通用的单字节编码系统，并等同于国际标准ISO/IEC 646。

　　ASCII第一次以规范标准的型态发表是在1967年，最后一次更新则是在1986年，至今为止共定义了128个字符，其中33个字符无法显示（这是以现今操作系统为依归，但在DOS模式下可显示出一些诸如笑脸、扑克牌花式等8-bit符号），且这33个字符多数都已是陈废的控制字符，控制字符的用途主要是用来操控已经处理过的文字，在33个字符之外的是95个可显示的字符，包含用键盘敲下空白键所产生的空白字符也算1个可显示字符（显示为空白）。





# Unicode编码

Unicode（[统一码](https://baike.baidu.com/item/%E7%BB%9F%E4%B8%80%E7%A0%81)、万国码、单一码）是计算机科学领域里的一项业界标准，包括字符集、编码方案等。Unicode 是为了解决传统的字符编码方案的局限而产生的，它为每种语言中的每个字符设定了统一并且唯一的[二进制](https://baike.baidu.com/item/%E4%BA%8C%E8%BF%9B%E5%88%B6)编码，以满足跨语言、跨平台进行文本转换、处理的要求。

作用：能够使计算机实现跨语言、[跨平台](https://baike.baidu.com/item/%E8%B7%A8%E5%B9%B3%E5%8F%B0)的文本[转换](https://baike.baidu.com/item/%E8%BD%AC%E6%8D%A2)及处理。

层次：Unicode 编码系统，可分为[编码方式](https://baike.baidu.com/item/%E7%BC%96%E7%A0%81%E6%96%B9%E5%BC%8F)和实现方式两个层次。

方式：Unicode是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。目前的Unicode字符分为17组编排，0x0000 至 0x10FFFF，每组称为平面（Plane），而每平面拥有65536个码位，共1114112个。然而目前只用了少数平面。[UTF-8](https://baike.baidu.com/item/UTF-8)、[UTF-16](https://baike.baidu.com/item/UTF-16)、[UTF-32](https://baike.baidu.com/item/UTF-32)都是将数字转换到程序数据的编码方案。

## UTF-8

UTF-8（8-bit Unicode Transformation Format）是一种针对Unicode的可变长度字符编码，又称万国码，由Ken Thompson于1992年创建。现在已经标准化为RFC 3629。UTF-8用1到6个字节编码Unicode字符。用在网页上可以统一页面显示中文简体繁体及其它语言（如英文，日文，韩文）。

# File类

1、java.io.File类用于表示文件（或者目录），可以用于操作硬盘上的文件或者目录。File类只用于表示文件（或者目录）的信息，如名称、大小，不能用于文件内容的访问。

2、File.separator 用于设置分隔符（\），不管是window系统还是linux系统都是适用的。

3、直接打印file对象输出的是文件的目录。

## 构造函数

### public File(String pathName)

pathName:文件的路径名字符串

### public File(String parent,String child)

parent:父路径字符串 child:子路径字符串

### public File(File parent, String child)

parent:父抽象的路径名

child：子路径名字符串

从一个父抽象的路径名和一个子路径字符串创建一个新的File实例，如果parent为null，那么就会创建新的文件实例，就像在给定的子路径名字符串上调用单参数文件构造函数一样，否则，使用父抽象路径名表示目录，使用子路径名字符串表示目录或文件。如果子路径名字符串是绝对的，那么它将以依赖于系统的方式转换为相对路径名。如果parent是空的抽象路径名，那么通过将child转换为抽象路径名并根据系统相关的默认目录解析结果创建新文件实例，否则，每个pathname字符串将转换为抽象的pathname，子抽象pathname将解析为父路径。

## 方法

### （1）String getName()

//获取文件（或者文件夹）的名称

### （2）boolean canRead()

//判断文件是否是可读的。

### （3）boolean canWrite()

//判断文件是否可被写入

### （4）boolean exists()

//判断文件是否存在

### （5）long length()

//获取文件的长度（以字节为单位）

### （6）String getAbsolutePath()

//获取文件的绝对路径

### （7）String getParent()

//获取文件的父路径

### （8）boolean isFile()

//判断是否是一个文件

### （9）boolean isDirectory()

//判断文件是否是一个目录

### （10）boolean isHidden()

//判断文件是否是隐藏文件

### （11）Long lastModified()

//获取文件最后的修改时间

### （12）boolean renameTo(File dest)

//重新命名此抽象路径名表示的文件

### （13）File[] listFiles()

//获取同一目录下文件的列表（包含文件和子目录中的文件抽象）

### （14）String[] list()

//用于列出当前目录下的子目录和文件（不会列出子目录下面的文件）

### （15）public boolean createNewFile()

//当且仅当不存在指定路径的文件时，创建一个新的空文件。

### （16）public boolean delete()

//删除抽象路径名表示的文件或目录

### （17）public boolean mkdir()

//创建文件夹

### （18）public boolean mkdirs()

//创建多级文件夹

### （19）public static File createTempFile(String prefix, String suffix)

在默认临时文件目录中创建一个空文件，使用给定的前缀和后缀生成它的名称

参数：

prefix: 用于生成文件名称的前缀字符串;必须至少有三个字符长

suffix: 用于生成文件名称的后缀字符串;可能是null，在这种情况下后缀将使用.tmp

返回表示新创建的空文件的抽象路径名

### （20）public static File createTempFile(String prefix, String suffix,File directory)

这个方法和上面的方法含义一样。

# Closeable接口

可关闭的数据源或目的地，调用close方法释放对象持有的资源(例如打开的文件)

## 方法

### public void close()

关闭此流并释放与之关联的任何系统资源。如果流已经关闭，则调用此方法无效。

# InputStream类

这个抽象类是表示字节输入流所有类的超类，需要定义InputStream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。

## 方法

### public abstract int read()

从输入流中读取下一个字节的数据，字节值以int的形式返回，范围为0到255，如果没有字节可用，因为流的末尾已经到达，则返回值是-1。此方法将阻塞直到输入数据可用、检测到流的结束或抛出异常为止。

返回下一个字节的数据，或者如果流的末尾已经到达，则返回-1。

### public int read(byte b[])

参数：b 读取缓冲区中的数据

返回：读入缓冲区的总字节数，或者-1是没有更多的数据，因为流的末尾已经到达。

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

参数：

b 被读取的缓存区的数据

off 数组b中写入数据的起始偏移量。

len 要读取的最大字节数

返回读入缓存区的总字节数，或者返回-1没有更多的数据，因为流的末尾已经到达。

### public int available()

从输入流读取(或跳过)字节数的估计，当它到达输入流的末尾时，不会阻塞或0。

### public void close()

关闭此输入流并释放与该流关联的任何系统资源。

## 代码示例

https://www.cnblogs.com/kongxianghao/articles/6879367.html

# OutputStream

这个抽象类是表示字节输出流的所有类的超类。输出流接受输出字节并将其发送到某个接收器。需要定义OutputStream子类的应用程序必须始终至少提供一个写入一个字节输出的方法。

## 方法

### public abstract void write(int b)

将指定的字节写入到输出流。写入的一般约定是将一个字节写入输出流。要写入的字节是参数b的8个低阶位，忽略b的24个高阶位。OutputStream的子类必须提供这个方法的一个实现。

参数：b 要写入的字节

### public void write(byte b[])

从指定的字节数组中写入b.length长度的字节到输出流中。

参数：b 写入的数据

### public void write(byte b[], int off, int len)

从偏移位置开始的指定字节数组中写入len字节到这个输出流。

参数：b 写入的数据

off 数据起始的偏移位置

len 写入的字节数目

### public void flush()

刷新此输出流并强制写入任何缓冲输出字节。flush的一般约定是，调用它表明，如果以前编写的任何字节都被输出流的实现缓冲了，那么应该立即将这些字节写到预期的目的地。如果流的目标是基础操作系统提供的抽象(例如文件)，则刷新流只保证之前写入到流的字节被传递给操作系统进行写入;它不能保证它们确实被写入到物理设备(如磁盘驱动器)中。

仅仅是刷新缓冲区(一般写字符时要用,因为字符是先进入的缓冲区)，将内存中的数据立刻写出，流对象还可以继续使用。

### public void close()

关闭此输出流并释放与此流关联的任何系统资源。close的一般约定是关闭输出流。关闭的流不能执行输出操作和不能重新打开。

关闭流对象，但是（也会）先刷新一次缓冲区，关闭之后，流对象不可以继续再使用了。

# ByteArrayOutputStream类

字节数组输出流在内存中创建一个字节数组缓冲区，所有发送到输出流的数据保存在该字节数组缓冲区中。

此类实现了一个输出流，其中的数据被写入一个 byte 数组。缓冲区会随着数据的不断写入而自动增长。可使用 toByteArray() 和 toString() 获取数据。

## 构造函数

### ByteArrayOutputStream()

这个方法会创建一个32字节（默认大小）的缓冲区。

### ByteArrayOutputStream(int a)

创建一个大小为n字节的缓冲区

## 方法

### void write(int w)

将指定的字节写入此字节数组输出流

### void write(byte []b, int off, int len)

将指定字节数组中从偏移量 off 开始的 len 个字节写入此字节数组输出流

### byte[] toByteArray()

创建一个新分配的字节数组。数组的大小和当前输出流的大小，内容是当前输出流的拷贝

### void reset()

将此字节数组输出流的 count 字段重置为零，从而丢弃输出流中目前已累积的所有数据输出。

### synchronized String toString()

使用平台默认的字符集，通过解码字节将缓冲区内容转换为字符串。

## 代码示例



# ByteArrayInputStream类

字节数组输入流在内存中创建一个字节数组缓冲区，从输入流读取的数据保存在该字节数组缓冲区中，创建字节数组输入流对象有以下构造函数。

## 构造函数

### ByteArrayInputStream(byte [] a)

接收字节数组作为参数创建

### ByteArrayInputStream(byte []a, int off, int len)

接收一个字节数组，和两个整形变量 off、len，off表示第一个读取的字节，len表示读取字节的长度

## 方法

### int read()

从此输入流中读取下一个数据字节

### int read(byte[] r, int off, int len)

将最多 len 个数据字节从此输入流读入字节数组

### int available()

返回可不发生阻塞地从此输入流读取的字节数

### void mark(int read)

设置流中的当前标记位置

### long skip(long n)

从此输入流中跳过 n 个输入字节。

### public synchronized void reset()

将缓冲区重置为已标记的位置。除非在构造函数中指定了另一个位置或偏移量，否则标记的位置为0。

## 代码示例



# FileInputStream类

[Java](http://lib.csdn.net/base/java)的FileInputStream默认的编码方式就是文件的编码方式。

## 构造函数

### public FileInputStream(String name)

参数：系统依赖的文件名称

通过打开到一个实际文件的连接来创建FileInputStream，该文件由文件系统中的路径名命名，将创建一个新的FileDescriptor对象来表示此文件连接。

### public FileInputStream(File file)

参数：打开要阅读的文件

## 方法

### private native void open(String name)

打开指定的文件去读

参数：文件的名称

### public native int read()

从输入流中读取数据的字节。如果没有可用的输入，此方法将阻塞。返回下一个字节的数据，或-1如果达到文件的末尾。

### public int read(byte b[])

读取b.length个字节的数据从输入流到字节数组。此方法将阻塞，直到有一些输入可用为止。

参数：读取数据的缓存区

返回读入缓存区的总字节数，或者-1是没有更多的数据，因为流的末尾已经到达。

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

读取len个字节的数据从输入流到字节数组。此方法将阻塞，直到有一些输入可用为止。

### private native int readBytes(byte b[], int off, int len)

将子数组读取为字节序列。

参数：

b 被写入的数据

off 数据的起始偏移位置

len 被写入的字节长度

## 代码示例



# FileOutputStream类

文件输出流是用于将数据写入到文件中。

## 构造函数

### public FileOutputStream(String name)

name：系统依赖的文件名

创建一个输出文件流，以使用指定的名称写入文件。将创建一个新的FileDescriptor对象来表示此文件连接。首先，如果有一个安全管理器，就调用它的checkWrite方法，并将名称作为参数。

如果该文件存在，但是是一个目录而不是常规文件，不存在但不能创建，或者由于任何其他原因不能打开，则抛出FileNotFoundException。

### public FileOutputStream(String name, boolean append)

name：系统依赖的文件名

append：如果是true，然后字节将被写到文件的末尾而不是开始

### public FileOutputStream(File file)

file：打开以便写入的文件

### public FileOutputStream(File file, boolean append)

file：打开以便写入的文件

append：如果是true，然后字节将被写到文件的末尾而不是开始

## 方法

### public native void write(int b)

b：写入一个字节

写入指定的字节到文件输出流中

### public void write(byte b[])

b：字节数组

从指定的字节数组中写入b.length个字节到文件输出流中

# BufferedOutputStream

BufferedOutputStream 是缓冲输出流，它继承于FilterOutputStream。BufferedOutputStream 的作用是为另一个输出流提供“缓冲功能”。

## 构造函数

### public BufferedOutputStream(OutputStream out)

创建新的缓冲输出流以将数据写入指定的基础输出流。

参数：out 底层输出流

## 方法

### public synchronized void write(int b)

将指定的字节写入此缓冲的输出流。

参数：写入的字节

### public synchronized void write(byte b[], int off, int len)

将从offset off开始的指定字节数组中的len个字节写入此缓冲输出流。

### public synchronized void flush()

刷新此缓冲的输出流。 这会强制将任何缓冲的输出字节写出到底层输出流。

# 遍历目录

//列出指定目录下（包括其子目录）的所有文件

public static void listDirectory(File dir) throws IOException{

if(!dir.exists()){

throw new IllegalArgumentException("目录"+dir+"不存在");

}

if(!dir.isDirectory()){

throw new IllegalArgumentException("目录"+dir+"不是目录");

}

//返回的是直接子目录（文件）的抽象

File[] files = dir.listFiles();

if(files!=null && files.length>0){

for(File file:files){

if(file.isDirectory()){

//递归

listDirectory(file);

}else{

System.out.println(file);

}

}

}

}

# 常见的文件类型处理

## 属性文件

属性文件一行表示一个属性，属性就是键值对，键和值用等号=或冒号分隔，一般用于配置程序的一些参数。处理属性文件使用字符流是比较容易的，Java中使用Properties类去处理属性文件。

使用Properties类去处理属性文件的好处：

（1）自动处理空格、分隔符=前后的空格会被自动忽略。

（2）可以忽略空行

（3）可以添加注释，以字符#或！开头的行会被视为注释，进行忽略。

使用Properties也有限制，它不能直接处理中文，在配置文件中，所有非ASCII字符需要使用Unicode编码。

## CSV文件

CSV文件表示逗号分隔值，一般而言，一行表示一条记录，一条记录包括多个字段，字段之间用逗号分隔。不过一般而言，分隔符不一定是逗号，可能是其他字符，如tab符’\t’ 冒号’:’ 分号’;’等。程序中的各种日志文件通常是CSV文件，在导入导出表格类型的数据时，CSV也是经常用的一种格式。

对于处理CSV文件，我们不能使用字符流去进行处理，有一个第三方类库：Apache Commons CSV，对处理CSV提供了良好的支持。

## Excel文件

excel主要有两种格式：.xls和.xlsx。使用POI处理excel文件，有如下类：

（1）Workbook：表示一个Excel文件对象。

（2）Sheet：表示一个工作表

（3）Row：表示一行

（4）Cell：表示一个单元格

使用POI也可以方便的解析excel文件，使用WorkbookFactory的create方法即可。

## 压缩文件

压缩文件有多种格式，Java SDK支持两种：gzip和zip，gzip只能压缩一个文件，而zip文件中可以包含多个文件。如果需要更多格式，可以考虑Apache Commons Compress。

先来看gzip，有两个主要的类：

GZIPOutputStream

GZIPInputStream

他们分别是OutputStream和InputStream的子类，都是装饰类，GZIPOutputStream加到已有的流上，就可以实现压缩。而GZIPInputStream加到已有的流上，就可以实现解压缩。

zip文件支持一个压缩文件中包含多个文件，java API中主要的类是：

ZipOutputStream

ZipInputStream

他们分别是OutputStream和InputStream的子类，都是装饰类，但不能像GZIPOutputStream和GZIPInputStream那样简单使用，ZipOutputStream可以写入多个文件，它有一个重要的方法：

void putNextEntry(ZipEntry e) throws IOException

在写入每一个文件前，必须要先调用该方法，表示准备写入一个压缩条目ZipEntry，每个压缩条目有个名称，这个名称是压缩文件的相对路径，如果名称以字符”/”结尾，表示目录，他的构造方法是：

public ZipEntry(String name)

# RandomAccessFile

## 作用

随机流（RandomAccessFile）不属于IO流，支持对文件的读取和写入随机访问。

## 随机访问文件的原理

首先把随机访问的文件对象看作存储在文件系统中的一个大型 byte 数组，然后通过指向该 byte 数组的光标或索引（即：文件指针 FilePointer）在该数组任意位置读取或写入任意数据。

## 存在的意义

1、是JAVA I/O流体系中功能最丰富的文件内容访问类，它提供了众多方法来访问文件内容。

2、由于可以自由访问文件的任意位置，所以如果需要访问文件的部分内容，RandomAccessFile将是更好的选择。

3、可以用来访问保存数据记录的文件，文件的记录的大小不必相同，但是其大小和位置必须是可知的。

这个类在很多资料上翻译成中文都是：随机访问文件，在中文里，随机是具有不确定的含义，指一会访问这里，一会访问那里的意思。如果以这种语义来解释的话，就会感到很困惑。其实，Random在英文中不仅仅有随机，还有任意的意思。如果中文名为任意访问文件是不是就会更好的理解。任意表示我们可以指定文件中任何一个位置去操作一个文件。

（1）RandomAccessFile支持随机访问文件，可以访问文件的任意位置。

（2）Java文件模型，在硬盘上的文件是byte存储的，是数据的集合。

（3）打开文件有两种模式，分别是：rw（读写），r（只读），文件指针，打开文件时指针在开头pointer=0;

（4）写方法：write(int)—只写一个字节（后8位），同时指针指向下一个位置，准备再次写入。

（5）读方法：read()—读一个字节。

（6）文件读写完成以后一定要关闭。

## 构造函数

（1）RandomAccessFile(String name, String mode)

throws FileNotFoundException

（2）public RandomAccessFile(File file, String mode)

throws FileNotFoundException

参数：name表示文件路径，file表示文件对象。mode表示打开模式，可以有4个取值：

1、r：只用于读

2、rw：用于读写

3、rws：和rw一样，用于读和写，另外它要求文件内容和元数据的任何更新都同步到设备上。

4、rwd：和rw一样，用于读和写，另外它要求文件内容的任何更新都同步到设备上，和rws的区别是，元数据的更新不要求同步。

## 方法

### public native int read()

//从该文件读取一个字节的数据，字节在0到255之间作为整数返回。

### private native int readBytes(byte b[], int off, int len)

//将子数组读取为字节序列

### public native void write(int b)

//将指定的字节写入此文件。写入从当前文件指针开始

### void write(byte b[])

//

### public final void writeUTF(String str)

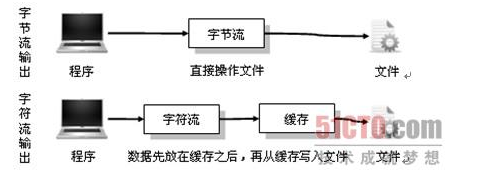
//写字符串到一个文件中，使用UTF-8编码，与机器无关的方式。

## 示例

# Reader类

Writer类与Reader类是以字符流传输数据，一个字符是两个字节，字符流除了是以字符方式（两个字节）传输数据外，另外一点与字节流不同的是字符流使用缓冲区，通过缓冲区再对文件进行操作，字节流通过字节对文件进行操作。使用字符流类时关闭字符流会强制将字符流缓冲区的内容输出，如果不想关闭也将字符流进行输出，使用Writer类的flush()方法。

实际上字节流在操作时本身不会用到缓冲区（内存），是文件本身直接操作的，而字符流在操作时使用了缓冲区，通过缓冲区再操作文件。



用于读取字符流的抽象类。子类必须实现的唯一方法是read（char []，int，int）和close()。但是，大多数子类将覆盖此处定义的一些方法，以提供更高的效率。

## 方法

### public int read(java.nio.CharBuffer target)

参数：将字符读入的缓冲区

返回：添加到缓冲区的字符数，如果此字符源位于其末尾，则为-1

### public int read()

读取一个字符，此方法将阻塞，直到字符可用，发生I / O错误或到达流的末尾。

返回：读取的字符作为0到65535范围内的整数，如果已到达流末尾则为-1

### public int read(char cbuf[])

将字符读入数组。 此方法将阻塞，直到某些输入可用，发生I / O错误或到达流的末尾。

参数：目标缓冲区

返回：读取的字符数，如果已到达流的末尾，则返回-1

### abstract public int read(char cbuf[], int off, int len)

参数：cbuf 目标缓冲区，off 开始存储字符的偏移量 len 要读取的最大字符数

返回：读取的字符数，如果已到达流的末尾，则返回-1

### abstract public void close()

关闭流并释放与其关联的所有系统资源。 一旦关闭了流，进一步的read（），ready（），mark（），reset（）或skip（）调用将抛出IOException。关闭先前关闭的流没有任何效果。

# Writer类

用于写入字符流的抽象类。子类必须实现的唯一方法是write（char []，int，int），flush（）和close（）。但是，大多数子类将覆盖此处定义的一些方法，以提供更高的效率。

## 方法

### public void write(int c)

写一个字符。 要写入的字符包含在给定整数值的16个低位中; 忽略16个高位。打算支持高效单字符输出的子类应该重写此方法。

参数：指定要写入的字符

### public void write(char cbuf[])

参数：cbuf 写入的字符数组

### abstract public void write(char cbuf[], int off, int len)

写一个字符数组的一部分。

参数：cbuf 字符数组 off从中开始写字符的偏移量 len 写入的字符数目

### public void write(String str)

参数：str 写入的字符串

### public void write(String str, int off, int len)

写入一个字符串的一部分

参数：str 一个字符串 off从中开始写字符的偏移量 len 写入的字符数目

### public Writer append(CharSequence csq)

将指定的字符序列追加到此writer。根据字符序列csq的toString规范，可能不会附加整个序列。 例如，调用字符缓冲区的toString方法将返回一个子序列，其内容取决于缓冲区的位置和限制

参数：要追加的字符序列。 如果csq为null，则将四个字符“null”附加到此writer

返回：Writer

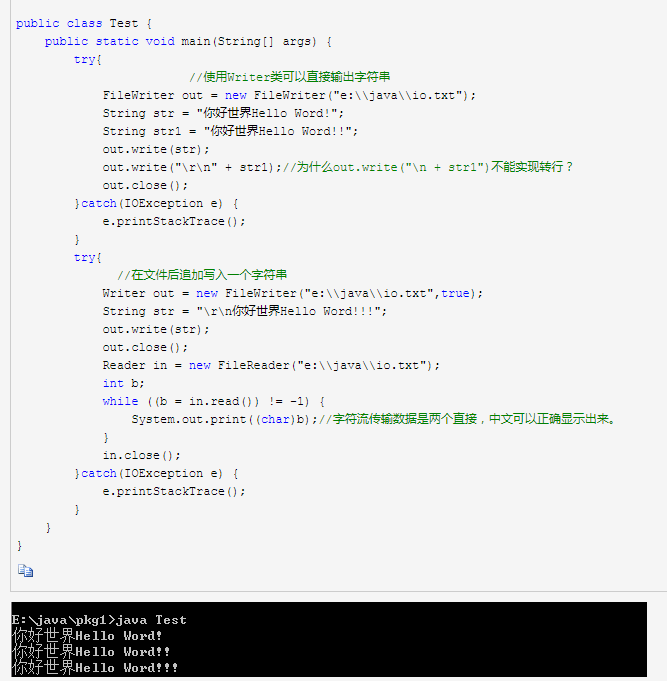
### abstract public void flush()

刷新流。 如果流已从缓冲区中的各种write（）方法保存任何字符，请立即将它们写入其预期目标。 然后，如果该目标是另一个字符或字节流，请将其刷新。 因此，一次flush()调用将刷新Writers和OutputStreams链中的所有缓冲区。（将缓冲区输出）

### abstract public void close()

关闭流，先冲洗它。 关闭流后，进一步的write（）或flush（）调用将导致抛出IOException。 关闭先前关闭的流无效。

## 代码示例





# OutputStreamWriter类

OutputStreamWriter是从字符流到字节流的桥梁：使用指定的{@link java.nio.charset.Charset charset}将写入其中的字符编码为字节。 它使用的字符集可以通过名称指定，也可以明确指定，或者可以接受平台的默认字符集。

每次调用write（）方法都会导致在给定字符上调用编码转换器。 生成的字节在写入底层输出流之前在缓冲区中累积。 可以指定此缓冲区的大小，但默认情况下，它足够大，可用于大多数用途。 请注意，传递给write（）方法的字符不会被缓冲。

为了获得最高效率，请考虑在BufferedWriter中包装OutputStreamWriter，以避免频繁的转换器调用。

## 构造函数

### public OutputStreamWriter(OutputStream out, String charsetName)

创建使用指定charset的OutputStreamWriter。

参数：out 输出流 charsetName 支持的字符集名称

## 方法

### public void write(int c)

写入一个字符

# InputStreamReader类

InputStreamReader是从字节流到字符流的桥接器：它使用指定的{@link java.nio.charset.Charset <code> charset </ code>}读取字节并将其解码为字符。 它使用的字符集可以通过名称指定，也可以明确指定，或者可以接受平台的默认字符集

## 构造函数

### public InputStreamReader(InputStream in)

创建一个使用默认字符集的InputStreamReader

## 方法

### public int read()

读一个字符

# FileReader类

## 构造函数

### public FileReader(String fileName)

参数：读取的文件名称

### public FileReader(File file)

参数：要读取的文件

# FileWriter类

## 构造函数

### public FileWriter(String fileName)

参数：系统单独的文件名字符串

### public FileWriter(String fileName, boolean append)

参数：fileName 系统单独的文件名字符串 append 如果是true，数据将被写入文件的末尾而不是开头

### public FileWriter(File file)

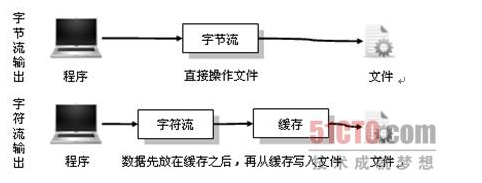
参数：写入的File对象

### public FileWriter(File file, boolean append)

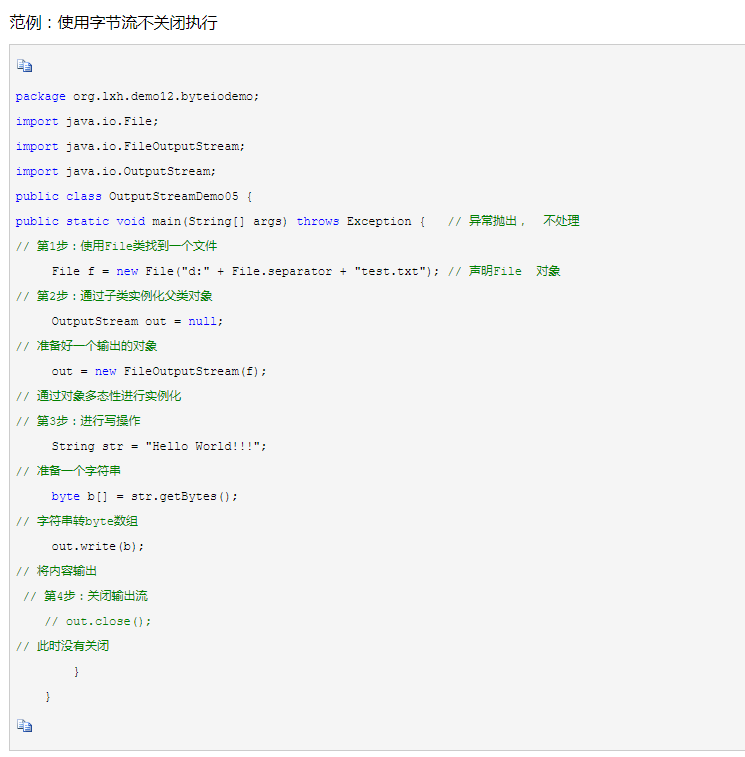
参数：file写入的File对象 append如果是true，数据将被写入文件的末尾而不是开头

# 字节流和字符流的区别

实际上字节流在操作时本身不会用到缓冲区（内存），是文件本身直接操作的，而字符流在操作时使用了缓冲区，通过缓冲区再操作文件，如图：



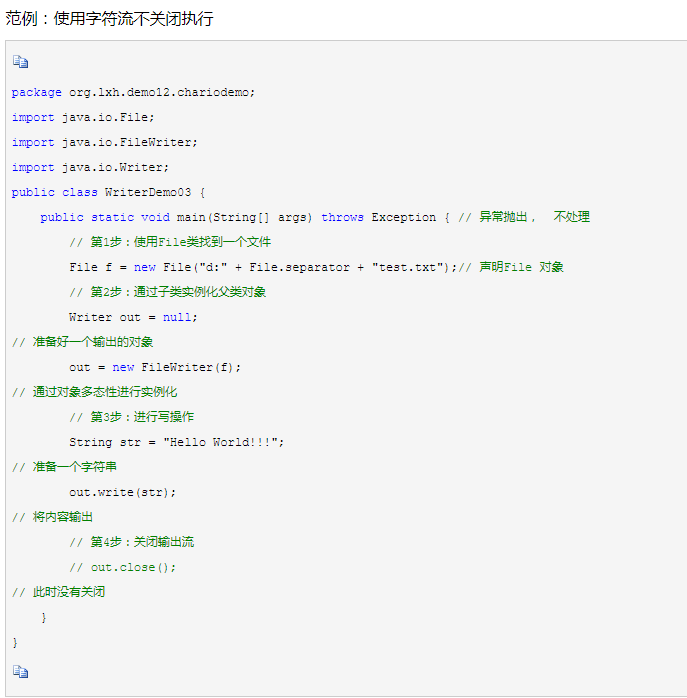
面以两个写文件的操作为主进行比较，但是在操作时字节流和字符流的操作完成之后都不关闭输出流



程序运行结果：



此时没有关闭字节流操作，但是文件中也依然存在了输出的内容，证明字节流是直接操作文件本身的。而下面继续使用字符流完成，再观察效果



程序运行结果：



程序运行后会发现文件中没有任何内容，这是因为字符流操作时使用了缓冲区，而   在关闭字符流时会强制性地将缓冲区中的内容进行输出，但是如果程序没有关闭，则缓冲区中的内容是无法输出的，所以得出结论：字符流使用了缓冲区，而字节流没有使用缓冲区。

缓冲区可以简单地理解为一段内存区域，可以简单地把缓冲区理解为一段特殊的内存。某些情况下，如果一个程序频繁地操作一个资源（如文件或数据库），则性能会很低，此时为了提升性能，就可以将一部分数据暂时读入到内存的一块区域之中，以后直接从此区域中读取数据即可，因为读取内存速度会比较快，这样可以提升程序的性能。

在字符流的操作中，所有的字符都是在内存中形成的，在输出前会将所有的内容暂时保存在内存之中，所以使用了缓冲区暂存数据。如果想在不关闭时也可以将字符流的内容全部输出，则可以使用Writer类中的flush()方法完成。



程序运行结果：



此时，文件中已经存在了内容，更进一步证明内容是保存在缓冲区的。这一点在读者日后的开发中要特别引起注意

我们在开发中还是使用字节流比较好。所有的文件在硬盘或在传输时都是以字节的方式进行的，包括图片等都是按字节的方式存储的，而字符是只有在内存中才会形成，所以在开发中，字节流使用较为广泛。

字节流是最基本的，所有的InputStream和OutputStream的子类都是,主要用在处理二进制数据，它是按字节来处理的，但实际中很多的数据是文本，又提出了字符流的概念，它是按虚拟机的encode来处理，也就是要进行字符集的转化这两个之间通过 InputStreamReader,OutputStreamWriter来关联，实际上是通过byte[]和String来关联在实际开发中出现的汉字问题实际上都是在字符流和字节流之间转化不统一而造成的。

在从字节流转化为字符流时，实际上就是byte[]转化为String时，public String(byte bytes[], String charsetName)，有一个关键的参数字符集编码，通常我们都省略了，那系统就用操作系统的lang而在字符流转化为字节流时，实际上是String转化为byte[]时，byte[]    String.getBytes(String charsetName)，也是一样的道理。

# IO流关闭的顺序

先打开的后关闭，后打开的先关闭。