

第13章

IO流



讲师：宋红康
新浪微博：尚硅谷-宋红康



目录



1

File类的使用

2

IO流原理及流的分类

3

节点流(或文件流)

4

缓冲流

5

转换流

6

标准输入、输出流

目录



7

打印流

8

数据流

9

对象流

10

随机存取文件流

11

NIO.2中Path、Paths、
Files类的使用



13-1 File类的使用



- `java.io.File`类：文件和文件目录路径的抽象表示形式，与平台无关
- `File` 能新建、删除、重命名文件和目录，但 `File` 不能访问文件内容本身。如果需要访问文件内容本身，则需要使用输入/输出流。
- 想要在Java程序中表示一个真实存在的文件或目录，那么必须有一个`File`对象，但是Java程序中的一个`File`对象，可能没有一个真实存在的文件或目录。
- `File`对象可以作为参数传递给流的构造器



- **public File(String pathname)**

以pathname为路径创建File对象，可以是绝对路径或者相对路径，如果pathname是相对路径，则默认当前路径在系统属性user.dir中存储。

- 绝对路径：是一个固定的路径,从盘符开始
- 相对路径：是相对于某个位置开始

- **public File(String parent,String child)**

以parent为父路径，child为子路径创建File对象。

- **public File(File parent,String child)**

根据一个父File对象和子文件路径创建File对象



13.1 File 类的使用：路径分隔符

- 路径中的每级目录之间用一个**路径分隔符**隔开。
- 路径分隔符和系统有关：
 - windows和DOS系统默认使用 “\” 来表示
 - UNIX和URL使用 “/” 来表示
- Java程序支持跨平台运行，因此路径分隔符要慎用。
- 为了解决这个隐患，File类提供了一个常量：
public static final String separator。根据操作系统，动态的提供分隔符。
- 举例：

```
File file1 = new File("d:\\atguigu\\info.txt");  
File file2 = new File("d:" + File.separator + "atguigu" + File.separator + "info.txt");  
File file3 = new File("d:/atguigu");
```




● File类的获取功能

- **public String getAbsolutePath():** 获取绝对路径
- **public String getPath() :** 获取路径
- **public String getName() :** 获取名称
- **public String getParent():** 获取上层文件目录路径。若无，返回null
- **public long length() :** 获取文件长度（即：字节数）。不能获取目录的长度。
- **public long lastModified() :** 获取最后一次的修改时间，毫秒值

- **public String[] list() :** 获取指定目录下的所有文件或者文件目录的名称数组
- **public File[] listFiles() :** 获取指定目录下的所有文件或者文件目录的File数组

● File类的重命名功能

- **public boolean renameTo(File dest):**把文件重命名为指定的文件路径



● File类的判断功能

- `public boolean isDirectory()`：判断是否是文件目录
- `public boolean isFile()`：判断是否是文件
- `public boolean exists()`：判断是否存在
- `public boolean canRead()`：判断是否可读
- `public boolean canWrite()`：判断是否可写
- `public boolean isHidden()`：判断是否隐藏



● File类的创建功能

- `public boolean createNewFile()`：创建文件。若文件存在，则不创建，返回false
- `public boolean mkdir()`：创建文件目录。如果此文件目录存在，就不创建了。如果此文件目录的上层目录不存在，也不创建。
- `public boolean mkdirs()`：创建文件目录。如果上层文件目录不存在，一并创建

注意事项：如果你创建文件或者文件目录没有写盘符路径，那么，默认在项目路径下。

● File类的删除功能

- `public boolean delete()`：删除文件或者文件夹

删除注意事项：

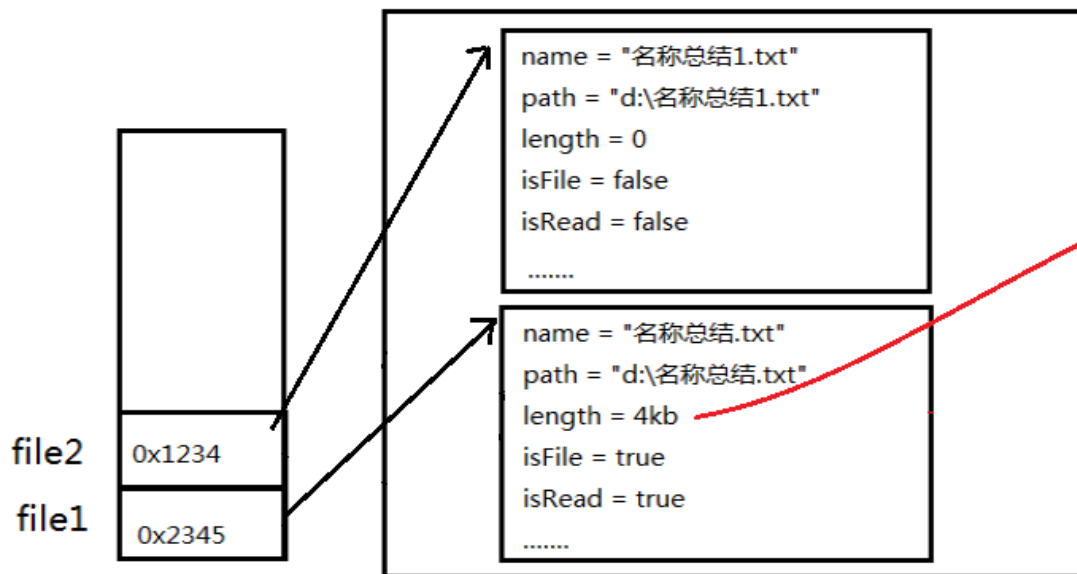
Java中的删除不走回收站。

要删除一个文件目录，请注意该文件目录内不能包含文件或者文件目录



13.1 File 类的使用

```
File file1 = new File("D:\\名词总结.txt");  
File file2 = new File("D:\\名词总结1.txt");
```



D盘

本地磁盘 (D:) ▾				
工具(T) 帮助(H)				
共享 ▾ 新建文件夹				
名称	修改日期	类型	大小	
名词总结.txt	2016/11/19 11:19	TXT 文件	4 KB	

当硬盘中真有一个真实的文件或目录存在时，创建File对象时，各个属性会显式赋值。

当硬盘中没有真实的文件或目录对应时，那么创建对象时，除了指定的目录和路径之外，其他的属性都是取成员变量的默认值。



13.1 File 类的使用：常用方法

```
File dir1 = new File("D:/IOTest/dir1");
if (!dir1.exists()) { // 如果D:/IOTest/dir1不存在，就创建为目录
    dir1.mkdir();
}
// 创建以dir1为父目录,名为"dir2"的File对象
File dir2 = new File(dir1, "dir2");
if (!dir2.exists()) { // 如果还不存在，就创建为目录
    dir2.mkdirs();
}
File dir4 = new File(dir1, "dir3/dir4");
if (!dir4.exists()) {
    dir4.mkdirs();
}
// 创建以dir2为父目录,名为"test.txt"的File对象
File file = new File(dir2, "test.txt");
if (!file.exists()) { // 如果还不存在，就创建为文件
    file.createNewFile();
}
```



练习

1. 利用File构造器，new 一个文件目录file
 - 1)在其中创建多个文件和目录
 - 2)编写方法，实现删除file中指定文件的操作
2. 判断指定目录下是否有后缀名为.jpg的文件，如果有，就输出该文件名称
3. 遍历指定目录所有文件名称，包括子文件目录中的文件。
 - 拓展1：并计算指定目录占用空间的大小
 - 拓展2：删除指定文件目录及其下的所有文件



13-2 IO流原理及流的分类



Google I/O 寓为 “开放中创新”
(Innovation in the Open)

Input/Output
二进制1,0



Java IO原理

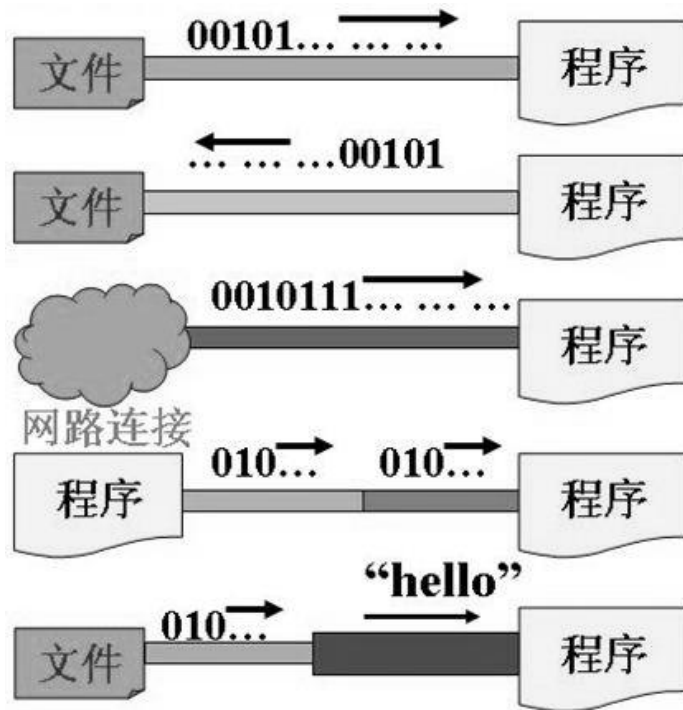
- I/O是Input/Output的缩写， I/O技术是非常实用的技术，用于**处理设备之间的数据传输**。如读/写文件，网络通讯等。
- Java程序中，对于数据的输入/输出操作以 **“流(stream)”** 的方式进行。
- java.io包下提供了各种“流”类和接口，用以获取不同种类的数据，并通过**标准的方法**输入或输出数据。



Java IO原理

●**输入input:** 读取外部数据（磁盘、光盘等存储设备的数据）到程序（内存）中。

●**输出output:** 将程序（内存）数据输出到磁盘、光盘等存储设备中。



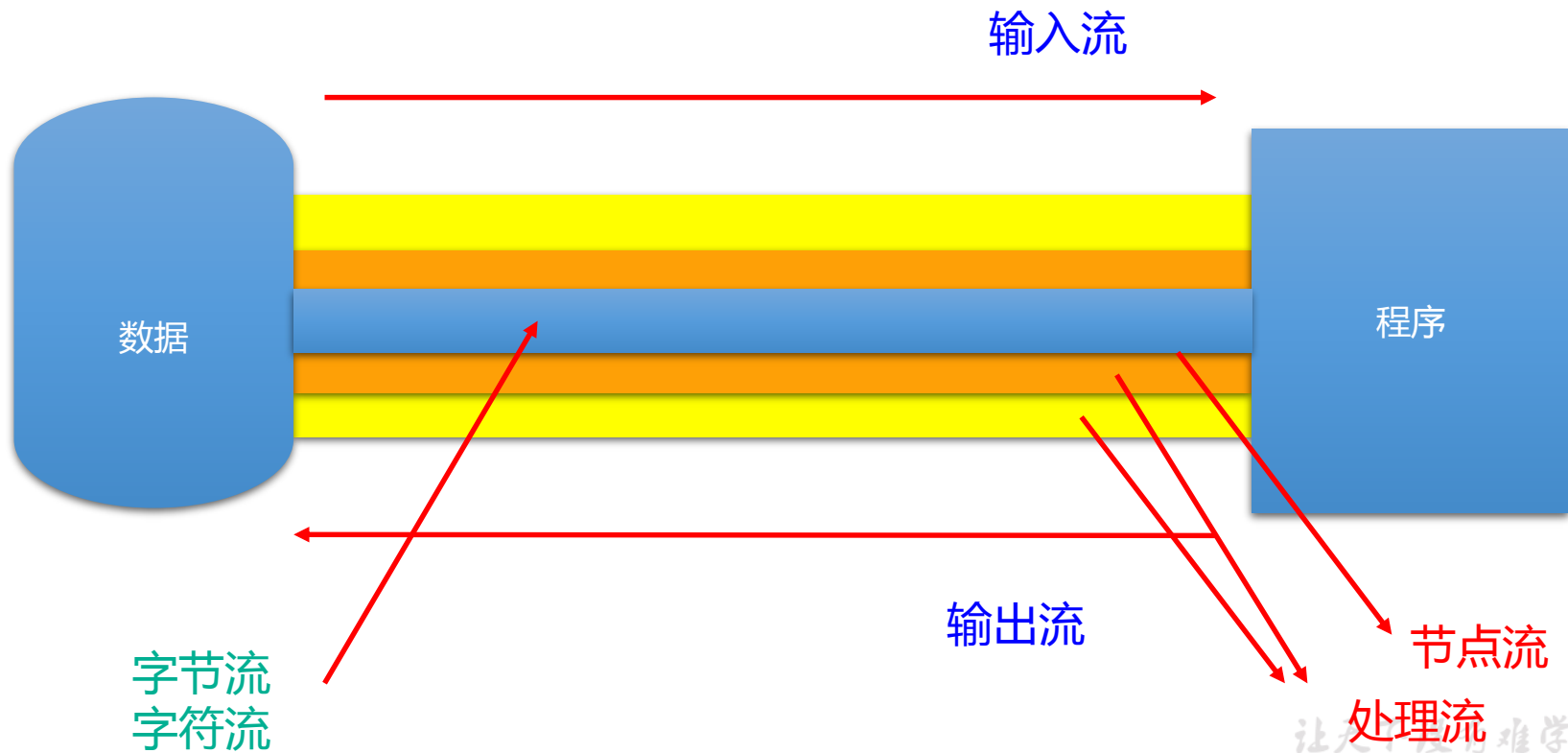


流的分类

- 按操作**数据单位**不同分为：字节流(8 bit)，字符流(16 bit)
- 按数据流的**流向**不同分为：输入流，输出流
- 按流的**角色**的不同分为：节点流，处理流

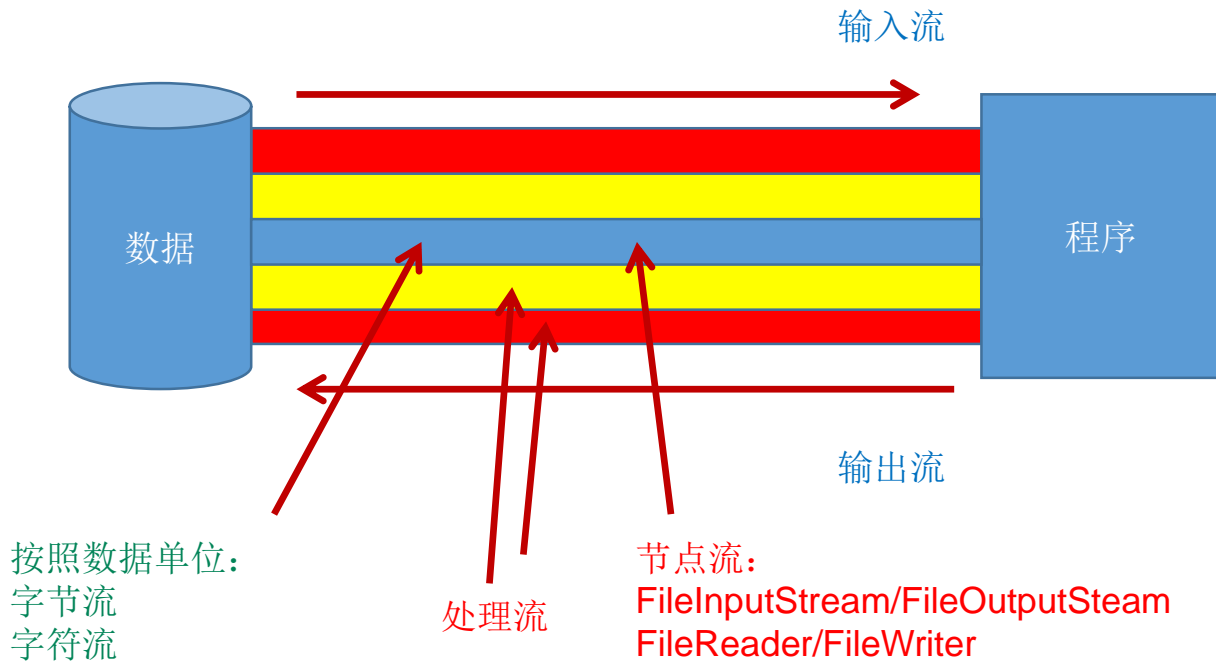
(抽象基类)	字节流	字符流
输入流	InputStream	Reader
输出流	OutputStream	Writer

1. Java的IO流共涉及40多个类，实际上非常规则，都是从如下4个抽象基类派生的。
2. 由这四个类派生出来的子类名称都是以其父类名作为子类名后缀。





流的分类





IO 流体系

分类	字节输入流	字节输出流	字符输入流	字符输出流
抽象基类	InputStream	OutputStream	Reader	Writer
<u>访问文件</u>	<u>FileInputStream</u>	<u>FileOutputStream</u>	<u>FileReader</u>	<u>FileWriter</u>
访问数组	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
访问管道	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
访问字符串			StringReader	StringWriter
<u>缓冲流</u>	<u>BufferedInputStream</u>	<u>BufferedOuputStream</u>	<u>BufferedReader</u>	<u>BufferedWriter</u>
<u>转换流</u>			<u>InputStreamReader</u>	<u>OutputStreamWriter</u>
<u>对象流</u>	<u>ObjectInputStream</u>	<u>ObjectOutputStream</u>		
	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
打印流		PrintStream		PrintWriter
推回输入流	PushbackInputStream		PushbackReader	
特殊流	DataInputStream	DataOutputStream		

字节流（文件流）←

处理流

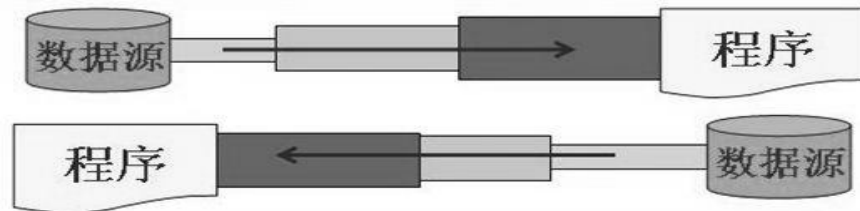


节点流和处理流

- 节点流：直接从数据源或目的地读写数据



- 处理流：不直接连接到数据源或目的地，而是“连接”在已存在的流（节点流或处理流）之上，通过对数据的处理为程序提供更为强大的读写功能。





InputStream & Reader

- InputStream 和 Reader 是所有输入流的基类。
- InputStream（典型实现：**FileInputStream**）
 - int read()
 - **int read(byte[] b)**
 - int read(byte[] b, int off, int len)
- Reader（典型实现：**FileReader**）
 - int read()
 - **int read(char [] c)**
 - int read(char [] c, int off, int len)
- 程序中打开的文件 IO 资源不属于内存里的资源，垃圾回收机制无法回收该资源，所以应该**显式关闭文件 IO 资源**。
- **FileInputStream** 从文件系统中的某个文件中获得输入字节。**FileInputStream** 用于读取非文本数据之类的原始字节流。要读取字符流，需要使用 **FileReader**



InputStream

- **int read()**

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，则返回值 -1。

- **int read(byte[] b)**

从此输入流中将最多 b.length 个字节的数据读入一个 byte 数组中。如果因为已经到达流末尾而没有可用的字节，则返回值 -1。否则以整数形式返回实际读取的字节数。

- **int read(byte[] b, int off, int len)**

将输入流中最多 len 个数据字节读入 byte 数组。尝试读取 len 个字节，但读取的字节也可能小于该值。以整数形式返回实际读取的字节数。如果因为流位于文件末尾而没有可用的字节，则返回值 -1。

- **public void close() throws IOException**

关闭此输入流并释放与该流关联的所有系统资源。



Reader

- **int read()**

读取单个字符。作为整数读取的字符，范围在 0 到 65535 之间 (0x00-0xffff)（2个字节的Unicode码），如果已到达流的末尾，则返回 -1

- **int read(char[] cbuf)**

将字符读入数组。如果已到达流的末尾，则返回 -1。否则返回本次读取的字符数。

- **int read(char[] cbuf,int off,int len)**

将字符读入数组的某一部分。存到数组cbuf中，从off处开始存储，最多读len个字符。如果已到达流的末尾，则返回 -1。否则返回本次读取的字符数。

- **public void close() throws IOException**

关闭此输入流并释放与该流关联的所有系统资源。



OutputStream & Writer

- OutputStream 和 Writer 也非常相似：
 - `void write(int b/int c);`
 - `void write(byte[] b/char[] cbuf);`
 - `void write(byte[] b/char[] buff, int off, int len);`
 - `void flush();`
 - `void close();` 需要先刷新，再关闭此流
- 因为字符流直接以字符作为操作单位，所以 Writer 可以用字符串来替换字符数组，即以 String 对象作为参数
 - `void write(String str);`
 - `void write(String str, int off, int len);`
- FileOutputStream 从文件系统中的某个文件中获得输出字节。FileOutputStream 用于写出非文本数据之类的原始字节流。要写出字符流，需要使用 FileWriter



OutputStream

- **void write(int b)**

将指定的字节写入此输出流。write 的常规协定是：向输出流写入一个字节。要写入的字节是参数 b 的八个低位。b 的 24 个高位将被忽略。即写入 0~255 范围的。

- **void write(byte[] b)**

将 b.length 个字节从指定的 byte 数组写入此输出流。write(b) 的常规协定是：应该与调用 write(b, 0, b.length) 的效果完全相同。

- **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。

- **public void flush() throws IOException**

刷新此输出流并强制写出所有缓冲的输出字节，调用此方法指示应将这些字节立即写入它们预期的目标。

- **public void close() throws IOException**

关闭此输出流并释放与该流关联的所有系统资源。



Writer

- **void write(int c)**

写入单个字符。要写入的字符包含在给定整数值的 16 个低位中，16 高位被忽略。即写入 0 到 65535 之间的 Unicode 码。

- **void write(char[] cbuf)**

写入字符数组。

- **void write(char[] cbuf, int off, int len)**

写入字符数组的某一部分。从 off 开始，写入 len 个字符

- **void write(String str)**

写入字符串。

- **void write(String str, int off, int len)**

写入字符串的某一部分。

- **void flush()**

刷新该流的缓冲，则立即将它们写入预期目标。

- **public void close() throws IOException**

关闭此输出流并释放与该流关联的所有系统资源。



13-3 节点流(或文件流)



13.3 节点流(或文件流)

读取文件

1. 建立一个流对象，将已存在的一个文件加载进流。

➤ **`FileReader fr = new FileReader(new File("Test.txt"));`**

2. 创建一个临时存放数据的数组。

➤ **`char[] ch = new char[1024];`**

3. 调用流对象的读取方法将流中的数据读入到数组中。

➤ **`fr.read(ch);`**

4. 关闭资源。

➤ **`fr.close();`**



13.3 节点流(或文件流)

```
FileReader fr = null;
try {
    fr = new FileReader(new File("c:\\test.txt"));
    char[] buf = new char[1024];
    int len;
    while ((len = fr.read(buf)) != -1) {
        System.out.print(new String(buf, 0, len));
    }
} catch (IOException e) {
    System.out.println("read-Exception :" + e.getMessage());
} finally {
    if (fr != null) {
        try {
            fr.close();
        } catch (IOException e) {
            System.out.println("close-Exception :" + e.getMessage());
        }
    }
}
```



13.3 节点流(或文件流)

写入文件

1.创建流对象，建立数据存放文件

➤ **FileWriter fw = new FileWriter(new File("Test.txt"));**

2.调用流对象的写入方法，将数据写入流

➤ **fw.write("atguigu-songhongkang");**

3.关闭流资源，并将流中的数据清空到文件中。

➤ **fw.close();**



13.3 节点流(或文件流)

```
FileWriter fw = null;
try {
    fw = new FileWriter(new File("Test.txt"));
    fw.write("atguigu-songhongkang");
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (fw != null)
        try {
            fw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
}
```



13.3 节点流(或文件流): 注意点

- 定义文件路径时, 注意: 可以用 “/” 或者 “\\”。
- 在写入一个文件时, 如果使用构造器 `FileOutputStream(file)`, 则目录下有同名文件将被覆盖。
- 如果使用构造器 `FileOutputStream(file,true)`, 则目录下的同名文件不会被覆盖, 在文件内容末尾追加内容。
- 在读取文件时, 必须保证该文件已存在, 否则报异常。
- 字节流操作字节, 比如: .mp3, .avi, .rmvb, mp4, .jpg, .doc, .ppt
- 字符流操作字符, 只能操作普通文本文件。最常见的文本文件: .txt, .java, .c, .cpp 等语言的源代码。尤其注意.doc, excel, ppt 这些不是文本文件。



13-4 缓冲流



- 为了提高数据读写的速度，Java API提供了带缓冲功能的流类，在使用这些流类时，会创建一个内部缓冲区数组，缺省使用8192个字节(8Kb)的缓冲区。

```
public
class BufferedInputStream extends FilterInputStream {

    private static int DEFAULT_BUFFER_SIZE = 8192;
```

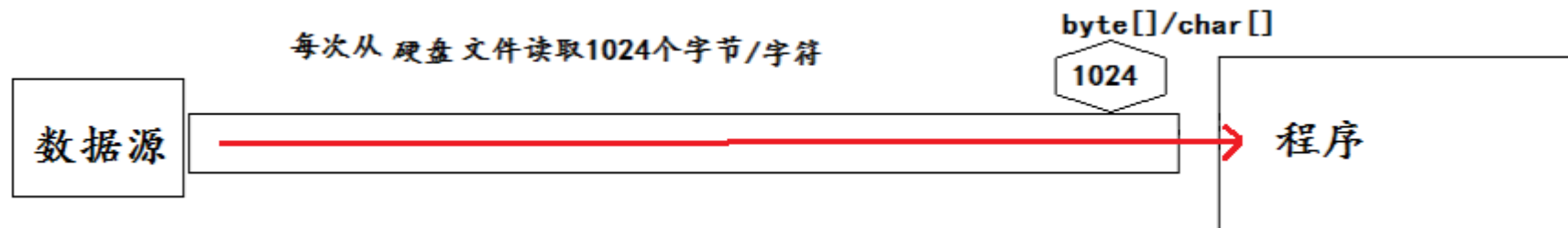
- 缓冲流要“套接”在相应的节点流之上，根据数据操作单位可以把缓冲流分为：
 - **BufferedInputStream** 和 **BufferedOutputStream**
 - **BufferedReader** 和 **BufferedWriter**



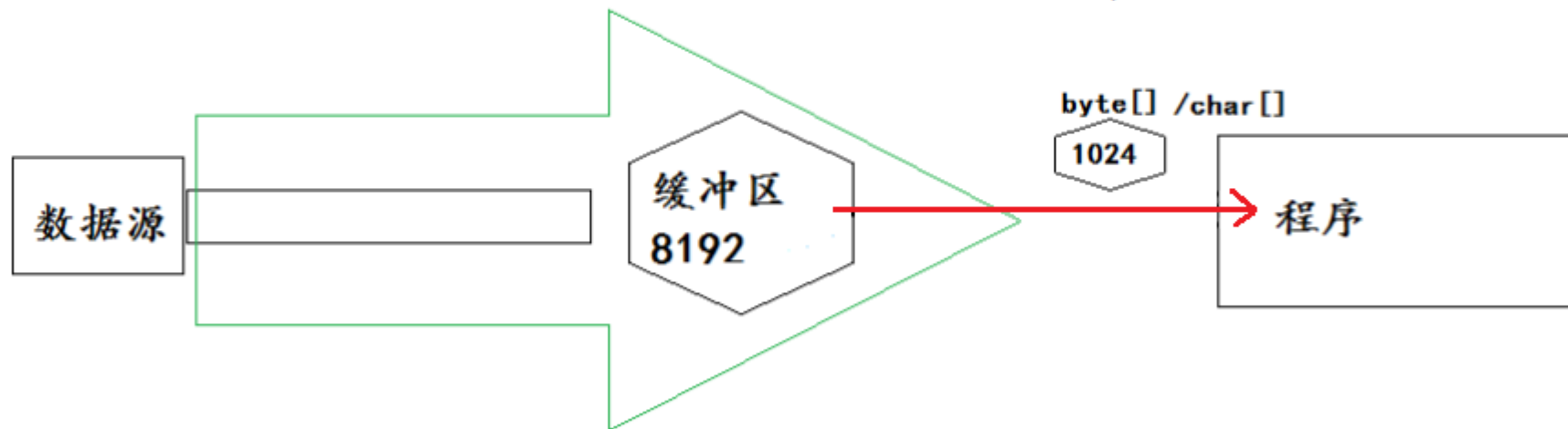
- 当读取数据时，数据按块读入缓冲区，其后的读操作则直接访问缓冲区
- 当使用`BufferedInputStream`读取字节文件时，`BufferedInputStream`会一次性从文件中读取8192个(8Kb)，存在缓冲区中，直到缓冲区装满了，才重新从文件中读取下一个8192个字节数组。
- 向流中写入字节时，不会直接写到文件，先写到缓冲区中直到缓冲区写满，`BufferedOutputStream`才会把缓冲区中的数据一次性写到文件里。使用方法`flush()`可以强制将缓冲区的内容全部写入输出流
- 关闭流的顺序和打开流的顺序相反。只要关闭最外层流即可，关闭最外层流也会相应关闭内层节点流
- `flush()`方法的使用：手动将buffer中内容写入文件
- 如果是带缓冲区的流对象的`close()`方法，不但会关闭流，还会在关闭流之前刷新缓冲区，关闭后不能再写出



13.4 处理流之一：缓冲流



先把文件的数据缓冲8192字节/字符的缓冲区的内存中，然后从缓冲区内存中读取1024个字节/字符，从内存读取要比从硬盘读取的效率高





```
BufferedReader br = null;
BufferedWriter bw = null;
try {
    // 创建缓冲流对象：它是处理流，是对节点流的包装
    br = new BufferedReader(new FileReader("d:\\IOTest\\source.txt"));
    bw = new BufferedWriter(new FileWriter("d:\\IOTest\\dest.txt"));
    String str;
    while ((str = br.readLine()) != null) { // 一次读取字符文本文件的一行字符
        bw.write(str); // 一次写入一行字符串
        bw.newLine(); // 写入行分隔符
    }
    bw.flush(); // 刷新缓冲区
} catch (IOException e) {
    e.printStackTrace();
} finally {
    // 关闭IO流对象
    try {
        if (bw != null) {
            bw.close(); // 关闭过滤流时,会自动关闭它所包装的底层节点流
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        if (br != null) {
            br.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```



练习

1. 分别使用节点流：FileInputStream、FileOutputStream和缓冲流：BufferedInputStream、BufferedOutputStream实现文本文件/图片/视频文件的复制。并比较二者在数据复制方面的效率
2. 实现图片加密操作。

提示：

```
int b = 0;
while((b = fis.read()) != -1){
    fos.write(b ^ 5);
}
```

3. 获取文本上每个字符出现的次数

提示：遍历文本的每一个字符；字符及出现的次数保存在Map中；将Map中数据写入文件



13-5 转换流



- 转换流提供了在字节流和字符流之间的转换
- Java API提供了两个转换流：
 - **InputStreamReader**: 将InputStream转换为Reader
 - **OutputStreamWriter**: 将Writer转换为OutputStream
- 字节流中的数据都是字符时，转成字符流操作更高效。
- 很多时候我们使用转换流来处理文件乱码问题。实现编码和解码的功能。



InputStreamReader

- 实现将字节的输入流按指定字符集转换为字符的输入流。
- 需要和InputStream “套接”。

- 构造器

- **public InputStreamReader(InputStream in)**

- **public InputStreamReader(InputStream in, String charsetName)**

如： Reader isr = new InputStreamReader(System.in, "gbk");

指定字符集

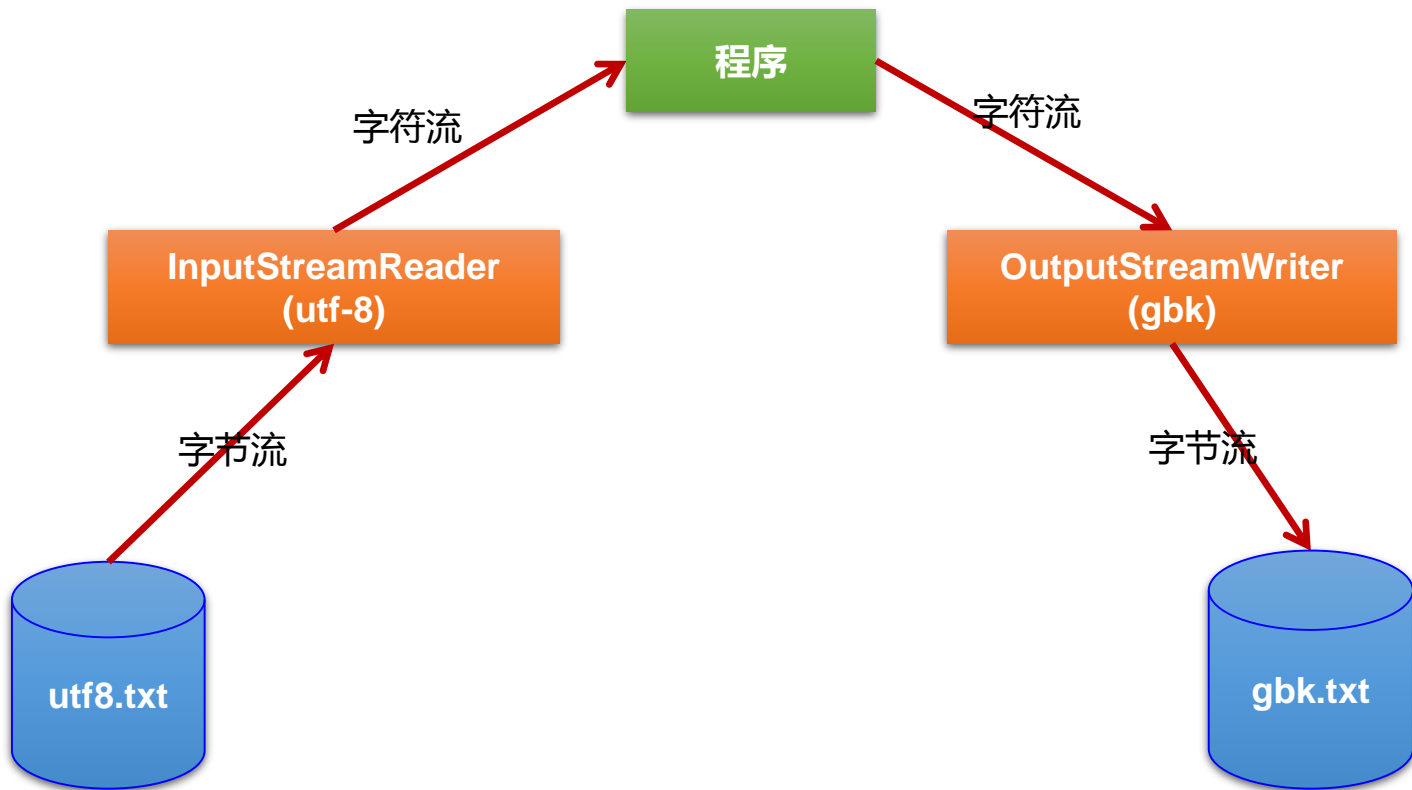


OutputStreamWriter

- 实现将字符的输出流按指定字符集转换为字节的输出流。
- 需要和OutputStream “套接”。
- 构造器
 - **public OutputStreamWriter(OutputStream out)**
 - **public OutputStreamWriter(OutputStream out, String charsetName)**



13.5 处理流之二：转换流





13.5 处理流之二：转换流

```
public void testMyInput() throws Exception {  
    FileInputStream fis = new FileInputStream("dbcp.txt");  
    FileOutputStream fos = new FileOutputStream("dbcp5.txt");  
  
    InputStreamReader isr = new InputStreamReader(fis, "GBK");  
    OutputStreamWriter osw = new OutputStreamWriter(fos, "GBK");  
  
    BufferedReader br = new BufferedReader(isr);  
    BufferedWriter bw = new BufferedWriter(osw);  
  
    String str = null;  
    while ((str = br.readLine()) != null) {  
        bw.write(str);  
        bw.newLine();  
        bw.flush();  
    }  
    bw.close();  
    br.close();  
}
```



补充：字符编码

● 编码表的由来

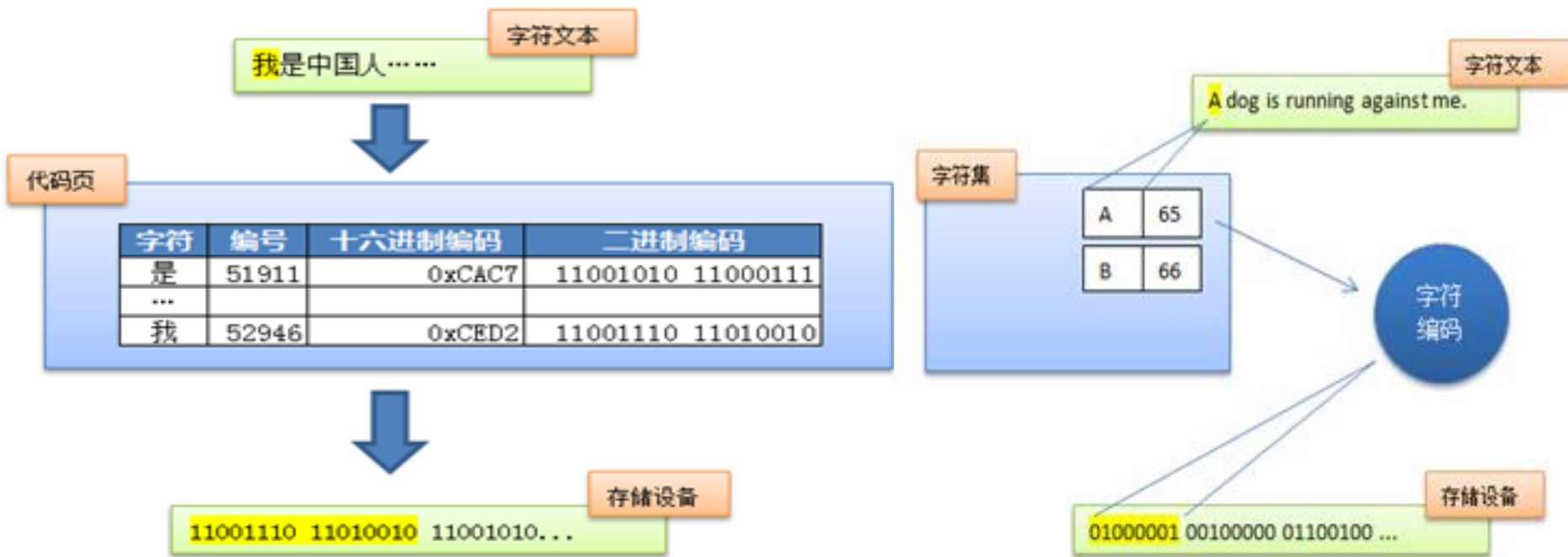
计算机只能识别二进制数据，早期由来是电信号。为了方便应用计算机，让它可以识别各个国家的文字。就将各个国家的文字用数字来表示，并一一对应，形成一张表。这就是编码表。

● 常见的编码表

- **ASCII**：美国标准信息交换码。
 - ✓ 用一个字节的7位可以表示。
- **ISO8859-1**：拉丁码表。欧洲码表
 - ✓ 用一个字节的8位表示。
- **GB2312**：中国的中文编码表。最多两个字节编码所有字符
- **GBK**：中国的中文编码表升级，融合了更多的中文文字符号。最多两个字节编码
- **Unicode**：国际标准码，融合了目前人类使用的所有字符。为每个字符分配唯一的字符码。所有的文字都用两个字节来表示。
- **UTF-8**：变长的编码方式，可用1-4个字节来表示一个字符。



13.5 处理流之二：转换流



- 在Unicode出现之前，所有的字符集都是和具体编码方案绑定在一起的（即字符集≈编码方式），都是直接将字符和最终字节流绑定死了。
- GBK等双字节编码方式，用最高位是1或0表示两个字节和一个字节。



补充：字符编码

- **Unicode**不完美，这里就有三个问题，一个是，我们已经知道，英文字母只用一个字节表示就够了，第二个问题是如何才能区别**Unicode**和**ASCII**？计算机怎么知道两个字节表示一个符号，而不是分别表示两个符号呢？第三个，如果和**GBK**等双字节编码方式一样，用最高位是1或0表示两个字节和一个字节，就少了很多值无法用于表示字符，不够表示所有字符。**Unicode**在很长一段时间内无法推广，直到互联网的出现。
- 面向传输的众多 **UTF**（**UCS Transfer Format**）标准出现了，顾名思义，**UTF-8**就是每次8个位传输数据，而**UTF-16**就是每次16个位。这是为传输而设计的编码，并使编码无国界，这样就可以显示全世界上所有文化的字符了。
- **Unicode**只是定义了一个庞大的、全球通用的字符集，并为每个字符规定了唯一确定的编号，具体存储成什么样的字节流，取决于字符编码方案。推荐的**Unicode**编码是**UTF-8**和**UTF-16**。



补充：字符编码

Unicode符号范围 | UTF-8编码方式
(十六进制) | (二进制)

0000 0000-0000 007F | 0xxxxxxx (兼容原来的ASCII)

0000 0080-0000 07FF | 110xxxxx 10xxxxxx

0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx

0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

尚

Unicode编码值：23578 十六进制 5C1A 二进制 0101 1100 0001 1010

1110xxxx 10xx xxxx 10xx xxxx
1110 0101 1011 0000 1001 1010

UTF-8编码：

1110 0101 1011 0000 1001 1010

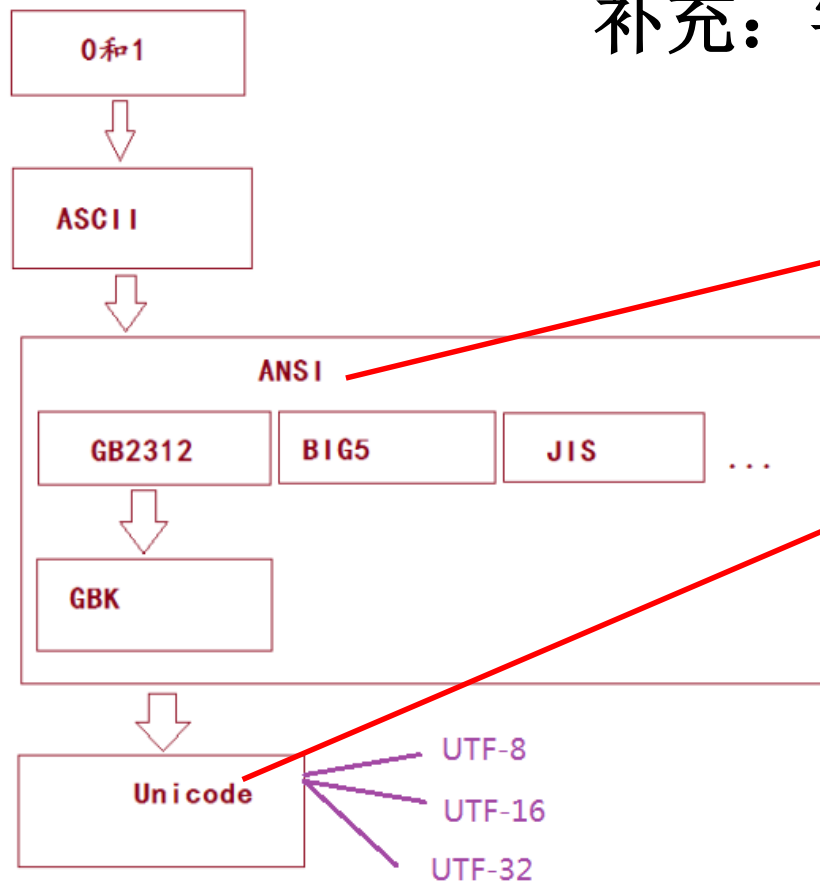
e5 b0 9a

[-27, -80, -102]

学的技术



补充：字符编码



ANSI编码，通常指的是平台的默认编码，例如英文操作系统中是ISO-8859-1，中文系统是GBK

Unicode字符集只是定义了字符的集合和唯一编号，Unicode编码，则是对UTF-8、UCS-2/UTF-16等具体编码方案的统称而已，并不是具体的编码方案。



补充：字符编码

- 编码：字符串 → 字节数组
- 解码：字节数组 → 字符串
- 转换流的编码应用
 - 可以将字符按指定编码格式存储
 - 可以对文本数据按指定编码格式来解读
 - 指定编码表的动作由构造器完成



13-6 标准输入、输出流



- **System.in**和**System.out**分别代表了系统标准的输入和输出设备
- 默认输入设备是：键盘，输出设备是：显示器
- **System.in**的类型是**InputStream**
- **System.out**的类型是**PrintStream**，其是**OutputStream**的子类
FilterOutputStream 的子类
- 重定向：通过**System**类的**setIn**，**setOut**方法对默认设备进行改变。
 - `public static void setIn(InputStream in)`
 - `public static void setOut(PrintStream out)`



例 题

从键盘输入字符串，要求将读取到的整行字符串转成大写输出。然后继续进行输入操作，直至当输入“e”或者“exit”时，退出程序。



13.6 处理流之三：标准输入、输出流(了解)

```
System.out.println("请输入信息(退出输入e或exit):");
// 把"标准"输入流(键盘输入)这个字节流包装成字符流,再包装成缓冲流
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String s = null;
try {
    while ((s = br.readLine()) != null) { // 读取用户输入的一行数据 --> 阻塞程序
        if ("e".equalsIgnoreCase(s) || "exit".equalsIgnoreCase(s)) {
            System.out.println("安全退出!!");
            break;
        }
        // 将读取到的整行字符串转成大写输出
        System.out.println("-->:" + s.toUpperCase());
        System.out.println("继续输入信息");
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (br != null) {
            br.close(); // 关闭过滤流时,会自动关闭它包装的底层节点流
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```





练习

Create a program named MyInput.java: Contain the methods for reading int, double, float, boolean, short, byte and String values from the keyboard.

```
Scanner s = new Scanner(System.in);  
s.nextInt();  
s.next();
```

```
class MyInput{  
    Scanner s = new Scanner(System.in);  
  
    public String readString(){  
        return s.next();  
    }  
    ....  
}
```





13-7 打印流



- 实现将**基本数据类型**的数据格式转化为**字符串**输出
- 打印流：**PrintStream**和**PrintWriter**
 - 提供了一系列重载的print()和println()方法，用于多种数据类型的输出
 - PrintStream和PrintWriter的输出不会抛出IOException异常
 - PrintStream和PrintWriter有自动flush功能
 - PrintStream 打印的所有字符都使用平台的默认字符编码转换为字节。
在需要写入字符而不是写入字节的情况下，应该使用 **PrintWriter** 类。
 - System.out返回的是PrintStream的实例



13.7 处理流之四：打印流(了解)

```
PrintStream ps = null;
try {
    FileOutputStream fos = new FileOutputStream(new File("D:\\IO\\text.txt"));
    // 创建打印输出流, 设置为自动刷新模式(写入换行符或字节 '\n' 时都会刷新输出缓冲区)
    ps = new PrintStream(fos, true);
    if (ps != null) { // 把标准输出流(控制台输出)改成文件
        System.setOut(ps);
    }
    for (int i = 0; i <= 255; i++) { // 输出ASCII字符
        System.out.print((char) i);
        if (i % 50 == 0) { // 每50个数据一行
            System.out.println(); // 换行
        }
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} finally {
    if (ps != null) {
        ps.close();
    }
}
```



13-8 数据流



- 为了方便地操作Java语言的基本数据类型和String的数据，可以使用数据流。
- 数据流有两个类：(用于读取和写出基本数据类型、String类的数据)
 - **DataInputStream** 和 **DataOutputStream**
 - 分别“套接”在 **InputStream** 和 **OutputStream** 子类的流上

● DataInputStream中的方法

boolean readBoolean()

char readChar()

double readDouble()

long readLong()

String readUTF()

byte readByte()

float readFloat()

short readShort()

int readInt()

void readFully(byte[] b)

● DataOutputStream中的方法

- 将上述的方法的read改为相应的write即可。



13.8 处理流之五：数据流(了解)

```
DataOutputStream dos = null;
try { // 创建连接到指定文件的数据输出流对象
    dos = new DataOutputStream(new FileOutputStream("destData.dat"));
    dos.writeUTF("我爱北京天安门"); // 写UTF字符串
    dos.writeBoolean(false); // 写入布尔值
    dos.writeLong(1234567890L); // 写入长整数
    System.out.println("写文件成功!");
} catch (IOException e) {
    e.printStackTrace();
} finally { // 关闭流对象
    try {
        if (dos != null) {
            // 关闭过滤流时,会自动关闭它包装的底层节点流
            dos.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```



13.8 处理流之五：数据流(了解)

```
DataInputStream dis = null;
try {
    dis = new DataInputStream(new FileInputStream("destData.dat"));
    String info = dis.readUTF();
    boolean flag = dis.readBoolean();
    long time = dis.readLong();
    System.out.println(info);
    System.out.println(flag);
    System.out.println(time);
} catch (Exception e) {
    e.printStackTrace();
} finally {
    if (dis != null) {
        try {
            dis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```




13-9 对象流



处理流之六：对象流

- **ObjectInputStream和ObjectOutputStream**

- 用于存储和读取基本数据类型数据或对象的处理流。它的强大之处就是可以把Java中的对象写入到数据源中，也能把对象从数据源中还原回来。
- 序列化：用ObjectOutputStream类保存基本类型数据或对象的机制
- 反序列化：用ObjectInputStream类读取基本类型数据或对象的机制
- ObjectOutputStream和ObjectInputStream不能序列化static和transient修饰的成员变量



对象的序列化

- **对象序列化机制** 允许把内存中的Java对象转换成平台无关的二进制流，从而允许把这种二进制流持久地保存在磁盘上，或通过网络将这种二进制流传输到另一个网络节点。//当其它程序获取了这种二进制流，就可以恢复成原来的Java对象
- 序列化的好处在于可将任何实现了 **Serializable** 接口的对象转化为 **字节数据**，使其在保存和传输时可被还原
- 序列化是 **RMI**（**Remote Method Invoke** – 远程方法调用）过程的参数和返回值都必须实现的机制，而 **RMI** 是 **JavaEE** 的基础。因此序列化机制是 **JavaEE** 平台的基础
- 如果需要让某个对象支持序列化机制，则必须让对象所属的类及其属性是可序列化的，为了让某个类是可序列化的，该类必须实现如下两个接口之一。
否则，会抛出 **NotSerializableException** 异常
 - **Serializable**
 - **Externalizable**



对象的序列化

- 凡是实现Serializable接口的类都有一个表示序列化版本标识符的静态变量：
 - **private static final long serialVersionUID;**
 - serialVersionUID用来表明类的不同版本间的兼容性。简言之，其目的是以序列化对象进行版本控制，有关各版本反序列化时是否兼容。
 - 如果类没有显示定义这个静态常量，它的值是Java运行时环境根据类的内部细节自动生成的。若类的实例变量做了修改，serialVersionUID 可能发生变化。故建议，显式声明。
- 简单来说，Java的序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性的。在进行反序列化时，JVM会把传来的字节流中的serialVersionUID与本地相应实体类的serialVersionUID进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常。(InvalidCastException)



使用对象流序列化对象

- 若某个类实现了 `Serializable` 接口，该类的对象就是可序列化的：
 - 创建一个 `ObjectOutputStream`
 - 调用 `ObjectOutputStream` 对象的 `writeObject(对象)` 方法输出可序列化对象
 - 注意写出一次，操作`flush()`一次
- 反序列化
 - 创建一个 `ObjectInputStream`
 - 调用 `readObject()` 方法读取流中的对象
- 强调：如果某个类的属性不是基本数据类型或 `String` 类型，而是另一个引用类型，那么这个引用类型必须是可序列化的，否则拥有该类型的 `Field` 的类也不能序列化



//序列化：将对象写入到磁盘或者进行网络传输。

//要求对象必须实现序列化

```
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("data.txt"));
Person p = new Person("韩梅梅", 18, "中华大街", new Pet());
oos.writeObject(p);
oos.flush();
oos.close();
```

//反序列化：将磁盘中的对象数据源读出。

```
ObjectInputStream ois = new ObjectInputStream(new FileInputStream("data.txt"));
Person p1 = (Person)ois.readObject();
System.out.println(p1.toString());
ois.close();
```



谈谈你对**java.io.Serializable**接口的理解，我们知道它用于序列化，是空方法接口，还有其它认识吗？

- 实现了**Serializable**接口的对象，可将它们转换成一系列字节，并可在以后完全恢复回原来的样子。这一过程亦可通过网络进行。这意味着序列化机制能自动补偿操作系统间的差异。换句话说，可以先在**Windows**机器上创建一个对象，对其序列化，然后通过网络发给一台**Unix**机器，然后在那里准确无误地重新“装配”。不必关心数据在不同机器上如何表示，也不必关心字节的顺序或者其他任何细节。
- 由于大部分作为参数的类如**String**、**Integer**等都实现了**java.io.Serializable**的接口，也可以利用多态的性质，作为参数使接口更灵活。



13-10 随机存取文件流



RandomAccessFile 类

- **RandomAccessFile** 声明在 `java.io` 包下，但直接继承于 `java.lang.Object` 类。并且它实现了 `DataInput`、`DataOutput` 这两个接口，也就意味着这个类既可以读也可以写。
- **RandomAccessFile** 类支持“随机访问”的方式，程序可以直接跳到文件的任意地方来读、写文件
 - 支持只访问文件的部分内容
 - 可以向已存在的文件后追加内容
- **RandomAccessFile** 对象包含一个记录指针，用以标示当前读写处的位置。
RandomAccessFile 类对象可以自由移动记录指针：
 - **`long getFilePointer()`**: 获取文件记录指针的当前位置
 - **`void seek(long pos)`**: 将文件记录指针定位到 `pos` 位置



RandomAccessFile 类

●构造器

- public RandomAccessFile([File](#) file, [String](#) mode)
- public RandomAccessFile([String](#) name, [String](#) mode)

●创建 RandomAccessFile 类实例需要指定一个 mode 参数，该参数指定 RandomAccessFile 的访问模式：

- **r**: 以只读方式打开
 - **rw**: 打开以便读取和写入
 - **rwd**: 打开以便读取和写入；同步文件内容的更新
 - **rws**: 打开以便读取和写入；同步文件内容和元数据的更新
- 如果模式为只读r。则不会创建文件，而是会去读取一个已经存在的文件，如果读取的文件不存在则会出现异常。 如果模式为rw读写。如果文件不存在则会去创建文件，如果存在则不会创建。



RandomAccessFile 类

我们可以用RandomAccessFile这个类，来实现一个多线程断点下载的功能，用过下载工具的朋友们都知道，下载前都会建立两个临时文件，一个是与被下载文件大小相同的空文件，另一个是记录文件指针的位置文件，每次暂停的时候，都会保存上一次的指针，然后断点下载的时候，会继续从上一次的地方下载，从而实现断点下载或上传的功能，有兴趣的朋友们可以自己实现下。



读取文件内容

```
RandomAccessFile raf = new RandomAccessFile("test.txt", "rw" );  
raf.seek(5);  
byte [] b = new byte[1024];  
  
int off = 0;  
int len = 5;  
raf.read(b, off, len);  
  
String str = new String(b, 0, len);  
System.out.println(str);  
  
raf.close();
```



写入文件内容

```
RandomAccessFile raf = new RandomAccessFile("test.txt", "rw");  
raf.seek(5);
```

//先读出来

```
String temp = raf.readLine();
```

```
raf.seek(5);
```

```
raf.write("xyz".getBytes());
```

```
raf.write(temp.getBytes());
```

```
raf.close();
```



```
RandomAccessFile raf1 = new RandomAccessFile("hello.txt", "rw");
```

```
raf1.seek(5);
```

```
//方式一：
```

```
//StringBuilder info = new StringBuilder((int) file.length());
```

```
//byte[] buffer = new byte[10];
```

```
//int len;
```

```
//while((len = raf1.read(buffer)) != -1){
```

```
////info += new String(buffer,0,len);
```

```
//info.append(new String(buffer,0,len));
```

```
//}
```

```
//方式二：
```

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

```
byte[] buffer = new byte[10];
```

```
int len;
```

```
while((len = raf1.read(buffer)) != -1){
```

```
baos.write(buffer, 0, len);
```

```
}
```

```
raf1.seek(5);
```

```
raf1.write("xyz".getBytes());
```

```
raf1.write(baos.toString().getBytes());
```

```
baos.close();
```

```
raf1.close();
```



流的基本应用小节

- 流是用来处理数据的。
- 处理数据时，一定要先明确**数据源**，与**数据目的地**
 - 数据源可以是文件，可以是键盘。
 - 数据目的地可以是文件、显示器或者其他设备。
- 而流只是在帮助数据进行传输,并对传输的数据进行处理，比如过滤处理、转换处理等。



13-11 NIO.2中Path、 Paths、Files类的使用



Java NIO 概述

- Java NIO (New IO, Non-Blocking IO)是从Java 1.4版本开始引入的一套新的IO API，可以替代标准的Java IO API。NIO与原来的IO有同样的作用和目的，但是使用的方式完全不同，NIO支持面向缓冲区的(IO是面向流的)、基于通道的IO操作。**NIO将以更加高效的方式进行文件的读写操作。**
- Java API中提供了两套NIO，一套是针对标准输入输出**NIO**，另一套就是网络编程**NIO**。
 - |-----java.nio.channels.Channel
 - ✓ |-----FileChannel:处理本地文件
 - ✓ |-----SocketChannel: TCP网络编程的客户端的Channel
 - ✓ |-----ServerSocketChannel:TCP网络编程的服务器端的Channel
 - ✓ |-----DatagramChannel: UDP网络编程中发送端和接收端的Channel



NIO. 2

- 随着 JDK 7 的发布，Java对NIO进行了极大的扩展，增强了对文件处理和文件系统特性的支持，以至于我们称他们为 NIO.2。因为 NIO 提供的一些功能，NIO已经成为文件处理中越来越重要的部分。



Path、Paths和Files核心API

- 早期的Java只提供了一个File类来访问文件系统，但File类的功能比较有限，所提供的方法性能也不高。而且，大多数方法在出错时仅返回失败，并不会提供异常信息。
- NIO. 2为了弥补这种不足，引入了Path接口，代表一个平台无关的平台路径，描述了目录结构中文件的位置。Path可以看成是File类的升级版本，实际引用的资源也可以不存在。
- 在以前IO操作都是这样写的：

```
import java.io.File;  
File file = new File("index.html");
```
- 但在Java7 中，我们可以这样写：

```
import java.nio.file.Path;  
import java.nio.file.Paths;  
Path path = Paths.get("index.html");
```



Path、Paths和Files核心API

- 同时，NIO.2在java.nio.file包下还提供了Files、Paths工具类，Files包含了大量静态的工具方法来操作文件；Paths则包含了两个返回Path的静态工厂方法。
- Paths 类提供的静态 get() 方法用来获取 Path 对象：
 - `static Path get(String first, String ... more)` : 用于将多个字符串串连成路径
 - `static Path get(URI uri)`: 返回指定uri对应的Path路径



Path接口

● Path 常用方法:

- String toString() : 返回调用 Path 对象的字符串表示形式
- boolean startsWith(String path) : 判断是否以 path 路径开始
- boolean endsWith(String path) : 判断是否以 path 路径结束
- boolean isAbsolute() : 判断是否是绝对路径
- Path getParent() : 返回Path对象包含整个路径, 不包含 Path 对象指定的文件路径
- Path getRoot() : 返回调用 Path 对象的根路径
- Path getFileName() : 返回与调用 Path 对象关联的文件名
- int getNameCount() : 返回Path 根目录后面元素的数量
- Path getName(int idx) : 返回指定索引位置 idx 的路径名称
- Path toAbsolutePath() : 作为绝对路径返回调用 Path 对象
- Path resolve(Path p) : 合并两个路径, 返回合并后的路径对应的Path对象
- File toFile() : 将Path转化为File类的对象



Files 类

- **java.nio.file.Files** 用于操作文件或目录的工具类。
- **Files**常用方法:
 - `Path copy(Path src, Path dest, CopyOption ... how)` : 文件的复制
 - `Path createDirectory(Path path, FileAttribute<?> ... attr)` : 创建一个目录
 - `Path createFile(Path path, FileAttribute<?> ... arr)` : 创建一个文件
 - `void delete(Path path)` : 删除一个文件/目录, 如果不存在, 执行报错
 - `void deleteIfExists(Path path)` : **Path**对应的文件/目录如果存在, 执行删除
 - `Path move(Path src, Path dest, CopyOption...how)` : 将 **src** 移动到 **dest** 位置
 - `long size(Path path)` : 返回 **path** 指定文件的大小



Files 类

- **Files**常用方法：用于判断

- `boolean exists(Path path, LinkOption ... opts)` : 判断文件是否存在
- `boolean isDirectory(Path path, LinkOption ... opts)` : 判断是否是目录
- `boolean isRegularFile(Path path, LinkOption ... opts)` : 判断是否是文件
- `boolean isHidden(Path path)` : 判断是否是隐藏文件
- `boolean isReadable(Path path)` : 判断文件是否可读
- `boolean isWritable(Path path)` : 判断文件是否可写
- `boolean notExists(Path path, LinkOption ... opts)` : 判断文件是否不存在

- **Files**常用方法：用于操作内容

- `SeekableByteChannel newByteChannel(Path path, OpenOption...how)` : 获取与指定文件的连接, how 指定打开方式。
- `DirectoryStream<Path> newDirectoryStream(Path path)` : 打开 path 指定的目录
- `InputStream newInputStream(Path path, OpenOption...how)` : 获取 `InputStream` 对象
- `OutputStream newOutputStream(Path path, OpenOption...how)` : 获取 `OutputStream` 对象

让天下没有难学的技术



尚硅谷