**流**

1. **四大基本流（抽象流）故流是以抽象类的方式组织在一起的**

**InputStream,OutputStream（字节流）**

**Reader,Writer（字符流）**

**二. 缓冲流**

**1.只有四大基本流有缓冲流**

**2.缓冲流是一个包裹流**

流的概念和作用

流是一组有顺序的，有起点和终点的字节集合，是对数据传输的总称或抽象。即数据在两设备间的传输称为流。

流的本质是数据传输，根据数据传输的不同特性将流抽象封装成不同的类，方便更直观的进行数据操作。

IO流的分类

* 根据处理数据类型的不同分为：字符流和字节流
* 根据数据流向不同分为：输入流和输出流

输入流和输出流

所谓输入流和输出流，实际是相对内存而言。

* 将外部数据读取到Java内存当中就是所谓的输入流
* 而将Java内存当中的数据写入存放到外部设备当中，就是所谓的输出流

字符流和字节流

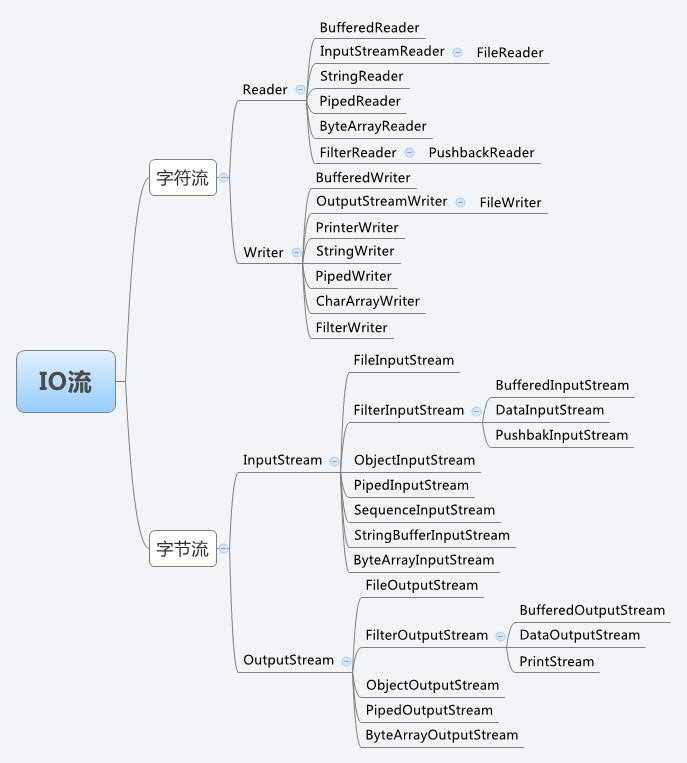
实际上最根本的流都是字节流，因为计算机上数据存储的最终形式实际上都是字节。

而字符流的由来是因为：不同文字的数据编码不同，所以才有了对字符进行高效操作的流对象。

本质其实就是基于字节流读取时，去查了指定的码表。 所以，字节流和字符流的区别在于：

* 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。
* 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。

通过一张关系图来了解一下Java中的IO流构成：



通过该图，我们除了可以看到IO流体系下，数量众多的具体子类。

还可以注意到：这些子类的命名有一个规律，那就是：

字符流体系当中的顶层类是Reader和Writer；字节流体系当中的顶层类是InputStream和OutputStream。

而它们对应的体系下的子类的命名中，都以该顶层类的命名作为其后缀，而前缀则说明了具体的特征。

正确选择合适的IO对象

在上面的Java-IO体系构成图当中，我们已经发现这个体系中拥有众多特定的子类，那么，我们应该如何在如此众多的选择中选取最适合的IO对象使用呢？

究其根本，在众多的子类当中，其实落实下来对于数据的操作方法基本上都很类似，都离不开对于数据的读(read())写(write())方法。

所以，只要了解不同IO流对象的自身特性之后，基本上就可以按照以下的步骤，来根据实际需求选取最合适的IO流对象使用：

(1)、明确你要做的操作是读还是写（也就是明确操作的是数据源还是数据目的）

* 如果是要读取数据(操作数据源)：InputStream或Reader
* 如果是要写入数据(操作数据目的)：OutputSteam或Writer

(2)、明确要处理的数据是否是纯文本数据

* 如果数据为纯文本：Reader或Writer (字符流对象操作文本数据效率更高)
* 否则使用：InputStream或OutputSteam

(3)、明确源或目的的具体设备

* 硬盘(外部文件) - File
* 内存 - 数组
* 网络数据 - Socket流

(4)、明确是否需要额外功能

假设以一个小的需求：“请复制一个txt文件当中的内容至新的txt文件当中”为例。我们来验证一下上面所说的步骤：

(1)、复制一个文件，自然读写操作都要涉及到。所以这里数据源可选：InputStream或Reader，数据目的可选：OutputSteam或Writer。

(2)、数据源是一个txt文件，其内容是纯文本数据。于是我们可以进一步缩小范围，数据源被确定为：Reader；同理，数据目的为:Writer。

(3)、明确数据涉及的具体设备，我们已经注意到关键词“文件”。那么对于具体的流对象的选择，莫数据源FileReader，数据目的FileWriter莫属了。

到了这里，要选取的IO对象已经明确了，剩下的工作无非就是通过对象调用对应的读写方法来操作数据了。

除此之外，我们说到，有时还需要明确的是：你是否需要一些额外的功能。例如：

如果你想要提高操作数据的效率，可以选择缓冲区流对象(Buffered...)来对原有的流对象进行装饰；

或者，假如你操作的数据源是字节流，而数据目的为字符流，那么可以通过转换流对象(InputStreamReader/OutputStreamWriter)，

来让输入流与输出流转换达成一致，从而提高读写操作的效率。

举个例子说：要获取用户在控制台输入的数据，存放到一个文本文件当中去。

这个时候，数据源为系统的输入流:System.in，该输入流属于字节流InputStream体系中。

但实际上我们知道用户通过键盘输入的数据其实就是纯文本数据，所以我们最好选择使用字符流Reader来处理数据，效率更高。

这个时候就可以使用转换流对象InputStreamReader来将原本的字节流转换为字符流来处理：

InputStreamReader isr = newInputStreamReader(System.in)；

而同时我们希望加入缓冲区技术，从而使效率进一步提升，最终的代码就变为了：

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));

IO流的具体应用

1、输入输出流的常用操作

首先就以“复制文件”为例，分别看一看字符流和字节流的常用操作。

通过字节流对象来完成上述操作，代码为：

package com.tsr.j2seoverstudy.io;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

/\*

\* 通过字节流对象:FileInputStream，完成文件内容复制

\*/

public class ByteStreamDemo {

private static final String LINE\_SEPARATOR = System

.getProperty("line.separator");

private static final int BUFFER\_SIZE = 1024;

public static void main(String[] args) {

writeContentInFile();

copyFile\_1();

copyFile\_2();

}

static void writeContentInFile() {

// 声明输出流对象.

FileOutputStream fos = null;

// 指定输出内容

String content = "first" + LINE\_SEPARATOR + "second";

try {

// 构造流对象，指定输出文件

fos = new FileOutputStream("byteStream\_1.txt");

// 向指定文件中写入指定内容

fos.write(content.getBytes());

} catch (FileNotFoundException e) {

System.out.println("未查找到对应文件!");

} catch (IOException e) {

System.out.println("书写异常!");

} finally {

if (fos != null)

try {

fos.close();

} catch (IOException e) {

System.out.println("关闭流对象异常.");

}

}

}

static void copyFile\_1() {

// 输入流对象

FileInputStream fis = null;

// 输出流对象

FileOutputStream fos = null;

try {

fis = new FileInputStream("byteStream\_1.txt");

fos = new FileOutputStream("byteStream\_1\_copy.txt");

int position = 0;

while ((position = fis.read()) != -1) {

fos.write(position);

fos.flush();

}

} catch (FileNotFoundException e) {

} catch (IOException e) {

} finally {

try {

if (fis != null)

fis.close();

if (fos != null)

fos.close();

} catch (IOException e) {

// TODO: handle exception

}

}

}

static void copyFile\_2() {

// 输入流对象

FileInputStream fis = null;

// 输出流对象

FileOutputStream fos = null;

try {

fis = new FileInputStream("byteStream\_1.txt");

fos = new FileOutputStream("byteStream\_1\_copy.txt");

byte[] buffer = new byte[BUFFER\_SIZE];

int length = 0;

while ((length = fis.read(buffer)) != -1) {

fos.write(buffer,0,length);

fos.flush();

}

} catch (FileNotFoundException e) {

} catch (IOException e) {

} finally {

try {

if (fis != null)

fis.close();

if (fos != null)

fos.close();

} catch (IOException e) {

// TODO: handle exception

}

}

}

}

而通过字符流的代码为：

package com.tsr.j2seoverstudy.io;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class CharacterStreamDemo {

private static final String LINE\_SEPARATOR = System

.getProperty("line.separator");

private static final int BUFFER\_SIZE = 1024;

public static void main(String[] args) {

writeContentInFile();

//copyFile\_1();

copyFile\_2();

}

private static void copyFile\_2() {

FileWriter fw = null;

FileReader fr = null;

try {

fw = new FileWriter("characterStream\_1\_copy.txt");

fr = new FileReader("characterStream\_1.txt");

int length = 0;

char [] buffer = new char[BUFFER\_SIZE];

while ((length = fr.read(buffer)) != -1) {

fw.write(buffer,0,length);

fw.flush();

}

} catch (IOException e) {

// TODO: handle exception

} finally {

try {

if (fr != null)

fr.close();

if (fw != null)

fw.close();

} catch (IOException e) {

// TODO: handle exception

}

}

}

private static void copyFile\_1() {

FileWriter fw = null;

FileReader fr = null;

try {

fw = new FileWriter("characterStream\_1\_copy.txt");

fr = new FileReader("characterStream\_1.txt");

int position = 0;

while ((position = fr.read()) != -1) {

fw.write(position);

}

} catch (IOException e) {

// TODO: handle exception

} finally {

try {

if (fr != null)

fr.close();

if (fw != null)

fw.close();

} catch (IOException e) {

// TODO: handle exception

}

}

}

private static void writeContentInFile() {

FileWriter fw = null;

String content = "first" + LINE\_SEPARATOR + "second";

try {

fw = new FileWriter("characterStream\_1.txt");

fw.write(content);

} catch (IOException e) {

// TODO: handle exception

} finally {

try {

if (fw != null) {

fw.close();

}

} catch (IOException e) {

// TODO: handle exception

}

}

}

}

通过上述的两个例子中，主要想要说明的几点是：  
1.可以看到对于所谓的字节流和字符流之间，对于数据的输入输出操作的方法使用，实际上大同小异的。

2.首先可以看到的不同是，字节流文件对象在向外部文件写入内容时，必须通过单个int型的值或byte数组的形式。

而字符流则可以通过int值，char型数组写入之外，还可以直接将字符串对象(String)形式写入。

3.对于流对象的读入数据的操作，都有两种方式：

第一种方式是，每次读取单个数据：字节流每次读取单个字节，字符流每次读入单个字符(2个字节)。

但同样都返回一个int型的值。字符流返回的这个值实际就读取到的字符在Unicode码表中的位置。返回值为-1则代表数据读取完毕。

第二种方式是，每次读取批量的数据到一个指定类型的缓冲区数组当中，从而提高了读取效率。不同的是：

字节流能够接受的缓冲区数组类型为byte型数组，而字符流接受的类型为char型数组。

它们也会返回一个int型的值，该值代表当前读取操作读取到的数据的实际长度，同样返回-1代表读取完毕。

4、调用完成流对象的写入操作之后，一定要记得使用flush刷新该流对象的缓冲，才能真正的将数据写入到外部文件。

flush和close的区别在于：flush用于刷新流的缓冲，但不关闭该流对象。

调用close方法关闭流对象，并且之前会自动的调用一次flush刷新缓冲。

5、通过上述几点的描述，我们也进一步验证了，当操作数据为纯文本时，应当选取字符流，因为效率更高。

但如果数据为非纯文本，例如赋值一张图片操作时，就只能选择字节流来完成。

2、缓冲区流对象

我们前面已经说过了，Java中IO体系下的各个类的命名都有很强的目的性。

所以顾名思义，缓冲区流对象就是指该体系下前缀为buffered的几个类。

缓冲区流对象，最大的好处在于：提高数据读写操作的效率。

如果仅仅是了解对于缓冲区流对象的使用方法，实际上并不复杂，其方式就是：

创建一个缓冲区流对象，然后将需要提高操作效率的流对象作为构造参数传递给创建的该缓冲区流对象就搞定了。

public class BufferedDemo {

public static void main(String[] args) {

try {

FileWriter fw = new FileWriter("buffer.txt");

BufferedWriter bw = new BufferedWriter(fw);

bw.write("缓冲区流对象用于调高读写效率");

bw.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

而我们需要了解的是，缓冲区流对象究竟是怎样实现提高效率的目的的呢？  
在前面我们用于说明输出流的使用时，你可能已经看到了这样的代码：

int length = 0;

char [] buffer = new char[BUFFER\_SIZE];

while ((length = fr.read(buffer)) != -1) {

fw.write(buffer,0,length);

fw.flush();

}

没错，这里我们将输入流读取到的指定数量的数据先存放在一个指定类型的数组当中。  
然后再将存放在数组中的数据取出，进行写入操作。这里数组就起到了一个临时存储的作用，也就是所谓的缓冲区。

而Java中的缓冲区流对象，其实现原理其实也是这样。只不过他们将这些较复杂的操作封装了起来，形成一个单独的类，更加方便使用。

了解了原理，其实我们自己也可以定义所谓的缓冲区流对象。例如：

public class MyBufferedReader extends Reader {

private Reader r;

//定义一个数组作为缓冲区。

private char[] buf = new char[1024];

//定义一个指针用于操作这个数组中的元素。当操作到最后一个元素后，指针应该归零。

private int pos = 0;

//定义一个计数器用于记录缓冲区中的数据个数。 当该数据减到0，就从源中继续获取数据到缓冲区中。

private int count = 0;

MyBufferedReader(Reader r){

this.r = r;

}

/\*\*

\* 该方法从缓冲区中一次取一个字符。

\* @return

\* @throws IOException

\*/

public int myRead() throws IOException{

if(count==0){

count = r.read(buf);

pos = 0;

}

if(count<0)

return -1;

char ch = buf[pos++];

count--;

return ch;

/\*

//1,从源中获取一批数据到缓冲区中。需要先做判断，只有计数器为0时，才需要从源中获取数据。

if(count==0){

count = r.read(buf);

if(count<0)

return -1;

//每次获取数据到缓冲区后，角标归零.

pos = 0;

char ch = buf[pos];

pos++;

count--;

return ch;

}else if(count>0){

char ch = buf[pos];

pos++;

count--;

return ch;

}\*/

}

public String myReadLine() throws IOException{

StringBuilder sb = new StringBuilder();

int ch = 0;

while((ch = myRead())!=-1){

if(ch=='\r')

continue;

if(ch=='\n')

return sb.toString();

//将从缓冲区中读到的字符，存储到缓存行数据的缓冲区中。

sb.append((char)ch);

}

if(sb.length()!=0)

return sb.toString();

return null;

}

public void myClose() throws IOException {

r.close();

}

@Override

public int read(char[] cbuf, int off, int len) throws IOException {

return 0;

}

@Override

public void close() throws IOException {

}

}

3、转换流

转换流是指InputStreamReader和OutputStreamWriter。顾名思义，也就是指用于在字符流与字节流之间做转换工作的流对象。

那么，我么已经知道了字符流与字节流之间的本质区别就是：字节流实际是通过查指定的编码表，一次提取对应数量的字节转换为字符。

所以，既然是转换流，自然就离不开一个重要的概念：编码。而在转换流对象的构造器中，也可以看到提供了指定编码形式的构造器(默认编码为unicode)。

那么，以我们在上面提到过的需求 “将用户从键盘输入的值，存储到一个外部txt文件当中”为例，通过转换流的实现方式为：

package com.tsr.j2seoverstudy.io;

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

import java.io.InputStream;

import java.io.InputStreamReader;

import java.io.Reader;

public class CastStreamDemo {

public static void main(String[] args) {

// 获取系统输入流

InputStream in = System.in;

// 因为用户从键盘输入的数据肯定是纯文本，所以为了提高效率，我们将其转换为字符流处理

BufferedReader reader = new BufferedReader(new InputStreamReader(in));

try {

//构造输出流

FileWriter fw = new FileWriter("castSteam.txt");

BufferedWriter writer = new BufferedWriter(fw);

String line = null;

//每次读取单行数据写入外部文件

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

//如果用户输入quit，代表退出当前程序

if(line.equals("quit"))

break;

writer.write(line);

writer.newLine();

writer.flush();

}

} catch (IOException e) {

}

}

}

4、File类

提到IO流，就不得不提到一个与之紧密相连的类的使用，也就是File类。

File类封装了一系列对于文件或文件夹的操作方法，包括：创建，删除，重命名，获取文件信息等等。

我们依然通过一段代码，来看一看File类的常用操作：

package com.tsr.j2seoverstudy.io;

import java.io.File;

import java.io.IOException;

import java.text.DateFormat;

import java.util.Date;

public class FileDemo {

public static void main(String[] args) {

creatNewFile();

deleteAllFile(new File("E:\\FileTest"));

getFileInfo();

}

/\*

\* 创建文件或文件夹

\*/

public static void creatNewFile() {

// 创建单个文件夹

File folder = new File("E:\\FileTest");

if (!folder.exists()) {// 判断指定文件夹是否已经存在

boolean b = folder.mkdir();

System.out.println("文件夹创建是否成功?" + b);

}

// 创建多级文件目录

File folders = new File("E:\\FileTest\\1\\2\\3\\4");

if (!folders.exists()) {

boolean b = folders.mkdirs();

System.out.println("多级文件目录创建是否成功?" + b);

}

// 创建文件

File txt = new File("E:\\FileTest\\file.txt");

if (!txt.exists()) {

try {

boolean b = txt.createNewFile();

System.out.println("txt文件是否创建成功?" + b);

} catch (IOException e) {

}

}

}

/\*

\* 删除文件，分为两种情况：

\* 1、删除单个文件或删除删除空的文件目录，这种情况很简单，直接调用delete方法就搞定了

\* 2、但是当要删除指定目录下的所有内容(包括该目录下的文件以及其子目录和子目录下的文件)， 这时就需要涉及深度的遍历，通过递归来完成。

\*/

public static void deleteAllFile(File dir) {

File[] files = dir.listFiles();

System.out.println(files.length);

// 遍历file对象

for (File file : files) {

// 如果是文件夹

if (file.isDirectory()) {

// 递归

deleteAllFile(file);

} else {// 否则直接删除文件

file.delete();

}

}

// 删除文件夹

dir.delete();

}

//重命名

public static void renameToDemo() {

File f1 = new File("c:\\9.mp3");

File f2 = new File("d:\\aa.mp3");

boolean b = f1.renameTo(f2);

System.out.println("b="+b);

}

//文件对象的一些常用判断方法

public static void decideFile(){

File folder = new File("E:\\FileTest");

//判断文件是否存在

boolean b = folder.exists();

//判断是不是文件夹

boolean b1 = folder.isDirectory();

//判断是不是文件

boolean b2 = folder.isFile();

//判断是不是隐藏文件

boolean b3 = folder.isHidden();

//测试指定路径是不是绝对路径

boolean b4 = folder.isAbsolute();

}

//获取File对象信息

public static void getFileInfo(){

File file = new File("E:\\FileTest\\");

String name = file.getName();//返回由此抽象路径名表示的文件或目录的名称

String absPath = file.getAbsolutePath();//绝对路径。

String path = file.getPath();//将此抽象路径名转换为一个路径名字符串。

long len = file.length();//返回由此抽象路径名表示的文件的长度

long time = file.lastModified();//最后修改时间

Date date = new Date(time);

DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG,DateFormat.LONG);

String str\_time = dateFormat.format(date);

//返回此抽象路径名父目录的路径名字符串；如果此路径名没有指定父目录，则返回 null

String parentPath = file.getParent();

System.out.println("parent:"+parentPath);

System.out.println("name:"+name);

System.out.println("absPath:"+absPath);

System.out.println("path:"+path);

System.out.println("len:"+len);

System.out.println("time:"+time);

System.out.println("str\_time:"+str\_time);

}

}

对于file的使用，需要值得注意的就是：  
关于对file对象的删除方法，如果指定的路径所代表的是一个文件目录，并且该目录下还存在其它文件或子目录。

那么该对象代表的文件夹是无法被删除的 ，原理很简单：你可以试着通过cmd命令行删除这样的文件夹，当然是行不通的。

所以如果要删除这样的文件目录，通常会通过递归对该文件目录下的内容进行深度的遍历，逐次将所有内容删除后，才能删除该文件夹。

而关于通过file对象的list或listFile方法进行文件内容的遍历时，还可以通过自定义拦截器(如FileFilter)来进行有条件的遍历。

例如：遍历出所有后缀为".txt"的文件。

5、通过Properties，生成你自己的属性文件

在实际的开发中，通常都会通过一系列的属性文件来记录我们软件的相关信息或者客户的使用情况。

例如：我们开发了一个软件，可以提供免费试用5次。当试用次数用完后，就必须付费购买正版。

public class PropertiesDemo {

/\*\*

\* @param args

\* @throws IOException

\* @throws Exception

\*/

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

getAppCount();

}

public static void getAppCount() throws IOException{

//将配置文件封装成File对象。

File confile = new File("count.properties");

if(!confile.exists()){

confile.createNewFile();

}

FileInputStream fis = new FileInputStream(confile);

Properties prop = new Properties();

prop.load(fis);

//从集合中通过键获取次数。

String value = prop.getProperty("time");

//定义计数器。记录获取到的次数。

int count =0;

if(value!=null){

count = Integer.parseInt(value);

if(count>=5){

throw new RuntimeException("您的使用次数已满5次，若喜欢我们的软件，请到指定网站购买正版.");

}

}

count++;

//将改变后的次数重新存储到集合中。

prop.setProperty("time", count+"");

FileOutputStream fos = new FileOutputStream(confile);

prop.store(fos, "");

fos.close();

fis.close();

}

}

Properties实际上是容器类HashTable的子类，而Properties 类表示的是一个持久化的属性集。  
Properties可保存在流中或从流中加载。属性列表中每个键及其对应值都是一个字符串。

也就是说，这个特殊的子类基本上都是用于与IO流相关的操作，用于一些常规数据的持久化。

它的使用过程大概就是：

通过load方法用于接收一个输入流对象，也就是指从输入流中读取属性列表（键和元素对）。

``当读取结束输入流中的属性列表后，就可以通过getProperty获取对应键的值或通过setProperty方法修改对应键的值。

当你修改完接收的属性列表当中的值之后，就可以通过store方法将此Properties 表中的属性列表（键和元素对）写入对应的输出流，以更新属性文件。

注：Java编写的软件的属性文件通常都使用“.properties” 作为文件格式后缀。

6、序列流

保存的文件为.ser文件。

实现序列化的类必须实现Serializable接口（标志接口）。

实现Serializable 接口的类型都应该制定一个serialVersionUID,如果没有这个ID,当文件被拷贝到其他的电脑中时，这个文件无法被其他电脑使用。

**private** **static** **final** **long** *serialVersionUID* = 1L;

序列流的基本应用

来看一看JDK API说明文档中对于序列输入流SequenceInputStream的说明：

SequenceInputStream 表示其他输入流的逻辑串联。它从输入流的有序集合开始，并从第一个输入流开始读取，直到到达文件末尾。

接着从第二个输入流读取，依次类推，直到到达包含的最后一个输入流的文件末尾为止。

通俗的说，也就是使用该流对象，可以将多个流对象当中的内容进行有序的合并(序列化)。

举例来说，假设我们现在有三个txt文件：a.txt、b.txt、c.txt。而我们需要将这个三个txt文件中的内容合并到一个txt文件当中。

按照基本的思路来说，我们应当先构造3个输入流对象分别关联3个不同的txt文件，然后依次将它们的内容写入到新的txt文件当中。

而通过序列流的使用，我们就可以直接将这个3个输入流进行序列化，直接写入新的文件。

public class SequenceInputStreamDemo {

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

String[] fileNames = { "a", "b", "c" };

ArrayList<FileInputStream> al = new ArrayList<FileInputStream>();

for (int x = 0; x < 3; x++) {

al.add(new FileInputStream(fileNames[x] + ".txt"));

}

Enumeration<FileInputStream> en = Collections.enumeration(al);

SequenceInputStream sis = new SequenceInputStream(en);

FileOutputStream fos = new FileOutputStream("d.txt");

byte[] buf = new byte[1024];

int len = 0;

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

fos.write(buf, 0, len);

}

fos.close();

sis.close();

}

}

对于序列流的使用，需要注意的也就是，其接受的需要进行序列化的流对象的容器类型必须是：Enumeration

文件的切割与合并

很多网站在上传文件的时候，因为文件过大会影响传输，所以都会选择对文件做切割。

也就是说将一个大的文件，分割为多个较小的文件。在Java中，就可以通过序列流来实现这样的功能：

public class FileCuttingUtil {

private static final int SIZE = 1024 \* 1024;

/\*\*

\* @param args

\* @throws Exception

\*/

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

File file = new File("F:\\Audio\\KuGou\\All in My Head.mp3");

File parts = new File("E:\\PartFiles");

splitFile(file);

mergeFile(parts);

}

private static void splitFile(File file) throws IOException {

// 用读取流关联源文件。

FileInputStream fis = new FileInputStream(file);

// 定义一个1M的缓冲区。

byte[] buf = new byte[SIZE];

// 创建目的。

FileOutputStream fos = null;

int len = 0;

int count = 1;

/\*

\* 切割文件时，必须记录住被切割文件的名称，以及切割出来碎片文件的个数。 以方便于合并。

\* 这个信息为了进行描述，使用键值对的方式。用到了properties对象

\*/

Properties prop = new Properties();

File dir = new File("E:\\partfiles");

if (!dir.exists())

dir.mkdirs();

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

fos = new FileOutputStream(new File(dir, (count++) + ".part"));

fos.write(buf, 0, len);

fos.close();

}

// 将被切割文件的信息保存到prop集合中。

prop.setProperty("partcount", count + "");

prop.setProperty("filename", file.getName());

fos = new FileOutputStream(new File(dir, count + ".properties"));

// 将prop集合中的数据存储到文件中。

prop.store(fos, "save file info");

fos.close();

fis.close();

}

public static void mergeFile(File dir) throws IOException {

/\*

\* 获取指定目录下的配置文件对象。

\*/

File[] files = dir.listFiles(new SuffixFilter(".properties"));

if (files.length != 1)

throw new RuntimeException(dir + ",该目录下没有properties扩展名的文件或者不唯一");

// 记录配置文件对象。

File confile = files[0];

// 获取该文件中的信息================================================。

Properties prop = new Properties();

FileInputStream fis = new FileInputStream(confile);

prop.load(fis);

String filename = prop.getProperty("filename");

int count = Integer.parseInt(prop.getProperty("partcount"));

// 获取该目录下的所有碎片文件。 ==============================================

File[] partFiles = dir.listFiles(new SuffixFilter(".part"));

if (partFiles.length != (count - 1)) {

throw new RuntimeException(" 碎片文件不符合要求，个数不对!应该" + count + "个");

}

// 将碎片文件和流对象关联 并存储到集合中。

ArrayList<FileInputStream> al = new ArrayList<FileInputStream>();

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

al.add(new FileInputStream(partFiles[x]));

}

// 将多个流合并成一个序列流。

Enumeration<FileInputStream> en = Collections.enumeration(al);

SequenceInputStream sis = new SequenceInputStream(en);

FileOutputStream fos = new FileOutputStream(new File(dir,filename));

byte[] buf = new byte[1024];

int len = 0;

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

fos.write(buf, 0, len);

}

fos.close();

sis.close();

}

}

class SuffixFilter implements FilenameFilter {

private String suffix;

public SuffixFilter(String suffix) {

super();

this.suffix = suffix;

}

@Override

public boolean accept(File dir, String name) {

return name.endsWith(suffix);

}

}

利用IO流实现对象的持久化

举例来说，我们一个程序中通常都会涉及到很多实体类(bean)，大多时候我们都需要对这些实体类对象存放的数据进行数据持久化。

持久化的方式有很多，实际开发中最常用的可能就是类似通过xml文件或者数据库的方式了。

那么，当我们想通过IO流对象，将一个实体对象的数据保存到一个外部文件当中时。应当选择什么样的流对象使用呢？

Java提供的两个类分别是：ObjectInputStream与ObjectOutputStream。

其中ObjectOutputStream流对象用于：将对象序列化后写入。

而ObjectInputStream则用于对那些经ObjectOutputStream序列化的数据进行反序列化。

你可能在最初学习Java时，就知道了一个关键字“transient”，但却很少使用。

该关键字的使用都是关联到对象序列化的，也就是说：一个对象中被transient修饰的字段，不会参与到对象的序列化工作。

该字段是瞬态的，它的值不会被持久化到外部文件当中。

举例来说：你的某个实体类含有“密码”字段，那么你当然不希望如此隐私的信息暴露给外部。就可以通过transient修饰该字段。

注：使用该流对象进行序列化工作的类必须实现Serializable接口。

package com.tsr.j2seoverstudy.io;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

public class ObjectStreamDemo {

/\*\*

\* @param args

\* @throws IOException

\* @throws ClassNotFoundException

\*/

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

ClassNotFoundException {

/\*

\* 输出结果为：小强:0。

\* 可以看到因为age被设为瞬态，所以该属性不会参与对象的序列化工作。

\* 当进行对象的反序列化时，则会取到一个该数据类型的默认初始化值

\*/

writeObj();

readObj();

}

public static void readObj() throws IOException, ClassNotFoundException {

ObjectInputStream ois = new ObjectInputStream(new FileInputStream(

"obj.object"));

// 对象的反序列化。

Person p = (Person) ois.readObject();

System.out.println(p.getName() + ":" + p.getAge());

ois.close();

}

public static void writeObj() throws IOException, IOException {

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(

"obj.object"));

// 对象序列化。 被序列化的对象必须实现Serializable接口。

oos.writeObject(new Person("小强", 30));

oos.close();

}

}

class Person implements Serializable {

private static final long serialVersionUID = 1L;

private String name;

private transient int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

管道流

管道流与其它IO流的最大不同之处在于：管道流是结合多线程使用的。

简单的说，其特点就是：输出管道流与输入管道流存在于不同的线程之间，但它们之间能够互相将管道连接起来。

也就是说，当我们在线程A中定义一个输入管道流，用于接收在线程B当中的输出管道流所书写的数据信息。

package com.tsr.j2seoverstudy.io;

import java.io.IOException;

import java.io.PipedInputStream;

import java.io.PipedOutputStream;

public class PipedStreamDemo {

/\*\*

\* @param args

\* @throws IOException

\*/

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

PipedInputStream input = new PipedInputStream();

PipedOutputStream output = new PipedOutputStream();

input.connect(output);

new Thread(new Input(input)).start();

new Thread(new Output(output)).start();

}

}

class Input implements Runnable {

private PipedInputStream in;

Input(PipedInputStream in) {

this.in = in;

}

public void run() {

try {

byte[] buf = new byte[1024];

int len = in.read(buf);

String s = new String(buf, 0, len);

System.out.println("s=" + s);

in.close();

} catch (Exception e) {

// TODO: handle exception

}

}

}

class Output implements Runnable {

private PipedOutputStream out;

Output(PipedOutputStream out) {

this.out = out;

}

public void run() {

try {

Thread.sleep(5000);

out.write("我是线程B当中的输出管道流，给你输出信息了，线程A当中的兄弟！".getBytes());

} catch (Exception e) {

// TODO: handle exception

}

}

}

DataStream-操作基本数据类型的流

简单的说，该流对象最大的特点就是对基本的输入输出流进行了一定的“装饰”。

专门针对于Java当中的基本数据类型的操作方法进行了封装，所以当操作的数据是基本数据类型时

就应当选择该中流对象使用，因为效率更高。

public class DataStreamDemo {

private static void write() {

try {

DataOutputStream dos = new DataOutputStream(new FileOutputStream(

"dataStream.txt"));

dos.writeByte(0);

dos.writeChar(0);

dos.writeInt(0);

dos.writeLong(0);

dos.writeDouble(0);

dos.writeFloat(0);

dos.writeUTF("hello");

dos.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

private static void read() {

try {

DataInputStream dis = new DataInputStream(new FileInputStream(

"dataStream.txt"));

dis.readByte();

dis.readChar();

dis.readInt();

dis.readLong();

dis.readFloat();

dis.readDouble();

dis.readUTF();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

操作内存数组的流

就是指：ByteArrayInputStream;ByteArrayOutputStream;CharArrayReader;CharArrayWriter;

与其它的输入输出流不同的是：该种流对象用于，当数据源和数据目的都位于内存当中的情况。

并且，这种类型的流对象调用close方法后，此类当中的方法还是可以被调用到。

public class ByteArrayStreamDemo {

/\*\*

\* @param args

\* @throws IOException

\*/

public static void main(String[] args) {

ByteArrayInputStream bis = new ByteArrayInputStream("abcedf".getBytes());

ByteArrayOutputStream bos = new ByteArrayOutputStream();

int ch = 0;

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

bos.write(ch);

}

System.out.println(bos.toString());

}

}

以上就是对Java IO体系当中常用的流对象和一些提供特殊功能的对象的常用操作做了一个总结。

我们之前已经说过了，无论IO体系下存在这么多不同的具体子类实现，其根本无非就是根据数据传输的不同特性将流抽象封装成不同的类。

所以，对于IO的掌握，重点还是了解不同的流对象的功能特点，从而根据实际需求选择最合适的流对象。从而使代码变得更简单，使读写效率更高。