# 阶段4-Java语言进阶【io流】

### 一、上节课回顾

集合框架

Connection

List：可重复

ArrayList：有序，底层是数组，可以直接通过索引操作元素，查询效率高。

LinkedList：有序，底层是双向链表，插入删除效率高。

Set：不可重复

HashSet：HashMap的key，元素唯一不能重复，无序

LinkedHashSet：LinkedHashMap的key，按插入顺序排序的。

TreeSet：TreeMap的key，元素唯一不能重复，自然顺序

Map 键值对，key唯一，当key相同的时候，会覆盖值。

HashMap：数组+单向链表 ，无序

LinkedHashMap：在HashMap的基础上进行了一个双向的操作。插入顺序

TreeMap：key根据的自然顺序排序，

### 二、作业点评

### 三、本次课知识点，重点难点

【知识点目标】

• File类介绍

• 流的概念

• Java中流的分类

• 字节流

• 字符流

• 转换流

• 进程和线程

• 线程的创建和启动

• 线程的生命周期

• 线程同步

【重点】

• File

• 字节流

• 字符流

• 转换流

• 线程的创建和启动

• 线程同步

【难点】

• 字节流

• 字符流

• 转换流

### 四、本次课程任务知识点讲解

**File类：**将文件系统中的文件和文件夹封装成了对象。提供了更多的属性和行为可以对这些文件和文件夹进行操作。这些是流对象办不到的，因为流只操作数据。

**File类常见方法：**

**1：创建。**

boolean **createNewFile()：**在指定目录下创建文件，如果该文件已存在，则不创建。而对操作文件的输出流而言，输出流对象已建立，就会创建文件，如果文件已存在，会覆盖。除非续写。

boolean **mkdir()：**创建此抽象路径名指定的目录。

boolean **mkdirs()：**创建多级目录。

**2：删除。**

boolean **delete()：**删除此抽象路径名表示的文件或目录。

void **deleteOnExit()：**在虚拟机退出时删除。

**注意：在删除文件夹时，必须保证这个文件夹中没有任何内容，才可以将该文件夹用delete删除。**

window的删除动作，是从里往外删。**注意：**java删除文件不走回收站。要慎用。

**3：获取.**

long **length()：**获取文件大小。

String **getName()：**返回由此抽象路径名表示的文件或目录的名称。

String **getPath()：**将此抽象路径名转换为一个路径名字符串。

String **getAbsolutePath()：**返回此抽象路径名的绝对路径名字符串。

String **getParent()：**返回此抽象路径名父目录的抽象路径名，如果此路径名没有指定父目录，则返回 null。

long **lastModified()：**返回此抽象路径名表示的文件最后一次被修改的时间。

**File.pathSeparator：**返回当前系统默认的路径分隔符，windows默认为 “；”。

**File.Separator：**返回当前系统默认的目录分隔符，windows默认为 “\”。

**4：判断：**

boolean **exists()：**判断文件或者文件夹是否存在。

boolean **isDirectory()：**测试此抽象路径名表示的文件是否是一个目录。

boolean **isFile()：**测试此抽象路径名表示的文件是否是一个标准文件。

boolean **isHidden()：**测试此抽象路径名指定的文件是否是一个隐藏文件。

boolean **isAbsolute()：**测试此抽象路径名是否为绝对路径名。

**5：重命名。**

boolean **renameTo(File dest)：可以实现移动的效果。剪切+重命名。**

**String[] list()：**列出指定目录下的当前的文件和文件夹的名称。包含隐藏文件。

如果调用list方法的File 对象中封装的是一个文件，那么list方法返回数组为null。如果封装的对象不存在也会返回null。只有封装的对象存在并且是文件夹时，这个方法才有效。

------------------------------------------------------------------------------------------------

**递归：**就是函数自身调用自身。

什么时候用递归呢？

当一个功能被重复使用，而每一次使用该功能时的参数不确定，都由上次的功能元素结果来确定。

简单说：**功能内部又用到该功能，但是传递的参数值不确定。**(每次功能参与运算的未知内容不确定)。

**递归的注意事项：**

1：一定要定义递归的条件。

2：递归的次数不要过多。容易出现 **StackOverflowError 栈内存溢出错误**。

**其实递归就是在栈内存中不断的加载同一个函数。**

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

**流也进行分类：**

1：输入流（读）和输出流（写）。

2：因为处理的数据不同，分为字节流和字符流。

**字节流：**处理字节数据的流对象。设备上的数据无论是图片或者dvd，文字，它们都以二进制存储的。二进制的最终都是以一个8位为数据单元进行体现，所以计算机中的最小数据单元就是字节。意味着，字节流可以处理设备上的所有数据，所以字节流一样可以处理字符数据。

**那么为什么要有字符流呢？**因为字符每个国家都不一样，所以涉及到了字符编码问题，那么GBK编码的中文用unicode编码解析是有问题的，所以需要获取中文字节数据的同时+ 指定的编码表才可以解析正确数据。为了方便于文字的解析，所以将字节流和编码表封装成对象，这个对象就是**字符流**。只要操作字符数据，优先考虑使用字符流体系。

注意：**流的操作只有两种：读和写。**

流的体系因为功能不同，但是有共性内容，不断抽取，形成继承体系。该体系一共有四个基类，而且都是抽象类。

**字节流：InputStream OutputStream**

**字符流：Reader Writer**

在这四个系统中，它们的子类，都有一个共性特点：子类名后缀都是父类名，前缀名都是这个子类的功能名称。

--------------------------------------------------------------------------------------------------------------------

public static void main(String[] args) throws IOException { **//读、写都会发生IO异常**

/\*

**1：创建一个字符输出流对象，用于操作文件。该对象一建立，就必须明确数据存储位置，是一个文件。**

**2：对象产生后，会在堆内存中有一个实体，同时也调用了系统底层资源，在指定的位置创建了一个存储数据的文件。**

**3：如果指定位置，出现了同名文件，文件会被覆盖。**

\*/

FileWriter fw = new FileWriter("demo.txt"); // FileNotFoundException

/\*

**调用Writer类中的write方法写入字符串。字符串并未直接写入到目的地中，而是写入到了流中，(其实是写入到内存缓冲区中)。怎么把数据弄到文件中？**

\*/

fw.write("abcde");

fw.flush(); **// 刷新缓冲区，将缓冲区中的数据刷到目的地文件中。**

fw.close(); **// 关闭流，其实关闭的就是java调用的系统底层资源。在关闭前，会先刷新该流。**

}

**close()和flush()的区别：**

flush()：将缓冲区的数据刷到目的地中后，流可以使用。

close()：将缓冲区的数据刷到目的地中后，流就关闭了，该方法主要用于结束调用的底层资源。这个动作一定做。

--------------------------------------------------------------------------------------------------------------------

**io异常的处理方式：io一定要写finally；**

**FileWriter**写入数据的细节：

1：window中的换行符：\r\n两个符号组成。 linux：\n。

2：续写数据，只要在构造函数中传入新的参数true。

3：目录分割符：window \\ /

public static void main(String[] args) {

FileWriter fw = null;

try {

fw = new FileWriter("demo.txt",true);

fw.write("abcde");

}

catch (IOException e ){

System.out.println(e.toString()+"....");

}

finally{

if(fw!=null)

try{

fw.close();

}

catch (IOException e){

System.out.println("close:"+e.toString());

}

}

}

--------------------------------------------------------------------------------------------------------------------

**FileReader：**使用Reader体系，读取一个文本文件中的数据。**返回 -1 ，标志读到结尾。**

import java.io.\*;

class FileReaderDemo {

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

/\*

**创建可以读取文本文件的流对象，FileReader让创建好的流对象和指定的文件相关联。**

\*/

FileReader fr = new FileReader("demo.txt");

int ch = 0;

while((ch = fr.read())!= -1) { **//条件是没有读到结尾**

System.out.println((char)ch); **//调用读取流的read方法，读取一个字符。**

}

fr.close();

}

}

--------------------------------------------------------------------------------------------------------------------

**读取数据的第二种方式：**第二种方式较为高效，**自定义缓冲区。**

import java.io.\*;

class FileReaderDemo2 {

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

FileReader fr = new FileReader("demo.txt"); **//创建读取流对象和指定文件关联。**

**//因为要使用read(char[])方法，将读取到字符存入数组。所以要创建一个字符数组，一般数组的长度都是1024的整数倍。**

char[] buf = new char[1024];

int len = 0;

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

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

}

fr.close();

}

}

--------------------------------------------------------------------------------------------------------------------

**IO中的使用到了一个设计模式：装饰设计模式。**

装饰设计模式解决：对一组类进行功能的增强。

包装：写一个类(包装类)对被包装对象进行包装；

\* 1、包装类和被包装对象要实现同样的接口；

\* 2、包装类要持有一个被包装对象；

\* 3、包装类在实现接口时，大部分方法是靠调用被包装对象来实现的，对于需要修改的方法我们自己实现；

**--------------------------------------------------------------------------------------------------------------------**

**字符流：**

**Reader：**用于读取字符流的抽象类。子类必须实现的方法只有 read(char[], int, int) 和 close()。

**|---BufferedReader：**从字符输入流中读取文本，缓冲各个字符，从而实现字符、数组和行的高效读取。 可以指定缓冲区的大小，或者可使用默认的大小。大多数情况下，默认值就足够大了。

**|---LineNumberReader：**跟踪行号的缓冲字符输入流。此类定义了方法 [setLineNumber(int)](mk:@MSITStore:D:\\shi\\java学习资料\\JDK_API_1_6_zh_CN.CHM::/java/io/LineNumberReader.html" \l "setLineNumber(int)) 和 [getLineNumber()](mk:@MSITStore:D:\\shi\\java学习资料\\JDK_API_1_6_zh_CN.CHM::/java/io/LineNumberReader.html" \l "getLineNumber())，它们可分别用于设置和获取当前行号。

**|---InputStreamReader：**是字节流通向字符流的桥梁：它使用指定的 [charset](mk:@MSITStore:D:\\shi\\java学习资料\\JDK_API_1_6_zh_CN.CHM::/java/nio/charset/Charset.html" \o "java.nio.charset 中的类) 读取字节并将其解码为字符。它使用的字符集可以由名称指定或显式给定，或者可以接受平台默认的字符集。

**|---FileReader：**用来读取字符文件的便捷类。此类的构造方法假定默认字符编码和默认字节缓冲区大小都是适当的。要自己指定这些值，可以先在 FileInputStream 上构造一个 InputStreamReader。

**|---CharArrayReader：**

**|---StringReader：**

-------------------------------------------------

**Writer：**写入字符流的抽象类。子类必须实现的方法仅有 write(char[], int, int)、flush() 和 close()。

**|---BufferedWriter：**将文本写入字符输出流，缓冲各个字符，从而提供单个字符、数组和字符串的高效写入。

**|---OutputStreamWriter：**是字符流通向字节流的桥梁：可使用指定的 [charset](mk:@MSITStore:D:\\shi\\java学习资料\\JDK_API_1_6_zh_CN.CHM::/java/nio/charset/Charset.html" \o "java.nio.charset 中的类) 将要写入流中的字符编码成字节。它使用的字符集可以由名称指定或显式给定，否则将接受平台默认的字符集。

**|---FileWriter：**用来写入字符文件的便捷类。此类的构造方法假定默认字符编码和默认字节缓冲区大小都是可接受的。要自己指定这些值，可以先在 FileOutputStream 上构造一个 OutputStreamWriter。

**|---PrintWriter：**

**|---CharArrayWriter：**

**|---StringWriter：**

---------------------------------

**字节流：**

**InputStream：**是表示字节输入流的所有类的超类。

**|--- FileInputStream：**从文件系统中的某个文件中获得输入字节。哪些文件可用取决于主机环境。FileInputStream 用于读取诸如图像数据之类的原始字节流。要读取字符流，请考虑使用 FileReader。

**|---** **FilterInputStream：**包含其他一些输入流，它将这些流用作其基本数据源，它可以直接传输数据或提供一些额外的功能。

**|--- BufferedInputStream：**该类实现缓冲的输入流。

**|--- Stream：**

**|---** **ObjectInputStream：**

**|---** **PipedInputStream：**

-----------------------------------------------

**OutputStream：**此抽象类是表示输出字节流的所有类的超类。

**|---** **FileOutputStream：**文件输出流是用于将数据写入 File 或 FileDescriptor 的输出流。

**|---** **FilterOutputStream：**此类是过滤输出流的所有类的超类。

**|---** **BufferedOutputStream：**该类实现缓冲的输出流。

**|--- PrintStream：**

**|--- DataOutputStream：**

**|---** **ObjectOutputStream：**

**|---** **PipedOutputStream：**

--------------------------------

缓冲区是提高效率用的，给谁提高呢？

**BufferedWriter：**是给字符输出流提高效率用的，那就意味着，缓冲区对象建立时，必须要先有流对象。明确要提高具体的流对象的效率。

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

BufferedWriter bufw = new BufferedWriter(fw);**//让缓冲区和指定流相关联。**

for(int x=0; x<4; x++){

bufw.write(x+"abc");

bufw.newLine(); **//写入一个换行符，这个换行符可以依据平台的不同写入不同的换行符。**

bufw.flush();//对缓冲区进行刷新，可以让数据到目的地中。

}

bufw.close();**//关闭缓冲区，其实就是在关闭具体的流。**

**-----------------------------**

**BufferedReader：**

FileReader fr = new FileReader("bufdemo.txt");

BufferedReader bufr = new BufferedReader(fr);

String line = null;

while((line=bufr.readLine())!=null){ **//readLine方法返回的时候是不带换行符的。**

System.out.println(line);

}

bufr.close();

**-----------------------------**

//记住，只要一读取键盘录入，就用这句话。

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

BufferedWriter bufw = new BufferedWriter(new OutputStreamWriter(System.out));//输出到控制台

String line = null;

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

if("over".equals(line))

break;

bufw.write(line.toUpperCase());//将输入的字符转成大写字符输出

bufw.newLine();

bufw.flush();

}

bufw.close();

bufr.close();

------------------------------

**流对象：**其实很简单，就是读取和写入。但是因为功能的不同，流的体系中提供N多的对象。那么开始时，到底该用哪个对象更为合适呢？这就需要明确流的操作规律。

**流的操作规律：**

**1，明确源和目的。**

需要读取，可以使用两个体系：InputStream、Reader；

需要写入，可以使用两个体系：OutputStream、Writer；

**2，操作的数据是否是纯文本数据？**

如果是：Reader、Writer

如果不是：InputStream、OutputStream

**3，虽然确定了一个体系，但是该体系中有太多的对象，到底用哪个呢？**

明确操作的数据设备。

数据源对应的设备：硬盘(File)，内存(数组)，键盘(System.in)

数据汇对应的设备：硬盘(File)，内存(数组)，控制台(System.out)。

**4，需要在基本操作上附加其他功能吗？**比如缓冲。

如果需要就进行装饰。

**转换流特有功能：**转换流可以将字节转成字符，原因在于，将获取到的字节通过查编码表获取到指定对应字符。

**转换流的最强功能就是基于 字节流 + 编码表 。没有转换，没有字符流。**

发现转换流有一个子类就是操作文件的字符流对象：

InputStreamReader

|--FileReader

OutputStreamWriter

|--FileWrier

想要操作文本文件，必须要进行编码转换，而编码转换动作转换流都完成了。所以操作文件的流对象只要继承自转换流就可以读取一个字符了。

**但是子类有一个局限性，就是子类中使用的编码是固定的，是本机默认的编码表，对于简体中文版的系统默认码表是GBK。**

FileReader fr = new FileReader("a.txt");

InputStreamReader isr = new InputStreamReader(new FileInputStream("a.txt"),"gbk");

以上两句代码功能一致，

如果仅仅使用平台默认码表，就使用FileReader fr = new FileReader("a.txt"); //因为简化。

如果需要制定码表，必须用转换流。

转换流 = 字节流+编码表。

转换流的子类File = 字节流 + 默认编码表。

**凡是操作设备上的文本数据，涉及编码转换，必须使用转换流。**

-----------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------------

Java.util.**Properties：**一个可以将键值进行持久化存储的对象。Map--Hashtable的子类。

Map

|--Hashtable

|--Properties：用于属性配置文件，键和值都是字符串类型。

**特点：1：可以持久化存储数据。2：键值都是字符串。3：一般用于配置文件。**

**|-- load()：**将流中的数据加载进集合。

**原理：**其实就是将读取流和指定文件相关联，并读取一行数据，因为数据是规则的key=value，所以获取一行后，通过 = 对该行数据进行切割，左边就是键，右边就是值，将键、值存储到properties集合中。

**|-- store()：**写入各个项后，刷新输出流。

**|-- list()：**将集合的键值数据列出到指定的目的地。

-------------------------------------------------------------------------------------------------

**以下介绍IO包中扩展功能的流对象：基本都是装饰设计模式。**

Java.io.outputstream.**PrintStream：**打印流

1：提供了更多的功能，比如打印方法。可以直接打印任意类型的数据。

2：它有一个自动刷新机制，创建该对象，指定参数，对于指定方法可以自动刷新。

3：它使用的本机默认的字符编码.

4：该流的print方法不抛出IOException。

**该对象的构造函数。**

PrintStream(File file) ：创建具有指定文件且不带自动行刷新的新打印流。

PrintStream(File file, String csn) ：创建具有指定文件名称和字符集且不带自动行刷新的新打印流。

PrintStream(OutputStream out) ：创建新的打印流。

PrintStream(OutputStream out, boolean autoFlush) ：创建新的打印流。

PrintStream(OutputStream out, boolean autoFlush, String encoding) ：创建新的打印流。

PrintStream(String fileName) ：创建具有指定文件名称且不带自动行刷新的新打印流。

PrintStream(String fileName, String csn)

**PrintStream可以操作目的：**1：File对象。2：字符串路径。3：字节输出流。

前两个都JDK1.5版本才出现。而且在操作文本文件时，可指定字符编码了。

当目的是一个字节输出流时，如果使用的println方法，可以在printStream对象上加入一个**true**参数。这样对于println方法可以进行自动的刷新，而不是等待缓冲区满了再刷新。最终print方法都将具体的数据转成字符串，而且都对IO异常进行了内部处理。

既然操作的数据都转成了字符串，那么使用PrintWriter更好一些。因为PrintWrite是字符流的子类，可以直接操作字符数据，同时也可以指定具体的编码。

--------------------------------------------------------

**PrintWriter：**具备了PrintStream的特点同时，还有自身特点：

该对象的目的地有四个：1：File对象。2：字符串路径。3：字节输出流。4：字符输出流。

**开发时尽量使用PrintWriter。**

**方法中直接操作文件的第二参数是编码表。**

**直接操作输出流的，第二参数是自动刷新。**

//**读取键盘录入将数据转成大写显示在控制台.**

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));//**源：键盘输入**

//**目的：把数据写到文件中，还想自动刷新。**

PrintWriter out = new PrintWriter(**new FileWriter("out.txt"),true**);//设置true后自动刷新

String line = null;

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

if("over".equals(line))

break;

out.println(line.toUpperCase());//转大写输出

}

//**注意：**System.in，System.out这两个标准的输入输出流，在jvm启动时已经存在了。随时可以使用。当jvm结束了，这两个流就结束了。但是，当使用了显示的close方法关闭时，这两个流在提前结束了。

out.close();

bufr.close();

------------------------------------------------------------------------------------------------

**SequenceInputStream：序列流**，作用就是将多个读取流合并成一个读取流。**实现数据合并。**

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。

这样做，可以更方便的操作多个读取流，其实这个序列流内部会有一个有序的集合容器，用于存储多个读取流对象。

该对象的构造函数参数是枚举，想要获取枚举，需要有Vector集合，但不高效。需用ArrayList，但ArrayList中没有枚举，只有自己去创建枚举对象。

但是方法怎么实现呢？因为枚举操作的是具体集合中的元素，所以无法具体实现，但是枚举和迭代器是功能一样的，所以，可以用迭代替代枚举。

**合并原理：**多个读取流对应一个输出流。

**切割原理：**一个读取流对应多个输出流。

import java.io.\*;

import java.util.\*;

class SplitFileDemo{

private static final String CFG = ".properties";

private static final String SP = ".part";

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

File file = new File("c:\\0.bmp");

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

meger(dir);

}

**//数据的合并。**

public static void meger(File dir)throws IOException{

if(!(dir.exists() && dir.isDirectory()))

throw new RuntimeException("指定的目录不存在，或者不是正确的目录");

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

if(files.length==0)

throw new RuntimeException("扩展名.proerpties的文件不存在");

**//获取到配置文件**

File config = files[0];

**//获取配置文件的信息。**

Properties prop = new Properties();

FileInputStream fis = new FileInputStream(config);

prop.load(fis);

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

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

//--------------------------

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

if(partFiles.length!=partcount)

throw new RuntimeException("缺少碎片文件");

//---------------------

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

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

al.add(new FileInputStream(new File(dir,x+SP)));

}

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

SequenceInputStream sis = new SequenceInputStream(en);

File file = new File(dir,fileName);

FileOutputStream fos = new FileOutputStream(file);

byte[] buf = new byte[1024];

int len = 0;

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

fos.write(buf,0,len);

}

fos.close();

sis.close();

}

**//带有配置信息的数据切割。**

public static void splitFile(File file)throws IOException{

//用一个读取流和文件关联。

FileInputStream fis = new FileInputStream(file);

//创建目的地。因为有多个。所以先创建引用。

FileOutputStream fos = null;

//指定碎片的位置。

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

if(!dir.exists())

dir.mkdir();

//碎片文件大小引用。

File f = null;

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

//因为切割完的文件通常都有规律的。为了简单标记规律使用计数器。

int count = 0;

int len = 0;

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

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

fos = new FileOutputStream(f);

fos.write(buf,0,len);

fos.close();

}

//碎片文件生成后，还需要定义配置文件记录生成的碎片文件个数。以及被切割文件的名称。

//定义简单的键值信息，可是用Properties。

String filename = file.getName();

Properties prop = new Properties();

prop.setProperty("filename",filename);

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

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

fos = new FileOutputStream(config);

prop.store(fos,"");

fos.close();

fis.close();

}

}

class SuffixFilter implements FileFilter{

private String suffix;

SuffixFilter(String suffix){

this.suffix = suffix;

}

public boolean accept(File file){

return file.getName().endsWith(suffix);

}

}

-----------------------------------------------------------------------------------------------

**RandomAccessFile:**

特点：

1：该对象即可读取，又可写入。

2：该对象中的定义了一个大型的byte数组，通过定义指针来操作这个数组。

3：可以通过该对象的**getFilePointer()**获取指针的位置，通过**seek()**方法设置指针的位置。

4：该对象操作的源和目的必须是文件。

5：其实该对象内部封装了字节读取流和字节写入流。

**注意：**实现随机访问，最好是数据有规律。

class RandomAccessFileDemo{

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

write();

read();

randomWrite();

}

**//随机写入数据，可以实现已有数据的修改。**

public static void randomWrite()throws IOException{

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

raf.seek(8\*4);

System.out.println("pos :"+raf.getFilePointer());

raf.write("王武".getBytes());

raf.writeInt(102);

raf.close();

}

public static void read()throws IOException{

RandomAccessFile raf = new RandomAccessFile("random.txt","r");//只读模式。

//指定指针的位置。

raf.seek(8\*1);//实现随机读取文件中的数据。注意：数据最好有规律。

System.out.println("pos1 :"+raf.getFilePointer());

byte[] buf = new byte[4];

raf.read(buf);

String name = new String(buf);

int age = raf.readInt();

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

System.out.println("pos2 :"+raf.getFilePointer());

raf.close();

}

public static void write()throws IOException{

//rw：当这个文件不存在，会创建该文件。当文件已存在，不会创建。所以不会像输出流一样覆盖。

RandomAccessFile raf = new RandomAccessFile("random.txt","rw");//rw读写模式

//往文件中写入人的基本信息，姓名，年龄。

raf.write("张三".getBytes());

raf.writeInt(97);

raf.close();

}

}

------------------------------------------------------------------------------------------------

**管道流：**管道读取流和管道写入流可以像管道一样对接上，管道读取流就可以读取管道写入流写入的数据。

**注意：**需要加入多线程技术，因为单线程，先执行read，会发生死锁，因为read方法是阻塞式的，没有数据的read方法会让线程等待。

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

PipedInputStream pipin = new PipedInputStream();

PipedOutputStream pipout = new PipedOutputStream();

pipin.connect(pipout);

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

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

}

------------------------------------------------------------------------------------------------

**对象的序列化：**目的：将一个具体的对象进行持久化，写入到硬盘上。

**注意：静态数据不能被序列化**，因为静态数据不在堆内存中，是存储在静态方法区中。

**如何将非静态的数据不进行序列化？**用**transient** 关键字修饰此变量即可。

**Serializable：**用于启动对象的序列化功能，可以强制让指定类具备序列化功能，该接口中没有成员，这是一个标记接口。这个标记接口用于给序列化类提供UID。这个uid是依据类中的成员的数字签名进行运行获取的。如果不需要自动获取一个uid，可以在类中，手动指定一个名称为serialVersionUID id号。依据编译器的不同，或者对信息的高度敏感性。最好每一个序列化的类都进行手动显示的UID的指定。

import java.io.\*;

class ObjectStreamDemo {

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

writeObj();

readObj();

}

public static void readObj()throws Exception{

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("obj.txt"));

Object obj = ois.readObject();**//读取一个对象。**

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

}

public static void writeObj()throws IOException{

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

oos.writeObject(new Person("lisi",25)); **//写入一个对象。**

oos.close();

}

}

class Person **implements Serializable**{

private static final long serialVersionUID = 42L;

private **transient** String name;**//用transient修饰后name将不会进行序列化**

public int age;

Person(String name,int age){

this.name = name;

this.age = age;

}

public String toString(){

return name+"::"+age;

}

}

-----------------------------------------------------------------------------------------------

**DataOutputStream、DataInputStream：**专门用于操作基本数据类型数据的对象。

DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.txt"));

dos.writeInt(256);

dos.close();

DataInputStream dis = new DataInputStream(new FileInputStream("data.txt"));

int num = dis.readInt();

System.out.println(num);

dis.close();

-----------------------------------------------------------------------------------------------

**ByteArrayInputStream：源：内存**

**ByteArrayOutputStream：目的：内存。**

这两个流对象不涉及底层资源调用，操作的都是内存中数组，所以不需要关闭。

直接操作字节数组就可以了，为什么还要把数组封装到流对象中呢？因为数组本身没有方法，只有一个length属性。为了便于数组的操作，将数组进行封装，对外提供方法操作数组中的元素。

**对于数组元素操作无非两种操作：设置（写）和获取（读），**而这两操作正好对应流的读写操作。这两个对象就是使用了流的读写思想来操作数组。

**//创建源：**

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

**//创建目的：**

ByteArrayOutputStream bos = new ByteArrayOutputStream();

int ch = 0;

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

bos.write(ch);

}

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

**进程**：正在进行中的程序。其实进程就是一个应用程序运行时的内存分配空间。

**线程**：其实就是进程中一个程序执行控制单元，一条执行路径。进程负责的是应用程序的空间的标示。线程负责的是应用程序的执行顺序。

一个进程至少有一个线程在运行，当一个进程中出现多个线程时，就称这个应用程序是多线程应用程序，每个线程在栈区中都有自己的执行空间，自己的方法区、自己的变量。

jvm在启动的时，首先有一个**主线程**，负责程序的执行，调用的是main函数。主线程执行的代码都在main方法中。

当产生垃圾时，收垃圾的动作，是不需要主线程来完成，因为这样，会出现主线程中的代码执行会停止，会去运行垃圾回收器代码，效率较低，所以由单独一个线程来负责垃圾回收。

**随机性的原理**：因为cpu的快速切换造成，哪个线程获取到了cpu的执行权，哪个线程就执行。

返回当前线程的名称：**Thread.currentThread().getName()**

线程的名称是由：Thread-编号定义的。编号从0开始。

线程要运行的代码都统一存放在了**run方法**中。

线程要运行必须要通过类中指定的方法开启。**start方法**。（启动后，就多了一条执行路径）

**start方法：**1）、启动了线程；2）、让jvm调用了run方法。

**创建线程的第一种方式：继承Thread ，由子类复写run方法。**

步骤：

1，定义类继承Thread类；

2，目的是复写run方法，将要让线程运行的代码都存储到run方法中；

3，通过创建Thread类的子类对象，创建线程对象；

4，调用线程的start方法，开启线程，并执行run方法。

**线程状态：**

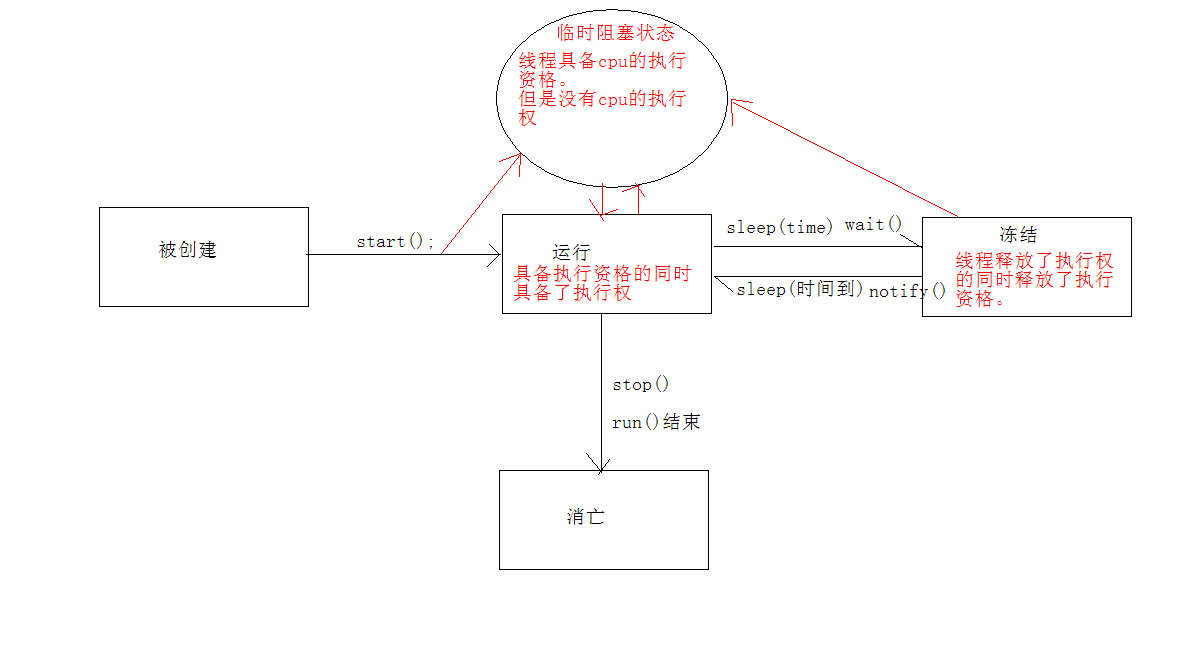
**被创建：**start()

**运行：**具备执行资格，同时具备执行权；

**冻结：**sleep(time),wait()—notify()唤醒；线程释放了执行权，同时释放执行资格；

**临时阻塞状态：**线程具备cpu的执行资格，没有cpu的执行权；

**消亡：**stop()



**创建线程的第二种方式：实现一个接口Runnable。**

步骤：

1，定义类实现Runnable接口。

2，覆盖接口中的run方法（用于封装线程要运行的代码）。

3，通过Thread类创建线程对象；

4，将实现了Runnable接口的子类对象作为实际参数传递给Thread类中的构造函数。

为什么要传递呢？因为要让线程对象明确要运行的run方法所属的对象。

5，调用Thread对象的start方法。开启线程，并运行Runnable接口子类中的run方法。

**Ticket t = new Ticket();**

/\*

直接创建Ticket对象，并不是创建线程对象。

因为创建对象只能通过new Thread类，或者new Thread类的子类才可以。

所以最终想要创建线程。既然没有了Thread类的子类，就只能用Thread类。

\*/

**Thread t1 = new Thread(t);** //创建线程。

/\*

只要将t作为Thread类的构造函数的实际参数传入即可完成线程对象和t之间的关联

为什么要将t传给Thread类的构造函数呢？其实就是为了明确线程要运行的代码run方法。

\*/

**t1.start();**

**为什么要有Runnable接口的出现？**

**1：**通过继承Thread类的方式，可以完成多线程的建立。但是这种方式有一个局限性，如果一个类已经有了自己的父类，就不可以继承Thread类，因为java单继承的局限性。

可是该类中的还有部分代码需要被多个线程同时执行。这时怎么办呢？

只有对该类进行额外的功能扩展，java就提供了一个接口Runnable。这个接口中定义了run方法，其实run方法的定义就是为了存储多线程要运行的代码。

所以，通常创建线程都用第二种方式。

**因为实现Runnable接口可以避免单继承的局限性。**

**2：**其实是将不同类中需要被多线程执行的代码进行抽取。将多线程要运行的代码的位置单独定义到接口中。为其他类进行功能扩展提供了前提。

所以Thread类在描述线程时，内部定义的run方法，也来自于Runnable接口。

**实现Runnable接口可以避免单继承的局限性。**而且，继承Thread，是可以对Thread类中的方法，进行子类复写的。但是不需要做这个复写动作的话，只为定义线程代码存放位置，实现Runnable接口更方便一些。所以**Runnable接口将线程要执行的任务封装成了对象**。

-------------------------------------------------------

**//面试**

new Thread(new Runnable(){ //匿名

public void run(){

System.out.println("runnable run");

}

}

{

public void run(){

System.out.println("subthread run");

}

}.start(); //**结果：subthread run**

---------------------------------------------------------

**Try {**

**Thread.sleep(10);**

**}catch(InterruptedException e){}**// 当刻意让线程稍微停一下，模拟cpu 切换情况。

**多线程安全问题的原因**：

通过图解：发现一个线程在执行多条语句时，并运算同一个数据时，在执行过程中，其他线程参与进来，并操作了这个数据。导致到了错误数据的产生。

**涉及到两个因素：**

**1，多个线程在操作共享数据。**

**2，有多条语句对共享数据进行运算。**

原因：这多条语句，在某一个时刻被一个线程执行时，还没有执行完，就被其他线程执行了。

**解决安全问题的原理**：

只要将操作共享数据的语句在某一时段让一个线程执行完，在执行过程中，其他线程不能进来执行就可以解决这个问题。

如何进行多句操作共享数据代码的封装呢？

**java中提供了一个解决方式：就是同步代码块。**

格式：

**synchronized(对象) { // 任意对象都可以。这个对象就是锁。**

**需要被同步的代码；**

**}**

---------------------------------------------------------------

**同步：★★★★★**

**好处：**解决了线程安全问题。

**弊端**：相对降低性能，因为判断锁需要消耗资源，产生了死锁。

**定义同步是有前提的**：

1，必须要有两个或者两个以上的线程，才需要同步。

2，多个线程必须保证使用的是同一个锁。

同步的第二种表现形式：

**同步函数**：其实就是将同步关键字定义在函数上，让函数具备了同步性。

**同步函数是用的哪个锁呢？**

通过验证，函数都有自己所属的对象this，所以**同步函数所使用的锁就是this锁**。

**当同步函数被static修饰时，这时的同步用的是哪个锁呢？**

静态函数在加载时所属于类，这时有可能还没有该类产生的对象，但是该类的字节码文件加载进内存就已经被封装成了对象，这个对象就是**该类的字节码文件对象**。

所以静态加载时，只有一个对象存在，那么静态同步函数就使用的这个对象。

这个对象就是 **类名.class**

**同步代码块和同步函数的区别？**

同步代码块使用的锁可以是任意对象。

**同步函数使用的锁是this，静态同步函数的锁是该类的字节码文件对象**。

**在一个类中只有一个同步，可以使用同步函数。如果有多同步，必须使用同步代码块，来确定不同的锁。所以同步代码块相对灵活一些。**

-------------------------------------------------------

**★考点问题：请写一个延迟加载的单例模式？写懒汉式；当出现多线程访问时怎么解决？加同步，解决安全问题；效率高吗？不高；怎样解决？通过双重判断的形式解决。**

//懒汉式：延迟加载方式。

当多线程访问懒汉式时，因为懒汉式的方法内对共性数据进行多条语句的操作。所以容易出现线程安全问题。为了解决，加入同步机制，解决安全问题。但是却带来了效率降低。

**为了效率问题，通过双重判断的形式解决。**

class Single{

private static Single s = null;

private Single(){}

public static Single getInstance(){ //**锁是谁？字节码文件对象；**

**if(s == null){**

**synchronized(Single.class){**

**if(s == null)**

**s = new Single();**

**}**

**}**

return s;

}

}

---------------------------------------------------------

**同步死锁**：通常只要将同步进行嵌套，就可以看到现象。同步函数中有同步代码块，同步代码块中还有同步函数。

**线程间通信**：思路：多个线程在操作同一个资源，但是操作的动作却不一样。

1：将资源封装成对象。

2：将线程执行的任务(任务其实就是run方法。)也封装成对象。

**等待唤醒机制：**涉及的方法：

**wait:**将同步中的线程处于冻结状态。释放了执行权，释放了资格。同时将线程对象存储到线程池中。

**notify：**唤醒线程池中某一个等待线程。

**notifyAll:**唤醒的是线程池中的所有线程。

注意：

1：这些方法都需要定义在同步中。

2：因为这些方法必须要标示所属的锁。

你要知道 A锁上的线程被wait了,那这个线程就相当于处于A锁的线程池中，只能A锁的notify唤醒。

3：这三个方法都定义在Object类中。为什么操作线程的方法定义在Object类中？

因为这三个方法都需要定义同步内，并标示所属的同步锁，既然被锁调用，而锁又可以是任意对象，那么能被任意对象调用的方法一定定义在Object类中。

**wait和sleep区别：** 分析这两个方法：从执行权和锁上来分析：

wait：可以指定时间也可以不指定时间。不指定时间，只能由对应的notify或者notifyAll来唤醒。

sleep：必须指定时间，时间到自动从冻结状态转成运行状态(临时阻塞状态)。

**wait：线程会释放执行权，而且线程会释放锁。**

**Sleep：线程会释放执行权，但不是不释放锁。**

**线程的停止**：通过stop方法就可以停止线程。但是这个方式过时了。

停止线程：原理就是：让线程运行的代码结束，也就是结束run方法。

怎么结束run方法？一般run方法里肯定定义循环。所以只要结束循环即可。

第一种方式：**定义循环的结束标记。**

第二种方式：如果线程处于了冻结状态，是不可能读到标记的，这时就需要通过Thread类中的interrupt方法，将其冻结状态强制清除。让线程恢复具备执行资格的状态，让线程可以读到标记，并结束。

**---------< java.lang.Thread >----------**

**interrupt()：**中断线程。

**setPriority(int newPriority)：**更改线程的优先级。

**getPriority()：**返回线程的优先级。

**toString()：**返回该线程的字符串表示形式，包括线程名称、优先级和线程组。

**Thread.yield()：**暂停当前正在执行的线程对象，并执行其他线程。

**setDaemon(true)：**将该线程标记为守护线程或用户线程。将该线程标记为守护线程或用户线程。当正在运行的线程都是守护线程时，Java 虚拟机退出。该方法必须在启动线程前调用。

**join**：临时加入一个线程的时候可以使用join方法。

当A线程执行到了B线程的join方式。A线程处于冻结状态，释放了执行权，B开始执行。A什么时候执行呢？只有当B线程运行结束后，A才从冻结状态恢复运行状态执行。

-----------------------------------------------------------

**Lock接口：多线程在JDK1.5版本升级时，推出一个接口Lock接口。**

解决线程安全问题使用同步的形式，(同步代码块，要么同步函数)其实最终使用的都是锁机制。

到了后期版本，直接将锁封装成了对象。线程进入同步就是具备了锁，执行完，离开同步，就是释放了锁。

在后期对锁的分析过程中，发现，获取锁，或者释放锁的动作应该是锁这个事物更清楚。所以将这些动作定义在了锁当中，并把锁定义成对象。

所以**同步是隐示的锁操作，而Lock对象是显示的锁操作**，它的出现就替代了同步。

在之前的版本中使用Object类中wait、notify、notifyAll的方式来完成的。那是因为同步中的锁是任意对象，所以操作锁的等待唤醒的方法都定义在Object类中。

而现在锁是指定对象Lock。所以查找等待唤醒机制方式需要通过Lock接口来完成。而Lock接口中并没有直接操作等待唤醒的方法，而是将这些方式又单独封装到了一个对象中。这个对象就是**Condition**，将Object中的三个方法进行单独的封装。并提供了功能一致的方法 **await()、signal()、signalAll()**体现新版本对象的好处。

**< java.util.concurrent.locks > Condition接口：await()、signal()、signalAll()；**

--------------------------------------------------------

class BoundedBuffer {

**final Lock lock = new ReentrantLock();**

final Condition notFull = **lock.newCondition();**

final Condition notEmpty = **lock.newCondition();**

final Object[] items = new Object[100];

int putptr, takeptr, count;

public void put(Object x) throws InterruptedException {

**lock.lock();**

**try {**

while (count == items.length)

**notFull.await();**

items[putptr] = x;

if (++putptr == items.length) putptr = 0;

++count;

**notEmpty.signal();**

**}**

**finally {**

**lock.unlock();**

**}**

}

public Object take() throws InterruptedException {

**lock.lock();**

**try {**

while (count == 0)

**notEmpty.await();**

Object x = items[takeptr];

if (++takeptr == items.length) takeptr = 0;

--count;

**notFull.signal();**

return x;

**}**

**finally {**

**lock.unlock();**

**}**

}

}

### 五、作业讲解

### 六、当日课程内容总结

### 本次直播课(后)复习巩固补充题

1. 将一个文件夹中的内容复制到其它目录中。（比如:把D:\\chenhao中的内容复制一份到 e:\\zhangsan文件夹中）

### 八、下次自主学习任务布置



需要通过平台提交作业。完成平台上所布置的作业。