# 1、IO流总结

## 1. 字节流

1.1 输入字节流：

- - - - | InputStream 所有输入字节流的基类 抽象类

- - - - - - - - | FileInputStream 读取文件数据的输入字节流

- - - - - - - - | BufferedInputStream 缓冲输入字节流。该类出现的目的是为了提高读取文件数据的效率。这个类其实只不过是在内部维护了一个8kb的字节数组而已。

1.2 输出字节流：

- - - - | OutputStream 所有输出字节流的基类。 抽象类。

- - - - - - - - | FileOutputStream 向文件输出数据的输出字节流

- - - - - - - - | BufferedOutputStream 缓冲输出字节流。该类出现的目的也是为了提高向文件写数据的效率。 这个类的也只不过是在内部维护了一个8kb的字节数组而已。

## 2. 字符流 : 字符流 = 字节流 + 编码（解码）

2.1 输入字符流:

- - - - | Reader 所有输入字符流的基类。 抽象类。

- - - - - - - - | FileReader 读取文件数据的输入字符流。

- - - - - - - - | BufferedReader 缓冲输入字符流。该类出现的目的是为了提高读取文件数据的效率与拓展FileReader的(readLine)功能。 这个类的也只不过是在内部维护了一个8kb的字符数组而已。

2.2 输出字符流:

- - - - | Writer 所有输出字符流的基类。 抽象类

- - - - - - - - | FileWriter 向文件输出数据的输出字符流

- - - - - - - - | BufferedWriter 缓冲输出字符流。该类出现的目的是为了提高写文件数据的效率与拓展FileWriter的(newLine)功能.

# 2、SequenceInputStream

- - - - | InputStream 所有输入字节流的基类 抽象类

- - - - - - - - | FileInputStream 读取文件数据的输入字节流

- - - - - - - - | BufferedInputStream 缓冲输入字节流。

- - - - - - - - | SequenceInputStream 序列输入字节流

## 1、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。

## 2、SequenceInputStream 的构造方法：

* SequenceInputStream(Enumeration<? extends InputStream> e)

通过记住参数来初始化新创建的 SequenceInputStream，该参数必须是生成运行时类型为 InputStream 对象的 Enumeration 型参数。

* SequenceInputStream(InputStream s1, InputStream s2)

通过记住这两个参数来初始化新创建的 SequenceInputStream（将按顺序读取这两个参数，先读取 s1，然后读取 s2），以提供从此 SequenceInputStream 读取的字节。

## 3、SequenceInputStream的使用步骤

1. 找到目标文件
2. 搭建数据通道
   1. 创建Vector集合，将携带数据的流对象以元素的形式添加到集合中
   2. 使用Vector集合的elements()方法获得迭代器。
   3. 将迭代器传入SequenceInputStream序列流中
3. 传输数据
4. 关闭资源

## 4、案例

需求：将E:\\aa\\music下的所有文件合并成一个文件，然后保存到E:\\aa下

|  |
| --- |
| // 将E:\\aa\\music下的所有文件合并成一个文件，然后保存到E:\\aa下 **public class** Dome3 {  **public static void** main(String[] args) **throws** IOException {  // 1.找到目标文件  File inFile = **new** File(**"E:\\aa\\music"**);// 源文件1  File outFile = **new** File(**"E:\\aa\\合并.mp3"**);//输出目标文件   // 2.搭建传输通道  FileOutputStream fileOutputStream = **new** FileOutputStream(outFile);//输出流   Vector<FileInputStream> vector = **new** Vector<FileInputStream>();//创建Vector集合  File[] files = inFile.listFiles();//获取music文件夹下的所有文件的集合  // 读取文件中的数据，并加入vector集合中  **for** (File file : files) {//遍历集合  vector.add(**new** FileInputStream(file));//将所有文件都添加到vector集合中  }  Enumeration<FileInputStream> elements = vector.elements();//获取集合的迭代器  SequenceInputStream sequenceInputStream = **new** SequenceInputStream(elements);//将迭代器传入SequenceInputStream序列流中    **byte**[] bytes = **new byte**[1024 \* 10];//缓冲数组，用于存储输入流读取的数据，提高读取效率  **int** len = 0;// 用于存储每次从文件中读取的字节个数  **while** ((len = sequenceInputStream.read(bytes)) != -1) {//从序列流中读取数据放入缓存数组中  fileOutputStream.write(bytes, 0, len);//将缓存数组中的数据写出到硬盘中。  }   // 释放资源，先开后关原则  sequenceInputStream.close();  fileOutputStream.close();  } } |

# 3、切割合并

需求：

将一首歌曲切割成n分，要求没分都可以正常播放

再讲切割后的n份歌曲合并成一首歌，并不影响正常播放

|  |
| --- |
| **package** character;  **import** java.io.\*; **import** java.util.Enumeration; **import** java.util.Vector;  // 将一首歌曲切割开，然后再合并起来，要求不影响数据大小 **public class** Dome2 {  // 程序入口  **public static void** main(String[] args) {  *incise*();//切割文件  *merge*();//合并文件  }   // 切割文件  **public static void** incise() {  FileInputStream fileInputStream = **null**;//输入流  FileOutputStream fileOutputStream = **null**;//输出流   **try** {  // 找到目标文件  File sourceFile = **new** File(**"E:\\aa\\野狼disco.mp3"**);//目标文件  //File sourceFile = new File("E:\\aa\\A.java");//读取目标文件  File dir = **new** File(**"E:\\aa\\music"**);//输出目标文件   // 判断输出目标文件夹是否存在,如果不存在则创建dir文件夹，如果存在则删除其中相关文件  **if** (!dir.exists() || dir.isFile()) {  dir.mkdirs();//如果dir不存在或dir是文件，则创建dir文件夹  } **else** {  // 如果dir是文件夹  **for** (File f : dir.listFiles()) {// 迭代文件夹中的文件  **if** (f.getName().matches(**"^野狼disco\\d+\\.mp3$"**)) {  f.delete();  }  }  }   // 搭建通道  fileInputStream = **new** FileInputStream(sourceFile);   // 传输数据  **byte**[] bytes = **new byte**[1024 \* 1000];//缓冲数组  **int** len = 0;//存储每次读取的数据  **int** count = 1;//记录切割数量  **while** ((len = fileInputStream.read(bytes)) != -1) {  // 每当从缓冲数组写出一次数据，就新建一个文件存储，所以需要重新创建输出流对象，从新找到输出目标文件  fileOutputStream = **new** FileOutputStream(**new** File(dir, **"野狼disco"** + count + **".mp3"**));  fileOutputStream.write(bytes, 0, len);//向硬盘写出数据  fileOutputStream.close();//需要创建新文件时，需要将不再使用的流关闭  count++;  }   } **catch** (FileNotFoundException e) {  **throw new** RuntimeException(e);  } **catch** (IOException e) {  **throw new** RuntimeException(e);  } **finally** {  **try** {  // 关闭资源  fileInputStream.close();  } **catch** (IOException e) {  **throw new** RuntimeException(e);  }  }  }    //合并文件  **public static void** merge() {  // 找到目标文件  File sourceFile = **null**;//输入目标  File destinationFile = **null**;//输出目标  FileOutputStream fileOutputStream = **null**;//输出流  SequenceInputStream sequenceInputStream = **null**;//序列流  File[] files = **null**;//用于接受文件夹中文件的数组  //搭建数据通道  **try** {  sourceFile = **new** File(**"E:\\aa\\music"**);  destinationFile = **new** File(**"E:\\aa\\野狼disco(合并).mp3"**);  fileOutputStream = **new** FileOutputStream(destinationFile, **true**);//文件输出目标   // 判断目标文件是否存在  **if** (sourceFile.exists()) {  files = sourceFile.listFiles();// 获取文件夹中的文件集合  System.***out***.println(**"获取文件夹中文件"**);  } **else** {  System.***out***.println(**"music文件夹不存在..."**);  **return**;  }  //判断输出目标文件是否存在  **if** (!destinationFile.exists()) {  destinationFile.mkdirs();  }   // 传输数据  Vector<FileInputStream> vector = **new** Vector<FileInputStream>();  // 遍历文件，并将文件数据添加到集合中  **for** (File file : files) {  **if** (file.getName().matches(**"^野狼disco\\d+\\.mp3$"**)) {  vector.add(**new** FileInputStream(file));//将输入流读取到的符合规则的文件的数据，添加到vector集合中。  }  }  Enumeration<FileInputStream> elements = vector.elements();//通过Vector获取迭代器  sequenceInputStream = **new** SequenceInputStream(elements);//将迭代器传入序列流   **byte**[] bytes = **new byte**[1024 \* 8];//缓冲数组  **int** len = 0;//用于存储每次读取到的数据  **while** ((len = sequenceInputStream.read(bytes)) != -1) {// 通过序列流读取文件  fileOutputStream.write(bytes, 0, len);  }  } **catch** (FileNotFoundException e) {  **throw new** RuntimeException(e);  } **catch** (IOException e) {  **throw new** RuntimeException(e);  } **finally** {  **try** {  fileOutputStream.close();  sequenceInputStream.close();  } **catch** (IOException e) {  **throw new** RuntimeException(e);  }  }  } } |

# 4、对象流、序列化Serializable

## 1.对象流的作用

对象的输入输出流主要作用是 读取和写出对象的信息。对象信息一旦写出到硬盘文件中，就可以做到持久化。

对象输入流：ObjectInputStream

对象输出流：ObjectOutputStream

## 2.对象流的使用步骤

1.找到目标文件

2.搭建数据通道

2.1 创建数据输出/输入流对象

2.2 创建对象输出/输入流对象，并传入数据流对象

3.把对象写出/读入

4.关闭资源

## 3. 序列化与反序列化案例

需求：将用户的信息进行持久化保存，并读取硬盘文件中的用户信息

|  |
| --- |
| **package** character;  **import** java.io.\*; **import** java.util.ArrayList; **import** java.util.List;  **public class** Dome4 {  **public static void** main(String[] args) **throws** IOException, ClassNotFoundException {  File file = **new** File(**"E:\\aa\\obj.txt"**);  List<User> inUsers = **new** ArrayList<User>();//用户集合  // 用户对象  User user1 = **new** User();  User user2 = **new** User();  User user3 = **new** User();  User user4 = **new** User();   user1.setUsername(**"aaa"**);  user2.setUsername(**"bbb"**);  user3.setUsername(**"ccc"**);  user4.setUsername(**"ddd"**);  user1.setPassword(**"111111"**);  user2.setPassword(**"222222"**);  user3.setPassword(**"333333"**);  user4.setPassword(**"444444"**);  // 将对象添加到集合中  inUsers.add(user1);  inUsers.add(user2);  inUsers.add(user3);  inUsers.add(user4);   //序列化 将用户信息写出到硬盘文件  *testWriteInfo*(file, inUsers);   //反序列化 从硬盘中读取用户信息  List<User> outUsers = *testReadInfo*(file);  **for** (User user : outUsers) {  System.***out***.println(user.toString());  }  }   // 序列化：将内存中的数据以二进制的形式写出到硬盘文件中的过程，叫做序列化。序列化可以实现数据持久化。  **public static void** testWriteInfo(File file, List<User> lists) **throws** IOException {  // 搭建数据通道  FileOutputStream fileOutputStream = **new** FileOutputStream(file, **true**);  //判断是否为第一次写入数据  **if** (file.length() < 1) {  // 创建对象的输出流对象  ObjectOutputStream objectOutputStream = **new** ObjectOutputStream(fileOutputStream);  **for** (User user : lists) {  // 将对象写出到硬盘文件  objectOutputStream.writeObject(user);  }  // 关闭资源  objectOutputStream.close();  } **else** {  // 创建对象的输出流对象  MyOutputStream myOutputStream = **new** MyOutputStream(fileOutputStream);  **for** (User user : lists) {  // 创建对象的输出流对象  myOutputStream.writeObject(user);  }  // 关闭资源  myOutputStream.close();  }  }    // 反序列化  **public static** List<User> testReadInfo(File file) **throws** IOException, ClassNotFoundException {  // 搭建数据通道  FileInputStream fileInputStream = **new** FileInputStream(file);  // 创建对象的输入流对象  ObjectInputStream objectInputStream = **new** ObjectInputStream(fileInputStream);  // 传输数据  List<User> users = **new** ArrayList<User>();//集合，用于存储用户信息  **while** (fileInputStream.available() > 0) {  User user = (User) objectInputStream.readObject();  users.add(user);//将用户信息添加到list集合中  }  //关闭资源  objectInputStream.close();  **return** users;  } }  // 用户类 Serializable 这个接口没有任何方法存在，他只是一个标识接口而已。实现Serializable接口才能进行序列化与反序列化操作， **class** User **implements** Serializable {   // serialVersionUID是用来记录class文件的版本信息的，创建对象时肯定是依赖对象所属的类的 class文件的。SerializableUID这个数字是通过工程名、包名、类名、成员(属性、方法)计算而出的。一旦这些信息发生变化，这个数字就会发生变化。当程序的SerialVersionUID与硬盘文件中的class文件所记载的serialVersionUID不一致时，就不能实现反序列化操作。  **private static final long *serialVersionUID*** = 1L;// 表示序列化的版本号，标识符  **private** String **username**;// 用户名  **private** String **password**;// 密码   **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** String getPassword() {  **return password**;  }   **public void** setPassword(String password) {  **this**.**password** = password;  }   @Override  **public** String toString() {  **return "[ 用户名:"** + **this**.**username** + **", 密码:"** + **this**.**password** + **" ]"**;  } }  // 重写对象输出流的writeStreamHeader方法 **class** MyOutputStream **extends** ObjectOutputStream {  **public** MyOutputStream(OutputStream out) **throws** IOException {  **super**(out);  }   // 如果想要一次性反序列化硬盘上的所有对象的信息，那么就需要重写对象输出流的writeStreamHeader()方法，并在写入硬盘是判断是否是第一次写入数据。  @Override  **protected void** writeStreamHeader() **throws** IOException {  **return**;  } } |

## 4.对象流、序列化注意事项

**1. Serializable：序列化标识接口**

1.1 如果对象需要被写出到硬盘上，那么对象所属的类必须实现Serializable接口，Serializable接口中没有任何的方法，它只是一个标识接口。

1.2 如果一个对象的所属类维护了另一个类的引用，那么另一个类也需要实现Serializable接口。

**2. 序列化：就是将java对象转换成字节序列的过程（将数据从内存存储到硬盘的过程）**

2.1 transient：短暂的，不要序列化的

（1）如果对象的某个属性不详被序列化到硬盘上，那么可以用transient修饰。

（2）执行序列化时，JVM会自动忽略transient修饰的变量的初始值，而是将默认值保存到硬盘中。

（3）transient修饰符只适用于变量，不适用于方法和类。

2.2 transient不能与哪些修饰符同用？

（1）static 与transient：静态变量是属于类的，不是属于对象的，所以被static修饰的属性不参加序列化。

（2）final 与transient：final修饰的变量值是固定的，变量将直接通过值参与序列化，因此将final修饰的变量声明为transient，是没用的。

**3. 反序列化：就是将字节序列转成成对象的过程（将数据从硬盘存储到内存的过程）**

3.1对象的反序列化在创建对象的时候并不会调用对象的构造方法。

**4. serialVersionUID ：序列化的版本号，标识符**

4.1 serialVersionUID是用来记录class文件的版本信息的。当我们创建对象时，是需要依赖对象所属类的class文件的。serialVersionUID的值是通过工程名、包名、类名、成员计算而来的，一旦这些信息发生变化，这个数字也会改变。

4.2 当我们使用ObjectOutputStream进行反序列化操作时，JVM会先读取硬盘文件中的SerialVersionUID的值，然后与程序中对象所属类的SerialVersionUID进行比较，如果这两个值不相等，那么反序列化就失败了。

4.3 如果序列化与反序列化的时候可能会修改类的成员，那么最好一开始实现Serializable接口时，就指定该类的SerialVersionUID的值，这样在序列化与反序列化的时候，JVM就不会自己去计算这个类的SerialVersionUID的值了。

# Properties配置文件类

## 1、Properties配置文件类

* + 用于生产（store）配置文件或获取（Loda）配置文件信息

## 2、Properties需要注意的细节

1. 如果配置信息中一旦使用了中文，那么在使用store生成配置文件时，就必须使用字符流来解决。如果使用字节流，默认使用的是iso8859-1码表进行编码存储，这样就会导致乱码。
2. 一旦配置文件中信息发生了改变，那么必须使用store重新生成一份配置文件，否则配置文件信息不会发生变化。

## 3、Properties的使用

|  |
| --- |
| **package** other;  **import** java.io.File; **import** java.io.FileReader; **import** java.io.FileWriter; **import** java.io.IOException; **import** java.util.Map; **import** java.util.Properties; **import** java.util.Set;  **public class** Dome2 {  // 程序入口  **public static void** main(String[] args) {  // 指定目标文件  File file = **new** File(**"E:\\aa\\users.properties"**); // testProductionProperties(file);  *testLoadProperties*(file);  }   // 生成配置文件  **public static void** testProductionProperties(File file) {  **try** {  // 1.创建配置文件对象  Properties properties = **new** Properties();  // 判断文件是否存在  **if** (!file.exists()) {  file.createNewFile();  }   // 2.将键值对写入集合中  properties.setProperty(**"猪一菲"**, **"123"**);  properties.setProperty(**"猪小贤"**, **"234"**);  properties.setProperty(**"猪小布"**, **"345"**);  properties.setProperty(**"猪大力"**, **"456"**);  properties.setProperty(**"猪美嘉"**, **"567"**);  properties.setProperty(**"猪伟"**, **"678"**);   // 3.生成配置文件store(FileWriter fileWriter, String str)//第一个参数是字符输出流对象，第二个参数是描述这个配置文件的信息的字符串  properties.store(**new** FileWriter(file), **"猪猪公寓用户信息"**);   } **catch** (IOException e) {  **throw new** RuntimeException(e);  }  }    // 加载配置文件  **public static void** testLoadProperties(File file) {  // 1.创建配置文件对象  Properties properties = **new** Properties();   **try** {  // 加载配置文件信息到properties对象中  properties.load(**new** FileReader(file));  Set<Map.Entry<Object, Object>> entries = properties.entrySet();// entrySet()：返回Map中各个键值对的映射集合   // 遍历set集合  **for** (Map.Entry<Object, Object> entry : entries) {  System.***out***.println(**"key:"** + entry.getKey() + **" value:"** + entry.getValue());  }   // 改变properties配置文件中的信息后，需要重新生成一个properties配置文件  properties.setProperty(**"猪小贤"**, **"345"**);  properties.store(**new** FileWriter(file), **"猪猪公寓用户信息"**);  } **catch** (IOException e) {  e.printStackTrace();  }   } } |

## 4、案例

* + 需求： 使用properties实现本软件只能 运行三次，超过了三次之后就提示购买正版，退jvm.

|  |
| --- |
| **package** other;  **import** java.io.File; **import** java.io.FileInputStream; **import** java.io.FileOutputStream; **import** java.io.IOException; **import** java.util.Properties; **public class** Dome3 {  **public static void** main(String[] args) {  // 指定文件位置  File file = **new** File(**"E:\\aa\\count.properties"**);  *testProperties*(file);  }   //需求： 使用properties实现本软件只能 运行三次，超过了三次之后就提示购买正版，退jvm.  **public static void** testProperties(File file) {  **int** count = 0;// 用于记录使用软件的次数  Properties properties = **new** Properties();// 创建配置文件对象  **try** {  // 判断配置文件是否存在  **if** (!file.exists()) {  file.createNewFile();  }  // 加载配置文件  properties.load(**new** FileInputStream(file));  String CountValue = properties.getProperty(**"count"**);//获取count键的value值  // 判断value是否为null  **if** (CountValue != **null**) {  count = Integer.*parseInt*(CountValue);//将字符串转换成数字  }  // 判断使用了几次了  **if** (count == 3) {  System.***out***.println(**"您已使用本软件三次，免费次数已用完，请购买正版软件"**);  System.*exit*(0);//退出系统  }  count++;  System.***out***.println(**"欢迎您第"** + count + **"次使用本软件"**);  properties.setProperty(**"count"**, count + **""**);// 设置配置文件信息  properties.store(**new** FileOutputStream(file), **"配置文件"**);// 生成配置文件   } **catch** (IOException e) {  **throw new** RuntimeException(e);  }  } } |

# 6、printStream 打印流

## 1、打印流（printStream）

* + 打印流可以打印任意类型的数据，而且打印数据之前都会先把数据转换成字符串再进行打印。

## 2、重新设置标准输出流对象

* + 默认标准的输出流就是向控制台输出的

|  |
| --- |
| // 找到file文件位置  File file = **new** File(**"E:\\aa\\printstream.log"**);  // 创建一个打印流对象 PrintStream printStream = **new** PrintStream(file);   // 1.打印流可以打印任意类型的数据，并且在打印之前将数据转换成字符串再进行打印  printStream.print(**'a'**);  // 2.重新设置标准输出流对象，默认标准输出流对象就是向控制台输出 System.***out***.println();// 默认标准输出流对象就是向控制台输出 System.*setOut*(printStream);// 重新设置了标准输出流对象 |

## 3、收集异常的日志信息

|  |
| --- |
| // 3.收集异常的日志信息 File logFile = **new** File(**"E:\\aa\\2020年1月30日.log"**);//日志文件 // 创建打印流对象 PrintStream logPrintStream = **new** PrintStream(**new** FileOutputStream(logFile, **true**));//参数使用FileOutputStream是为了能够将异常信息追加到日志中，而不是先清空再写入 **try** {  **int** count = 4 / 0;// by zero异常  **int** arr[] = **null**;  System.***out***.println(arr.**length**);// 空指针异常 } **catch** (Exception e) {  e.printStackTrace(logPrintStream);//该方法是将此Throwable及其追踪输出到指定的输出流对象中 } |

# 7、编码与解码

## 1、编码与解码

* + 编码：将字符串转变成码值的过程,叫做编码
  + 解码：将码值转变成字符串的过程,叫做解码。

注意：编码与解码的过程中，我们都要使用统一的码表，否则容易出现乱码。

## 2、常见的编码与解码

* Unicode 指的是 utf-16
* jdk默认的编码格式是GBK
* getBytes()/getBytes(String charset)：对字符串进行编码
* String(byte[] bytes)/String(byte[] bytes,String charset)：对码值进行解码

# 8、转流

## 1、转换流只能字节流转字符流，不能字符转字节

* 输入字节流的转换流：InputStreamReader是 输入字节流 通向 输入字符流 的通道
* 输出字节流的转换流：OutputStreamWriter是 输出字节流 通向 输出字符流 的通道

## 2、转换流的作用

1、如果 字节流需要转换成字符流使用，那么可以使用转换流

2、转换流可以指定码表进行读写文件

## 3、案例

|  |
| --- |
| **package** other;  **import** java.io.\*; **public class** Dome5 {  **public static void** main(String[] args) **throws** IOException {  *test1*();  *test2*();  }    // 输入转换流  **public static void** test1() **throws** IOException {  File file = **new** File(**"E:\\aa\\users.properties"**);  FileInputStream fileInputStream = **new** FileInputStream(file);//字节输入流  InputStreamReader inputStreamReader = **new** InputStreamReader(fileInputStream, **"utf-8"**);//字节流转字符流  BufferedReader bufferedReader = **new** BufferedReader(inputStreamReader);// 向缓冲字符输入流中传入输入转换流  String line = **null**;  **while** ((line = bufferedReader.readLine()) != **null**) {  System.***out***.println(line);  }  }    // 输出转换流  **public static void** test2() **throws** IOException {  File file = **new** File(**"E:\\aa\\users.properties"**);  FileOutputStream fileOutputStream = **new** FileOutputStream(file);  OutputStreamWriter outputStreamWriter = **new** OutputStreamWriter(fileOutputStream);  String str = **"aaaaaaaa"**;  outputStreamWriter.write(str);  outputStreamWriter.close();  } } |

# 9、递归

## 1、什么是递归？

* 函数的自身调用自身

## 2、递归的使用前提

* 必须要有条件的前提下使用

## 3、案例

1.列出一个文件夹的子孙文件与目录。

|  |
| --- |
| **public static void** test1(File dir) {  File[] files = dir.listFiles();//获取所有文件及文件夹  **for** (File file : files) {  // 如果是文件，直接输出文件名  **if** (file.isFile()) {  System.***out***.println(file.getName());  } **else if** (file.isDirectory()) {// 如果是文件夹,输出文件夹名，并调用自身，继续获取下级文件夹中的文件  System.***out***.println(file.getName());  *test1*(file);  }  } } |

2、列出指定目录中所有的子孙文件与子孙目录名，要求名称前面要有相应数量的空格：第一级前面有0个，第二级前面有1个，第三级前面有2个...，以此类推。

|  |
| --- |
| **public static void** test2(File dir, String space) {  File[] files = dir.listFiles();//获取所有文件及文件夹  **for** (File file : files) {  // 如果是文件，直接输出文件名  **if** (file.isFile()) {  System.***out***.println(space + file.getName());  } **else if** (file.isDirectory()) {// 如果是文件夹,输出文件夹名，并调用自身，继续获取下级文件夹中的文件  System.***out***.println(space + file.getName());  *test2*(file, **" "** + space);  }  } } |

3、列出指定目录中所有的子孙文件与子孙目录名，要求要是树状结构

|  |
| --- |
| **public static void** test3(File dir, String space) {  File[] files = dir.listFiles();//获取所有文件及文件夹  **for** (File file : files) {  // 如果是文件，直接输出文件名  **if** (file.isFile()) {  System.***out***.println(space + file.getName());  } **else if** (file.isDirectory()) {// 如果是文件夹,输出文件夹名，并调用自身，继续获取下级文件夹中的文件  System.***out***.println(space + file.getName());  *test3*(file, **"| "** + space);  }  } } |