# 1. File类

在程序运行时，变量中存储的值都是在内存中的，数据是临时的，程序一旦关闭，数据就消失。如果想永久保留数据，获得上次程序结束时的数据，那么就需要用到文件。因此掌握程序操作文件非常重要。

首先学习File类。File类用于描述文件或文件夹，提供了一些对文件或目录的基本操作。

## 1.1 体验File类

下面的程序在File的一个构造中传入了文件的真实路径，获得了一个File对象，此时File对象就是该物理文件的抽象表现形式，可以调用File对象的方法来获得文件信息。注意File对象只是抽象表现形式，只要传递合法的路径，就能得到File对象，不管硬盘上是否真的有该文件或目录。

调用的方法判断了此文件是否存在、是否是目录和是否是文件。

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.File;  **public class** Demo {  **public static void** main(String[] args) {  *// Windows中，各级路径间用反斜杠 \ 分隔  // 由于 \ 是转义字符，因此再加上一个 \ 转义* File file = **new** File(**"d:\\1.txt"**);  *// Linux中路径间用 / 分隔。  // 实际上Java中也支持在Windows上用斜杠 / ，这里也可以写成 d:/1.txt* System.***out***.println(file.exists()); *// 判断文件是否存在* System.***out***.println(file.isFile()); *// 判断是否是文件* System.***out***.println(file.isDirectory()); *// 判断是否是目录   // 由于该文件不存在，因此上述的输出都是false* } } |

## 1.2 File类的构造方法

File类常用构造方法有3个：

（1）public File(String pathname)；根据文件或目录路径创建File对象。

（2）public File(String parent, String child)；根据一个路径和另一个路径得到File对象。效果就是以parent和child的混合路径创建一个对象。所以一般parent是目录，child是该目录下的文件或目录。

（3）public File(File parent, String child)；根据一个父File对象和一个子文件/目录得到File对象。此方法和上一方法类似，只不过参数1是File对象。

需要注意的是，File类并没有无参数的构造。

使用File类需要导包，即java.io.File。在操作文件中用到的所有类基本都要导入java.io包。

## 1.3 路径问题

首先，File类的对象，既可以代表文件也可以代表文件夹，因此传入的路径既能是目录，也能是文件。

在Windows中，各级目录分隔符为“\”，而在Linux中分隔符为“/”。注意字符“\”是主要转义的。为了跨平台方便，更专业的做法是不手动写目录分隔符，而是用系统提供的变量：File.separatorChar代替。该值会根据不同的操作系统得到相应的目录分隔符。例子：

|  |
| --- |
| File file = **new** File(**"d:"** + File.***separatorChar*** + **"1.txt"**); |

绝对路径和相对路径：找到一个文件既能通过绝对路径，也能通过相对路径。

绝对路径就是从根目录开始的路径，Linux系统中根目录就是“/”，比如“/home/zhang/1.txt”就是一个绝对路径。而Windows的绝对路径是从某个盘符开始的，比如“c:\1.txt”就是一个绝对路径。

相对路径是相对于当前目录的路径，即从当前目录开始找到所需文件。如果当前目录是“d:\code”，那么我们在此目录中，直接用“exam\1.txt”就能找到文件“d:\code\exam\1.txt”。在相对路径中，“..”表示上级目录，“.”表示本目录，所以用“.\exam\1.txt”和“exam\1.txt”的效果一致。“..\file1\2.txt”就进入目录“d:\file1\2.txt”。如果是Linux，那就是斜杠/，建议学习一下Linux常用操作。

执行Java程序时，当前目录为执行java命令时所在的目录。在IDE中，Java命令在src目录外执行，因此如果程序中写相对路径比如“1.txt”，那么表示此文件位置在src目录外的项目文件夹中。但是在程序中写“src/1.txt”这样也不好，因为IDE编译时，会把src下源码编译到一个专门的文件夹中，而把其他类型文件复制到该文件夹中，特点是会保持源码和本项目需要的其他文件的位置关系。至于这应该怎么使用，以后讲。

## 1.4 File类中常用方法

### 1.4.1 创建方法

（1）public boolean createNewFile()：创建该文件，若成功返回true。如果已经存在该文件，则不会再创建，并返回false。

注意，要想成功创建文件，此文件的上面各级目录必须已经存在，并不能直接创建好各级目录并创建文件，否则会抛出IOException异常，说明找不到文件（此方法本身就要处理IOException异常）。下面的方法就是用于创建目录的。

（2）public boolean mkdir()：创建文件夹。如果存在这样的文件夹，就不创建了。同样，该方法需要存在父级目录。

（3）public boolean mkdirs()：创建文件夹，如果各级父文件夹不存在，会帮你创建出来。这是终极方法，会创建路径中所有不存在的目录。

要清楚创建的是文件还是文件夹，不要调错了方法和写错了路径。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.File; **import** java.io.IOException;  **public class** Demo {  *// 需要抛出异常* **public static void** main(String[] args) **throws** IOException {  File directory = **new** File(**"d:"** + File.***separatorChar*** + **"mycode"**);  *// 创建目录* directory.mkdirs();  File file = **new** File(directory, **"1.txt"**);  *// 创建目录下的文件* file.createNewFile();  *// 现在应该存在了* System.***out***.println(file.exists());  } } |

### 1.4.2 删除和重命名（剪切）功能

（1）public boolean delete()：删除一个文件，若删除成功返回true。如果要删除一个文件夹，则此文件夹必须为空，否则不能删除返回false。程序中的删除是不经过回收站，直接从硬盘中清除。

（2）public boolean renameTo(File dest)：将该文件重命名成指定的File对象。如果dest与该文件父路径相同，就是重命名；如果他们父路径不同，就是剪切并且重命名。重命名的情况下可操作非空文件夹，若是剪切的情况，不能操作非空文件夹。

### 1.4.3 判断功能

（1）public boolean isDirectory()：判断是否是目录

（2）public boolean isFile()：判断是否是文件

（3）public boolean exists()：判断是否存在

（4）public boolean canRead()：判断是否可读

（5）public boolean canWrite()：判断是否可写

（6）public boolean isHidden()：判断是否隐藏

（7）public boolean isAbsolute()：判断此对象的路径是否为绝对路径。

### 1.4.4 获取功能

（1）public String getAbsolutePath()：获取绝对路径。

（2）public String getPath()：获得此对象抽象路径名的字符串形式。

（3）public String getName()：返回该对象的文件名或目录名，该名称是路径名名称序列中的最后一个名称。

（4）public long length()：返回由此抽象路径名表示的文件的长度（字节数）。如果此路径名表示一个目录，则返回值是不确定的。

（5）public long lastModified()：获取文件最后一次的修改时间，毫秒值。

（6）public String getParent()：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。

（7）public File getParentFile()：返回此抽象路径名父目录的抽象路径名（即File对象）；如果此路径名没有指定父目录，则返回 null。

（8）File类中有一个静态方法：File.listRoots()，方法的作用是列出可用的文件系统根。即在Windows系统上，可列出所有的盘符。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.File; **import** java.util.Arrays;  **public class** Demo {  **public static void** main(String[] args) {  System.***out***.println(Arrays.*toString*(File.*listRoots*()));  *// 输出结果是：[C:\, D:\, E:\, F:\, G:\]，不同的计算机输出不一样* } } |

（8）public String[] list()：获取该目录下的所有文件或者文件夹的名称，以字符串数组返回。

（9）public File[] listFiles()：获取指定目录下的所有文件或者文件夹，以File数组返回。

（10）上面的两个方法有重载，即：

public String[] list(FilenameFilter filter)

public File[] listFiles(FilenameFilter filter)

public File[] listFiles(FileFilter filter)

就是传入一个“文件名称过滤器接口”的实现，来自定义筛选的条件。

比如现在要列出D盘目录下所有的文本文件（扩展名是txt的文件），那么一般的代码是：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.File; **import** java.util.ArrayList;  **public class** Demo {  **public static void** main(String[] args) {  File dir = **new** File(**"d:"**);  *// 先得到目录下所有文件和目录* File[] allFiles = dir.listFiles(); *// 用listFiles返回File数组，是为了调用File的方法* ArrayList<File> resultList = **new** ArrayList<>();  *// 该File集合用于得到所需要的文件* **for**(**int** i = 0; i < allFiles.**length**; i++) {  **if**(allFiles[i].isFile() && allFiles[i].getPath().endsWith(**".txt"**)) {  *// 符合是文件的条件并且是txt扩展名才能加入到集合中* resultList.add(allFiles[i]);  }  }  *// 输出结果，其中都是txt文件* System.***out***.println(resultList);  } } |

如果采用“文件名称过滤器接口”的重载的话，那么直接在接口中定义过滤条件，然后list()方法或者listFiles()方法直接返回满足条件的数组。

在FilenameFilter接口中都定义了accept方法，我们需要实现。传递过来的参数dir和name分别是每个被找到文件的父级目录和文件名。如果accept方法返回true，就接受该文件到数组中，否则不接受。

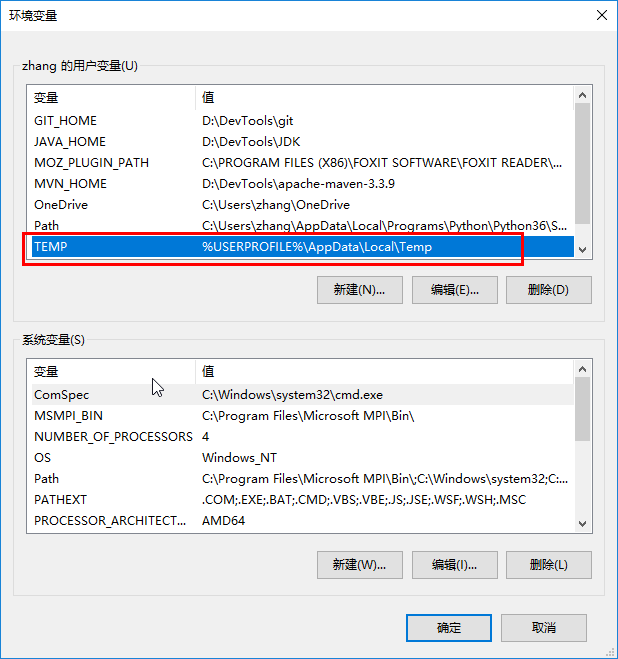
FileFilter接口也类似，只是传递过来的参数只有被找到的文件对象。下面用FileFilter实现和上面代码一样的功能。

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.File; **import** java.io.FileFilter; **import** java.util.Arrays;  **public class** Demo {  **public static void** main(String[] args) {  File dir = **new** File(**"d:"**);  *// 过滤器实现的是FileFilter接口* File[] resultFiles = dir.listFiles(**new** FileFilter() {  @Override  **public boolean** accept(File pathname) {  **if**(pathname.isFile() && pathname.getPath().endsWith(**".txt"**)) {  *// 符合条件就返回true* **return true**;  } **else** {  *// 否则返回false* **return false**;  }  }  });  *// 直接输出得到的结果* System.***out***.println(Arrays.*toString*(resultFiles));  } } |

### 1.4.5 操作临时文件

操作系统有专门用于存放临时文件的文件夹。临时文件在软件安装和使用时应用较多，比如安装软件时把需要的资源放在系统临时文件夹中，然后安装完后再把这些文件删除。

Windows的默认临时文件夹目录在：C:\Users\用户名\AppData\Local\Temp中，其实这是可以在环境变量中配置的，如图：



Java的API中提供了创建临时文件的方法，这是File中的静态方法：

public static File createTempFile(String prefix, String suffix)，参数1用于生成文件名的前缀字符串；必须至少是三字符长；参数2用于生成文件名的后缀字符串；可以为 null，在这种情况下，将使用后缀 ".tmp"，返回的是在临时目录下创建好的File对象。

还能用此函数的另一个重载实现自定义临时目录。更多信息还是参见API文档，是个好东西。

还有一个方法在这里常用到，就是deleteOnExit()，获得临时文件后，应该调用此方法，该方法的作用就是确保Java虚拟机退出时（即程序结束时），删除该临时文件。但是在程序运行过程中，此文件是存在的，可以使用，例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.File; **import** java.io.IOException;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 程序开始，创建临时文件* File tempFile = File.*createTempFile*(**"jas"**, **null**);  *// 可一开始就调用deleteOnExit()方法* tempFile.deleteOnExit();  *// 程序运行过程中可以使用此文件* System.***out***.println(tempFile.exists());  *// 程序结束文件即删除* } } |

## 1.5 递归查找目录下所有文件

目录结构是树状结构，在一个目录打开Windows控制台，使用tree命令就能看到目录的树状结构。

如果想要找到某个目录下所有层次的目录和文件，那么就需要使用递归。这样才能找到所有文件/目录。Java中使用递归需要注意递归次数不能过多，否则可能内存溢出。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.File;  **public class** Demo {  **public static void** main(String[] args) {  *// 显示D盘所有文件和目录  showDirectory*(**new** File(**"d:\\"**));  }   */\*\*  \* 此方法递归显示某目录下所有的文件和目录  \** ***@param dir*** *文件和目录的集合  \*/* **public static void** showDirectory(File dir) {  File[] files = dir.listFiles();  *// 一定要进行为空判断。因为对文件调用listFiles返回null  // 而null.length调用会出现空指针异常* **if**(files != **null**) {  *// 遍历* **for**(**int** i = 0; i < files.**length**; i++) {  **if**(files[i].isDirectory()) {  *// 如果是目录，递归查找  showDirectory*(files[i]);  } **else** {  System.***out***.println(files[i].getAbsolutePath());  }  }  }  } } |

用递归的方法能删除非空文件夹、剪切非空文件夹等。

# 2. 输入输出（IO）流

## 2.1 输入输出流简介

File类只是对文件整体进行操作，如果用程序操作文件的内容，那么就需要用到输入输出流（Input and Output Stream，即IO流 ）技术。

按照流的方向划分，流分为输入流和输出流，这是从程序的角度定义的。将数据从文件读取到程序中，数据流的方向是“输入”到程序中，使用“输入流”；反之，将程序中的数据写入到文件中，数据流的方向是“输出”到文件中，使用“输出流”。

“流”屏蔽了实际的I/O设备处理数据的细节。IO流用来处理设备之间的数据传输。设备包括硬盘、内存、键盘录入、网络等。Java用于操作流的对象都在IO包中。所以使用IO流还是需要导入java.io包中的类。

## 2.2 字节流

字节流可用于处理任何类型的文件，计算机中存储的是二进制数据，字节流处理的单元是一个字节，所以都可用字节处理的方式。用字节流对文件进行读写操作，需要用到以下类：

字节输出流：java.io.OutputStream

字节输入流：java.io.InputStream

从API文档中看出，上述类是抽象类，他们是所有字节流的抽象基类。

### 2.2.1 字节输出流FileOutputStream

OutputStream是抽象类，为了在程序中写文件，可以用它的子类FileOutputStream。

FileOutputStream的常用构造方法有：

（1）public FileOutputStream(File file)：根据File对象创建文件字节输出流

（2）public FileOutputStream(File file, boolean append)：根据File对象创建文件字节输出流。如果append参数为true，表示向该文件中追加数据，如果为false，就表示新创建文件。

（3）public FileOutputStream(String name)：根据文件名创建文件字节输出流

（4）public FileOutputStream(String name, boolean append)：根据文件名创建文件字节输出流，append还是表示是否追加文件。

实际上即便传入的是String文件名，内部还是转换成File对象。所有的一个参数的表示是“不追加”，即和传入false一样，会新创建文件（如果存在该文件，会覆盖文件内容）。

创建好对象后，通过write()方法向文件中写内容，write()有以下重载：

（1）public void write(int b)：写一个字节，写入的是ASCII对应的字符

（2）public void write(byte[] b)：写一个字节数组

（3）public void write(byte[] b,int off,int len)：写一个字节数组的一部分

最后，流使用完毕后，必须关闭流对象，即调用close()方法释放流资源。close()之后，该流就不能再使用了，如果不在程序中关闭流对象，那么系统的资源很容易被大量耗费，影响性能。

上述就是流操作的一般步骤，总结如下：

（1）创建流对象；

（2）使用流操作文件内容；

（3）关闭流对象。在IO中有很多异常，一般在finally中关闭流对象。这里为了方便就不这样处理了。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.\*;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 创建字节输出流* FileOutputStream fis = **new** FileOutputStream(**"d:"** + File.***separatorChar*** + **"1.txt"**);  *// 写两个字符到文件中* fis.write(**'1'**);  fis.write(**'2'**);  *// 最后关闭流* fis.close();  } } |

上述代码每次运行都会创建d:\1.txt文件，内容都是12，如果想每次都是追加文件，那么就添加true参数。即new FileOutputStream(文件, true)。

如果写入数据较多，可用字节数组的方式提高效率：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.File; **import** java.io.FileOutputStream; **import** java.io.IOException;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 这次是追加* FileOutputStream fos = **new** FileOutputStream(**new** File(**"d:\\1.txt"**), **true**);  *// 使用字符串的getBytes()方法将字符串转化成字节数组* fos.write(**"这是新内容\n新一行"**.getBytes());  fos.close();  } } |

换行问题：如果在文件中写入了换行符“\n”，Windows记事本中不会显示换行，因为Windows系统认为“\r\n” 才是文件中的换行。如果用其他操作系统，或者用其他高级的文本编辑器，就不会有这样的问题，会正确显示换行。

### 2.2.2 字节输入流FileInputStream

InputStream是抽象类，为了在程序中读文件，可以用它的子类FileInputStream。

FileInputStream的常用构造方法有：

（1）public FileInputStream(File file)：通过File对象创建文件输入流。

（2）public FileInputStream(String name)：通过文件名创建文件输入流。

创建好输入流后，可使用read方法读数据，有重载：

（1）int read()：一次读取一个字节，返回读到的对应的整型数据。若没有数据返回-1

（2）int read(byte[] b)：一次读取一个字节数组，将数据存在这个字节数组中，返回值是实际读取的字节个数，若没有数据返回-1。

read()方法每执行一次，读文件的指针就向后移动一个字节，因此需要循环读。使用完毕后同样需要关闭流。

例子1：用read()方法每次读取一个字节。

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.FileInputStream; **import** java.io.IOException;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  FileInputStream fis = **new** FileInputStream(**"d:\\1.txt"**);  **int** res = 0; *// 用于记录每次读到的结果* **while**( ( res = fis.read() ) != -1 ) {  *// 输出* System.***out***.print((**char**)res);  }  fis.close();  } } |

上述的输出，很可能是乱码，只有英文能正常显示。如果文件不是纯文本文件或者文件中有中文，那么就会乱码。因为非文本文件有自己的格式，转换成字符输出不能识别。而对于中文等字符，编码方式不同，很多情况下是用两个字节表示一个汉字的，因此这样一个字节一个字节的输出会乱码，后面会学到如何处理（用字符流）。

不过，使用字节流整体操作文件是不受影响的，比如加密解密(以前有个取反加密的例子)、复制文件等。因为这些操作不关心数据的输出显示，只关心二进制形式能否正确传递或者转换和还原（对于加密/解密来说）。

上述用read()方法一次读取一个字节的效率太低了，若是大文件，循环次数非常多。可以用read()的第二个重载，一次读取一个字节数组，把当前读到的数据暂存在数据中。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.FileInputStream; **import** java.io.IOException;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  FileInputStream fis = **new** FileInputStream(**"d:\\1.txt"**);  *// 定义该buffer字节数组用于暂存数据，一般设置成1024的整数倍* **byte**[] buffer = **new byte**[1024];  **int** length = 0; *// 用于记录本次读取的字节数。用于判断是否需要再读以及判断buffer数组中的有效数据长度* **while**( (length = fis.read(buffer)) != -1 ) {  *// 获得读到的数据* String res = **new** String(buffer, 0, length);  System.***out***.println(res);  }  *// 关闭流* fis.close();  } } |

这里用new String(buffer, 0, length)获取每次数组中的字符串结果，就是把数组中0-length范围的byte元素转成字符串。这是因为读取的时候，不是每次读到的数据都能把buffer数组装满，在最后读取到数据时，基本上读到的数据只能装满buffer数组的一部分，因此需要这样操作。

发现上述程序输出中文时不乱码了，这是因为一次把整个数组数据都转成字符串，而不是一个字节一个字节转化的。前面讲过，大部分以两个字节表示一个汉字，所以这里很少乱码。当然，如果处理图片等文件，还是必定乱码（实际也不能叫乱码，就是无法看懂的玩意），因为这些文件本来就不是文本格式。

练习：学习了上面的字节输入输出流，就能实现文件的复制了。即通过输入流读取文件内容，再通过输出流将内容写入新文件中，这样就完成复制。

代码示例：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.File; **import** java.io.FileInputStream; **import** java.io.FileOutputStream; **import** java.io.IOException;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  File file1 = **new** File(**"d:\\1.txt"**);  File file2 = **new** File(**"d:\\2.txt"**);  *copyFile*(file1, file2); *// 调用复制方法* }   **public static void** copyFile(File src, File target) **throws** IOException {  *// 创建输入输出流* FileInputStream fis = **new** FileInputStream(src);  FileOutputStream fos = **new** FileOutputStream(target);  *// 记录读取的数据和长度的变量* **byte**[] buffer = **new byte**[1024 \* 1024 \* 1]; *// 每次1M* **int** length = 0;  **while**((length = fis.read(buffer)) != -1) {  fos.write(buffer, 0, length); *// 直接用fos的write重载* }  *// 关闭两个流对象。* fis.close();  fos.close();  } } |

示例中使用了缓冲数组的方式读取，这样处理大文件时效率高。读者可以试一下用read每次只读一个字节，然后复制一个十几M的大文件，会明显感觉采用缓冲数组的方式快。可以在调用copyFile方法前后获取下系统时间（毫秒值），以此计算复制方法的耗时。

### 2.2.3 字节缓冲流（高效流）

上述用字节流时发现使用缓冲数据能大大提高效率。实际JDK本身也提供了带缓冲区的IO流。其中，上面的文件字节流对应的缓冲流是：

BufferedOutputStream：缓冲输出流；

BufferedInputStream：缓冲输入流。

通过查看API文档，知道实际上缓冲流就是默认在内部维护了一个8KB的字节数组。使用缓冲流操作大文件能看到很大的效率提升。

查看缓冲流的构造方法，发现有两个重载。一个重载是直接将字节流传递进去，这是通过字节流来创建高效流。另一个重载还要传递一个int类型数据，这就是指定缓冲流内部维护的字节数组大小。

字节缓冲流的使用方法和字节流的使用方法是一致的，没什么变化。对于字节缓冲输入流，既能采用read()每次读取一个字节，也能每次读取一个数组，都是可以的。即便每次读取一个字节，效率也比使用字节流读取一个字节的效率高很多。如果在缓冲流中每次读取一个数组，那么效率将会更高，推荐使用。

使用高效流改进文件复制方法：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.\*;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  File file1 = **new** File(**"d:\\1.txt"**);  File file2 = **new** File(**"d:\\2.txt"**);  *copyFile*(file1, file2);  }  **public static void** copyFile(File src, File target) **throws** IOException {  BufferedOutputStream bos = **new** BufferedOutputStream(**new** FileOutputStream(src));  BufferedInputStream bis = **new** BufferedInputStream(**new** FileInputStream(target));  *// 缓冲数组和长度记录* **byte**[] buffer = **new byte**[1024];  **int** length = 0;  **while**((length = bis.read(buffer)) != -1) {  bos.write(buffer, 0, length);  }  bis.close();  bos.close();  } } |

## 2.3 字符流

用字节流处理文本文档时，多有不便。因为文本涉及到文字的编码问题。字符流是建立在字节流之上的，它提供字符层次的编码和解码。例如，在写入一个字符时，JVM会将字符转为文件指定的编码存储；在读取字符时，再将用该编码将二进制数据转化为字符。

简单说，字符流=字节流+编码表。编码表就是由字符及其对应的数值组成的一张表。常见的编码表有：

（1）ASCII：美国标准信息交换码。用一个字节的7位就可表示。

（2）ISO8859-1：拉丁码表。欧洲码表。用一个字节的8位表示。又称Latin-1或“西欧语言”。它以ASCII为基础，在1个字节中ASCII没有使用的范围内，加入192个字母及符号。只是比ASCII更全面。

（3）GB2312：中文编码表。英文占一个字节，中文占两个字节。

（4）GBK：中文编码表升级，融合了更多的中文文字符号。

（5）GB18030：GBK的取代版本。

（6）BIG-5：通行于台湾、香港地区的一个繁体字编码方案，俗称“大五码”。

（7）Unicode：国际标准码，融合了多种文字。所有文字都用两个字节来表示，Java语言使用的就是Unicode

（8）UTF-8：最多用三个字节来表示一个字符。

以后我们接触较多的就是ISO8859-1、GBK、GB2312和UTF-8。不同编码所映射的字符是不一样的，比如要存储中文数据，就不能使用ISO8859-1字符集来存储，因为该字符集中无中文。同样，如果想要文件不产生乱码，读写文件时必须用一致的编码，即文件的编码与解码要一致，否则乱码。

### 2.3.1 字符串编码与解码方法

（1）用指定字符集编码字符串为字节数组：byte[] getBytes(String charsetName)，例如：" hello world".getBytes("UTF-8")。

（2）用指定字符集将字节数组解码为字符串：public String(byte[] bytes, String charsetName)

以前，使用这两个方法时都没有指定字符集，采用本机默认的字符集进行编码和解码（Windows中文操作系统默认字符集是GBK）。

### 2.3.2 字符流Reader和Writer

在字节流中我们遇到了中文乱码，体验到用字节流的不便。现在用字符流来处理文本。Java中提供的字符流是：

Reader：字符输入流，用于读取文件；

Writer：字符输出流，用于写入文件。

Reader和Writer是抽象类，我们需要使用输入字符流InputStreamReader和输出字符流OutputStreamWriter。这两个类的构造方法相似，都是用字节输出/字节输出流来创建字符流对象，并且有重载可以指定流的编码。即：

public OutputStreamWriter(OutputStream out)：根据输出字节流创建输出字符流，使用系统默认编码；

public OutputStreamWriter(OutputStream out, String charsetName)：根据输出字节流创建输出字符流，使用指定编码；

public InputStreamReader(InputStream is)：根据输入字节流创建输入字符流，使用系统默认编码；

public InputStreamReader(InputStream is, String charsetName)：根据输入字节流创建输入字符流，使用指定编码。

创建好字符流后，就能调用其中的方法读写数据了。

对于输出字符流，用write方法写数据：

（1）public void write(int c)：写一个字符(写入该数字对应的字符)

（2）public void write(char[] cbuf)：写一个字符数组

（3）public void write(char[] cbuf,int off,int len)：写一个字符数组的一部分

（4）public void write(String str)：写一个字符串，常用

（5）public void write(String str,int off,int len)：写一个字符串的一部分

注意有时使用flush()方法刷新缓冲区，特别是大文件时要隔一段时间刷新一遍使数据完全写入到文件中。记得调用close()方法释放资源，因为close()之前会自动调用flush().

对于输出字符流，用read方法读数据：

（1）int read()：读取单个字符。读取的字符，如果已到达流的末尾，则返回 -1

（2）int read(char[] chs)：将字符读入数组。读取的字符，如果已到达流的末尾，则返回 -1。这里就是存入到字符数组中了，而不是字节数组。

最后，完成读写操作后，别忘了关闭流资源。

这里以读文件为例写代码，使用了缓冲数组，使用的编码为UTF-8。

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.File; **import** java.io.FileInputStream; **import** java.io.IOException; **import** java.io.InputStreamReader;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 创建输入字符流，使用UTF-8编码* InputStreamReader isr = **new** InputStreamReader(**new** FileInputStream(**new** File(**"d:\\1.txt"**)), **"UTF-8"**);  *// 缓冲数组和长度。* **char**[] buffer = **new char**[256];  **int** length = 0;  **while**((length = isr.read(buffer)) != -1) {  *// 将字符数组部分转换成字符串输出* System.***out***.println(**new** String(buffer, 0, length));  }  isr.close();  } } |

此外，API提供了FileWriter和FileReader专门处理本地默认编码的文件。使用该流可直接传递File对象来创建字符流，只是这样只能使用默认编码，不灵活。其他使用的方法都是一样的。可参见API。

根据字符流原理，一般就不用字符流拷贝文件，因为其他文件的编码和字符编码不同。

### 2.3.3 字符缓冲流BufferedWriter和BufferedReader

为了提高效率，字符流也提供了字符缓冲流。字符流对应的缓冲字符流是BufferedWriter和BufferedReader。其内部是维护了一个数组。

字符缓冲流的构造方法可接受Reader或者Writer对象，因此我们可以传递Reader或Writer的子类，即InputStreamReader或OutputStreamWriter，这样，字符缓冲流也能指定处理的编码了，是通过字符流的编码传进去的。当然也能传递FileReader或FileWriter，只不过只能处理默认编码。

另外，字符缓冲流还增强了字符流的功能，表现在：

（1）BufferedWriter类有方法：newLine()，直接向文件写入一个换行符。

（2）BufferedReader类有方法：readLine()，直接从文件中读取一行，并将指针移到下一行，返回String字符串，如果已到达流末尾，则返回null。

这样读写文件有时就更加方便。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.BufferedReader; **import** java.io.FileInputStream; **import** java.io.IOException; **import** java.io.InputStreamReader;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 创建高效流* BufferedReader br = **new** BufferedReader(**new** InputStreamReader(**new** FileInputStream(**"d:\\1.txt"**), **"UTF-8"**));  *// 直接一行一行的读* String lineStr = **null**;  **while**((lineStr = br.readLine()) != **null**) {  System.***out***.println(lineStr);  }  br.close();  } } |

上述直接一行一行的输出，方便。

## 2.4 IO流中的异常处理

把可能出现的异常代码包含在try中，然后catch中输出异常信息并抛出运行时异常停止程序（按照实际进行处理）。最后在finally中关闭流。关闭前需要进行为空判断。而且close()方法还有异常需要处理。如果有两个流需要关闭，就不能直接写在第一个finally中，因为如果第一个流关闭有异常的话，第二个流就不会关闭了。因此要把第二个流写在第一个流关闭异常处理的finally代码块中。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.File; **import** java.io.FileInputStream; **import** java.io.FileOutputStream; **import** java.io.IOException;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  File file1 = **new** File(**"d:\\1.txt"**);  File file2 = **new** File(**"d:\\2.txt"**);  *copyFile*(file1, file2); *// 调用复制方法* }   **public static void** copyFile(File src, File target) {  *// 创建输入输出流* FileInputStream fis = **null**;  FileOutputStream fos = **null**;  **try** {  fis = **new** FileInputStream(src);  fos = **new** FileOutputStream(target);  *// 记录读取的数据和长度的变量* **byte**[] buffer = **new byte**[1024 \* 1024 \* 1]; *// 每次1M* **int** length = 0;  **while** ((length = fis.read(buffer)) != -1) {  fos.write(buffer, 0, length); *// 直接用fos的write重载* }  } **catch** (IOException e) {  e.printStackTrace();  **throw new** RuntimeException();  } **finally** {  **if**(fos != **null**) {  **try** {  fos.close();  } **catch** (IOException e) {  e.printStackTrace();  **throw new** RuntimeException();  } **finally** {  **if**(fis != **null**) {  **try** {  fis.close();  } **catch** (IOException e) {  e.printStackTrace();  **throw new** RuntimeException();  }  }  }  }  }  } } |