1. File类

1.1. File类说明

存储在变量,数组和对象中的数据是暂时的,当程序终止时他们就会丢失.为了能够永久的保存程序中创建的数据,需要将他们存储到硬盘或光盘的文件中.这些文件可以移动,传送,亦可以被其他程序使用.由于数据存储在文件中,所以我们需要学习一个和文件有密切关系的类,叫做File类,将要掌握获取文件的属性以及删除和重命名文件.最终如何向文件中写入数据和从文件中读取数据.

那么 File 类关心的是在磁盘上文件的存储.

File 类描述的是一个文件或文件夹。(文件夹也可以称为目录)

该类的出现是对文件系统的中的文件以及文件夹进行对象的封装。可以通过对象的思想来操作文件以及文件夹。

可以用面向对象的处理问题，通过该对象的方法，可以得到文件或文件夹的信息方便了对文件与文件夹的属性信息进行操作。

文件包含很多的信息:如文件名、创建修改时间、大小、可读可写属性等。

1.2. 体验 File 类

检验指定路径下是否存在指定的目录或者文件.

```
// 检验指定路径下是否存在指定的目录或者文件.
        File file = new File("c:\\a.txt");
        System.out.println(file.exists());
// File对象是否是目录
        System.out.println(file.isDirectory());
// 对象是否是文件
        System.out.println(file.isFile());
```

结论：File 对象也可以表示不存在的文件。其实代表了一个抽象路径

构建一个 File 类的实例并不会在机器上创建一个文件.不管文件是否存在都可以创建任意文件名的 File 实例,可以调用 File 实例的 exists 方法判断文件或目录是否存在

1.3. 构造一个 File 类实例:

```
new File(String pathname);
        通过将给定路径来创建一个新 File 实例。
new File(String parent, String child);
```

根据 parent 路径名字符串和 child 路径名创建一个新 File 实例。
parent 是指上级目录的路径，完整的路径为 parent+child.
new File(File parent, String child);
根据 parent 抽象路径名和 child 路径名创建一个新 File 实例。
parent 是指上级目录的路径，完整的路径为 parent.getPath()+child.
说明：
如果指定的路径不存在（没有这个文件或是文件夹），不会抛异常，这时 file.exists()返回 false。

新建 File 对象 File file=new File();

```
public static void main(String[] args) {
        File file = new File();
    }
```

1:创建 File 对象需要导包， import java.io.File
2:File 对象没有无参数构造.创建对象需要传参.
根据 API 文档提示,传入一个文件的字符串路径. String path="c:/a.txt";
(a.txt 文件在 c 盘下已经存在)

```
        //file 是一个文件对象
        String path = "c:/a.txt";
        File file = new File(path);
```

File 类的对象，既可以代表文件也可以代表文件夹。

```
public static void main(String[] args) {
        //file 是一个文件夹
        String path = "c:/test";
        File file = new File(path);
    }
```

1.4. 路径：

路径就是文件或文件夹所在的位置。

1.4.1. 路径分割符：

上下级文件夹之间使用分隔符分开：
在 Windows 中分隔符为'\'，在 Unix/Linux 中分隔符为'/'。
跨平台的目录分隔符

更专业的做法是使用 File.separatorChar,这个值就会根据系统得到的相应的分割符。

例：new File("c:" + File.separatorChar + "a.txt");

注意，如果是使用"\"，则需要进行转义，写为"\\"才可以，如果是两个"\"，则写为"\\\\"。

实验：

在以下代码的 path 处写不同的路径试一下，并观察输出结果。

```
File file = new File(path);
System.out.println(file.getAbsolutePath());
```

1.4.2. 绝对路径与相对路径：

对于 UNIX 平台，绝对路径名的前缀是"/"。相对路径名没有前缀。

对于 Windows 平台，绝对路径名的前缀由驱动器号和一个":"组成，例"c:\\..."。相对路径没有盘符前缀。

相对路径：

相对路径是指相对于某位置的路径，是指相对于当前目录。

在执行 Java 程序时，相对路径为执行 java 命令时当前所在的目录。

实验：

在不同的路径下执行 java 命令运行以下程序，并观察输出结果。

```
File file = new File("a.txt");
System.out.println(file.getAbsolutePath());
```

一般在使用时，建议用绝对路径，因为相对路径容易出问题，不好确定到底在什么地方。

```
//相对路径
//File file = new File("src/a.txt");
```

1.5. File 类中常用的方法：

创建：

createNewFile() 在指定位置创建一个空文件，成功就返回 true，如果已存在就不创建然后返回 false

mkdir() 在指定位置创建目录，这只会创建最后一级目录，如果上级目录不存在就抛异常。

mkdirs() 在指定位置创建目录，这会创建路径中所有不存在的目录。

renameTo(File dest) 重命名文件或文件夹，也可以操作非空的文件夹，文件

不同时相当于文件的剪切,剪切时候不能操作非空的文件夹。移动/重命名成功则返回 true，失败则返回 false。

删除：

delete()　　删除文件或一个空文件夹，如果是文件夹且不为空，则不能删除，成功返回 true，失败返回 false。

deleteOnExit()　在虚拟机终止时，请求删除此抽象路径名表示的文件或目录，保证程序异常时创建的临时文件也可以被删除

判断：

exists()　　文件或文件夹是否存在。
isFile()　　是否是一个文件，如果不存在，则始终为 false。
isDirectory()　是否是一个目录，如果不存在，则始终为 false。
isHidden()　　是否是一个隐藏的文件或是否是隐藏的目录。
isAbsolute()　测试此抽象路径名是否为绝对路径名。

获取：

getName()　　获取文件或文件夹的名称，不包含上级路径。
getPath()　　返回绝对路径，可以是相对路径，但是目录要指定
getAbsolutePath()　获取文件的绝对路径，与文件是否存在没关系
length()　　获取文件的大小（字节数），如果文件不存在则返回 0L，如果是文件夹也返回 0L。
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。
lastModified()　获取最后一次被修改的时间。

文件夹相关：

staic File[] listRoots()　列出所有的根目录（Window 中就是所有系统的盘符）
list()　　返回目录下的文件或者目录名，包含隐藏文件。对于文件这样操作会返回 null。
list(FilenameFilter filter)　返回指定当前目录中符合过滤条件的子文件或子目录。对于文件这样操作会返回 null。
listFiles()　　返回目录下的文件或者目录对象（File 类实例），包含隐藏文件。对于文件这样操作会返回 null。
listFiles(FilenameFilter filter)　返回指定当前目录中符合过滤条件的子文件或子目录。对于文件这样操作会返回 null。

1.6. 案例：

1，列出指定目录中所有的子文件名与所有的子目录名。

2，列出指定目录中所有的子文件名与所有的子目录名，要求目录名与文件名分开列出，格式如下：

子目录：

...

...

子文件：

...

...

3，列出指定目录中所有扩展名为.java 的文件。

4，列出指定目录中所有扩展名为.class 的文件。

5，思考第 3 与第 4 题，代码是不是重复呢，如果想要列出其中的所有.txt 文件，是不是要再写一个类呢？

a，请自行设计一个工具方法，可以传递指定的扩展名，工具方法会过列出指定目录中指定扩展名的所有子文件与子文件夹。

b，请利用 FilenameFilter 接口写一个工具类，可以传递指定的过滤规则。

6，列出指定目录中所有的子孙文件与子孙目录名，只需要列出名称即可。

解题：列出指定目录中所有的子文件名与所有的子目录名。

需求 1：获取所有的 c:/test 即 test 目录下的所有文件和文件夹
解题思路：
代码需要封装,就需要创建方法,并在 main 方法中调用和测试. 方法名要有意义：listAllFilesAndDirs
第一步：创建 File 对象
第二步：查找 File 类中可用的方法，想要获取该目录下的所有子文件和子目录
第三步：显示这些文件和文件夹的名称

实现：

```
/**
     * 列出指定目录中所有包含的子文件与子目录
     */
    public static void listAllFilesAndDirs(String path) {
        // 1.创建File对象，表示这个目录
        File dir = new File(path);
        // 2.通过list方法得到所包含的所有子目录与子文件名称
        String[] names = dir.list();
        // 3显示这些名称
        for (int i = 0; i < names.length; i++) {
            System.out.println(names[i]);
        }
    }
```

解题：列出指定目录中所有的子文件名与所有的子目录名，要求目录名与文件名分开列出
案例 1 把文件和文件夹都列了出来，但是无法区分文件和文件夹.File 类有判断文件和文件夹的方法，但是 list 方法返回的是 String 数组，这个 list() 方法无法满足我们的需求.继续查找 File 的方法.查看 api 找到 File[] listFiles() 发现该方法返回的是一个 File 数组。

思路：
第一步:创建 listAllFilesAndDirs(String path)方法,接受路径
第二步：创建 File 对象表示这个目录
第三步：通过 listFiles 方法得到所包含的所有子目录与子文件名称
第四步：得到所有的文件名集合，与所有的文件夹名集合
第五步：分别显示文件名与文件夹名

实现

```
public static void listAllFilesAndDirs2(String path) {
        // 1.创建File对象，表示这个目录
        File dir = new File(path);
        // 2通过listFiles方法得到所包含的所有子目录与子文件名称
        File[] names = dir.listFiles();
        // 3,分别显示文件名与文件夹名
        for (int i = 0; i < names.length; i++) {
            File file = names[i];
            if (file.isFile()) {
                System.out.println(("子文件: "));
                System.out.println("\t" + file.getName());
            } else if (file.isDirectory()) {
                System.out.println(("子目录: "));
                System.out.println("\t" + file.getName());
            }
        }
    }
```

实现二：

```
public static void listAllFilesAndDirs(String path) {
        //1创建File对象表示这个目录
        File dir = new File(path);

        //2通过listFiles方法得到所包含的所有子目录与子文件名称
        File[] names = dir.listFiles();

        //3,得到所有的文件名集合，与所有的文件夹名集合
        List<File> filesList = new ArrayList<File>();
        List<File> dirsList = new ArrayList<File>();
        for (int i = 0; i < names.length; i++) {
            File file = names[i];
            if (file.isFile()) {
                filesList.add(file);
```

```
            } else if (file.isDirectory()) {
                dirsList.add(file);
            }
        }

        //4，分别显示文件名与文件夹名
        System.out.println("子文件： ");
        for (int i = 0; i < filesList.size(); i++) {
            System.out.println("\t" + filesList.get(i).getName());
        }
        System.out.println("子目录： ");
        for (int i = 0; i < dirsList.size(); i++) {
            System.out.println("\t" + dirsList.get(i).getName());
        }
    }
```

练习 3，列出指定目录中所有扩展名为.java 的文件。

需求：从指定目录中找到指定扩展名的文件，并列出来
思路
第一步：创建 lsitAllFiles 方法,接受路径和文件后缀名
第二步：获取所有的子文件和子文件夹
第三步：从中找出符合条件的文件并显示出来

注意：不同系统对于路径的 windows 系统使用斜线作为路径分隔符 "\" linux 系统使用反斜线作为路径分隔符"/"java 是跨平台的语言，java 程序如果部署到 linux 系统上，如果程序中有 File 对象， 可以使用 File 类 *separatorChar* （字段）

```
public class FileTest2 {
    public static void main(String[] args) {
        String path = "c:" + File.separatorChar + "test";
        File file = new File(path);
        listtAllFiles(file, "java");
    }

    /**
     * 从指定目录中找到指定扩展名的文件，并列出来
     *
     */
    public static void listtAllFiles(File dir, String extension) {
        // 1.获取所有的子文件和子文件夹
        File[] files = dir.listFiles();

        // 2.从中找出符合条件的文件并显示出来
        for (int i = 0; i < files.length; i++) {
```

```
            File file = files[i];
            // 3.需要以指定文件后缀结尾才算符合条件
            if (file.getName().endsWith(extension)) {
                System.out.println(file.getName());
            }
        }
    }
}
```

练习 4：

```
public class FileTest2 {
    public static void main(String[] args) {
        String path = "c:" + File.separatorChar + "test";
        File file = new File(path);
        listtAllFiles2(file, "txt");
    }


    /**
     * FilenameFilter接口写一个工具类，可以传递指定的过滤规则。
     *  从指定目录中找到指定扩展名的文件，并列出来
     *
     * */
    public static void listtAllFiles2(File dir, String name) {
        // 1.获取所有的子文件和子文件夹
        String[] files = dir.list(new DirFilter("txt"));

        // 2显示名称
        for (int i = 0; i < files.length; i++) {
            System.out.println(files[i]);
        }
    }
}

class DirFilter implements FilenameFilter {
    private String extension;

    public DirFilter() {

    }

    public DirFilter(String extension) {
        this.extension = extension;
    }
```

```
    @Override
    public boolean accept(File dir, String name) {
        return name.endsWith(extension);
    }
}
```

注意：DirFilter 就是实现了 accept 方法.提供给 File 类的 list 方法使用.

2. IO 流体验与简介

File 对象可以表示存在的文件或文件夹，也可以表示不存在的。
我们想要得到文件的内容怎么办，File 只是操作文件，文件的内容如何处理就需要使用 io 流技术了。
例如在 C 盘下有一个名称为 a.txt 的文本文件.想要通过 Java 程序读出来文件中的内容，需要使用 IO 流技术.同样想要将程序中的数据,保存到硬盘的文件中,也需要 IO 流技术.
读和写文件文件示例：

```
public class IoTest {
    public static void main(String[] args) throws FileNotFoundException,
            IOException {
        writFileTest();

        readFileTest();
    }

    private static void writFileTest() throws FileNotFoundException,
            IOException {
        // 创建文件对象
        File file = new File("c:\\a.txt");
        // 创建文件输出流
        FileOutputStream fos = new FileOutputStream(file);
        fos.write('g');
        fos.write('z');
        fos.write('i');
        fos.write('t');
        fos.write('c');
        fos.write('a');
        fos.write('s');
        fos.write('t');
        fos.close();
    }
```

```
    private static void readFileTest() throws FileNotFoundException,
            IOException {
        // 创建文件对象
        File file = new File("c:\\a.txt");
        // 创建文件输入流
        FileInputStream fis = new FileInputStream(file);
        // 有对多长，就读多少字节。
        for (int i = 0; i < file.length(); i++) {
            System.out.print((char) fis.read());
        }
        fis.close();
    }
}
```

当完成流的读写时,应该通过调用close方法来关闭它,这个方法会释放掉十分有限的操作系统资源.如果一个应用程序打开了过多的流而没有关闭它们,那么系统资源将被耗尽.

IO流简介:(Input/Output)

I/O类库中使用"流"这个抽象概念。Java对设备中数据的操作是通过流的方式。

表示任何有能力产出数据的数据源对象，或者是有能力接受数据的接收端对象。"流"屏蔽了实际的I/O设备中处理数据的细节。IO流用来处理设备之间的数据传输。设备是指硬盘、内存、键盘录入、网络等。

Java用于操作流的对象都在IO包中。IO流技术主要用来处理设备之间的数据传输。

由于Java用于操作流的对象都在IO包中。所以使用IO流需要导包如：import java.io.*;

IO流的分类

流按操作数据类型的不同分为两种：字节流与字符流。

流按流向分为：输入流，输出流（以程序为参照物，输入到程序，或是从程序输出）

3. 字节流

什么是字节流

计算机中都是二进制数据,一个字节是8个2进制位.字节可以表示所有的数据,比如文本,音频,视频.图片,都是作为字节存在的.也就是说字节流处理的数据非常多。

在文本文件中存储的数据是以我们能读懂的方式表示的。而在二进制文件中存储的数据是用二进制形式表示的。我们是读不懂二进制文件的，因为二进制文件是为了让程序来读取而设计的。例如，Java的源程序(.java源文件)存储在文本文件中,可以使用文本编辑器阅读,但是Java的类(字节码文件)存储在二进制文件中,可以被Java虚拟机阅读。二进制文件的优势在于它的处理效率比文本文件高。

我们已经知道File对象封装的是文件或者路径属性，但是不包含向(从)文件读(写)

数据的方法。为了实现对文件的读和写操作需要学会正确的使用 Java 的 IO 创建对象。
字节流的抽象基类：
输入流：java.io.InputStream
输出流：java.io.OutputStream
特点：
字节流的抽象基类派生出来的子类名称都是以其父类名作为子类名的后缀。
如：FileInputStream， ByteArrayInputStream 等。
说明：
字节流处理的单元是一个字节，用于操作二进制文件（计算机中所有文件都是二进制文件）

3.1. InputStream

案例：读取"c:/a.txt"文件中的所有内容并在控制台显示出来。
注意：事先准备一个 a.txt 并放到 c:/下，不要保存中文。
a，使用 read()方法实现。
b，使用 int read(byte[] b)方法实现。
写代码读取"c:/a.txt"文件中的所有的内容并在控制台显示出来
实现：
查看 api 文档(自己一定要动手)
InputStream 有 read 方法，一次读取一个字节，OutputStream 的 write 方法一次写一个 int。发现这两个类都是抽象类。意味着不能创建对象，那么需要找到具体的子类来使用。
通过查看 api 文档，找到了 FileInputStream 类，该类正是我们体验 Io 流的一个输入流。

实现；显示指定文件内容。
明确使用流，使用哪一类流？使用输入流，FileInputStream
第一步：
1：打开流（即创建流）
第二步：
2：通过流读取内容
第三步：
3：用完后，关闭流资源

显然流是 Java 中的一类对象，要打开流其实就是创建具体流的对象，由于是读取硬盘上的文件，应该使用输入流。所以找到了 InputStream 类，但是 InputStream 是抽象类，需要使用它的具体实现类来创建对象就是 FileInputStream。通过 new 调用 FileInputStream 的构造方法来创建对象。发现 FileInputStream 的构造方法需要指定文件的来源。查看构造方法，可以接受字符串也可以接受 File 对象。我们通过构建 File 对象指定文件路径。

使用流就像使用水管一样，要打开就要关闭。所以打开流和关闭流的动作是比不可少的。如何关闭流？使用 close 方法即可，当完成流的读写时，应该通过调用 close 方法来关闭它，这个方法会释放掉十分有限的操作系统资源．如果一个应用程序打开了过多的流而没有关闭它们，那么系统资源将被耗尽．

如何通过流读取内容？

查找 api 文档通过 read 方法，查看该方法，发现有返回值，并且是 int 类型的，该方法一次读取一个字节（byte）

3.1.1. 输入流读取方式 1：

read 方法（）

一次读取一个字节,读到文件末尾返回-1.

仔细查看 api 文档发现 read 方法如果读到文件的末尾会返回-1。那么就可以通过 read 方法的返回值是否是-1 来控制我们的循环读取。

```
/**
     * 根据read方法返回值的特性，如果独到文件的末尾返回-1，如果不为-1就继续向下读。
     * */
private static void showContent(String path) throws IOException {
        // 打开流
        FileInputStream fis = new FileInputStream(path);

        int len = fis.read();
        while (len != -1) {
            System.out.print((char)len);
            len = fis.read();

        }
        // 使用完关闭流
        fis.close();
    }
```

我们习惯这样写：

```
/**
     * 根据read方法返回值的特性，如果独到文件的末尾返回-1，如果不为-1就继续向下读。
     * */
    private static void showContent(String path) throws IOException {
        // 打开流
        FileInputStream fis = new FileInputStream(path);

        int len;
        while ((len = fis.read()) != -1) {
            System.out.print((char) len);
        }
        // 使用完关闭流
        fis.close();
```

```
    }
```

3.1.2. 输入流读取方式 2：

使用 read(byte[] b) 方法。使用缓冲区(关键是缓冲区大小的确定)
使用 read 方法的时候，流需要读一次就处理一次，可以将读到的数据装入到字节数组中，一次性的操作数组，可以提高效率。

问题 1:缓冲区大小

那么字节数组如何定义？定义多大？
可以尝试初始化长度为 5 的 byte 数组。通过 read 方法，往 byte 数组中存内容
那么该 read 方法返回的是往数组中存了多少字节。

```
/**
     * 使用字节数组存储读到的数据
     * */
    private static void showContent2(String path) throws IOException {
        // 打开流
        FileInputStream fis = new FileInputStream(path);

        // 通过流读取内容
        byte[] byt = new byte[5];
        int len = fis.read(byt);
        for (int i = 0; i < byt.length; i++) {
            System.out.print((char) byt[i]);
        }

        // 使用完关闭流
        fis.close();
    }
```

问题 1：缓冲区太小：

数据读取不完.

测试发现问题，由于数组太小，只装了 5 个字节。而文本的字节大于数组的长度。那么很显然可以将数组的长度定义大一些。例如 1024 个。

```
/**
     * 使用字节数组存储读到的数据
     * */
    private static void showContent2(String path) throws IOException {
        // 打开流
        FileInputStream fis = new FileInputStream(path);

        // 通过流读取内容
        byte[] byt = new byte[1024];
        int len = fis.read(byt);
        for (int i = 0; i < byt.length; i++) {
```

```
            System.out.print(byt[i]);
        }

        // 使用完关闭流
        fis.close();
    }
```

问题三:缓冲区有默认值.

测试,打印的效果打印出了很多 0，因为数组数组有默认初始化值，所以，我们将数组的数据全部都遍历和出来.现在需要的是取出数组中的部分数据.需要将循环条件修改仔细查看 api 文档。发现该方法 read(byte[] b)返回的是往数组中存入了多少个字节。就是数组实际存储的数据个数。

```
/**
     * 使用字节数组存储读到的数据
     * */
    private static void showContent2(String path) throws IOException {
        // 打开流
        FileInputStream fis = new FileInputStream(path);

        // 通过流读取内容
        byte[] byt = new byte[1024];
        int len = fis.read(byt);
        for (int i = 0; i <len; i++) {
            System.out.print(byt[i]);
    }

        // 使用完关闭流
        fis.close();
    }
```

总结:

问题一：为什么打印的不是字母而是数字，
是字母对应的码值。
如何显示字符，强转为 char 即可

问题二：注意：回车和换行的问题。
windows 的换车和换行是"\r\n" 对应码表是 13 和 10 。

3.1.3. 输入流读取方式 3:

使用 read(byte[] b,int off,int len)
查看 api 文档，
b 显然是一个 byte 类型数组，当做容器来使用
off，是指定从数组的什么位置开始存字节
len，希望读多少个
其实就是把数组的一部分当做流的容器来使用。告诉容器，从什么地方开始装要装多少。

```
/**
     * 把数组的一部分当做流的容器来使用
     * read(byte[] b,int off,int len)
     */
    private static void showContent3(String path) throws IOException {
        // 打开流
        FileInputStream fis = new FileInputStream(path);

        // 通过流读取内容
        byte[] byt = new byte[1024];
        // 从什么地方开始存读到的数据
        int start = 5;

        // 希望最多读多少个(如果是流的末尾，流中没有足够数据)
        int maxLen = 6;

        // 实际存放了多少个
        int len = fis.read(byt, start, maxLen);

        for (int i = start; i < start + maxLen; i++) {
            System.out.print((char) byt[i]);
        }

        // 使用完关闭流
        fis.close();
    }
```

需求 2：测试 skip 方法
通过 Io 流,读取"c:/a.txt"文件中的第 9 个字节到最后所有的内容并在控制台显示出来。
分析：其实就是要跳过文件中的一部分字节，需要查找 API 文档。可以使用 skip 方法 skip(long n)，参数跟的是要跳过的字节数。
我们要从第 9 个开始读，那么要跳过前 8 个即可。

```
/**
     * skip方法
     *
     * */
    private static void showContent4(String path) throws IOException {
        // 打开流
        FileInputStream fis = new FileInputStream(path);

        // 通过流读取内容
        byte[] byt = new byte[1024];
        fis.skip(8);
        int len = fis.read(byt);
```

```
        System.out.println(len);
        System.out.println("**********");
        for (int i = 0; i < len; i++) {
            System.out.println((char) byt[i]);
        }
        // 使用完关闭流
        fis.close();
    }
```

3.1.4. 输入流读取方式 4：

使用缓冲(提高效率),并循环读取(读完所有内容).

总结：读完文件的所有内容。很显然可以使用普通的 read 方法，一次读一个字节直到读到文件末尾。为了提高效率可以使用 read(byte[] byt);方法就是所谓的使用缓冲提高效率。我们可以读取大文本数据测试(大于 1K 的文本文件.)

```
/**
     * 使用字节数组当缓冲
     * */
    private static void showContent5(String path) throws IOException {
        FileInputStream fis = new FileInputStream(path);
        byte[] byt = new byte[1024];
        int len = fis.read(byt);
        System.out.println(len);
        String buffer = new String(byt, 0, len);
        System.out.print(buffer);
    }
```

注意：如何将字节数组转成字符串？ 可以通过创建字符串对象即可。

发现:一旦数据超过 1024 个字节，数组就存储不下。

如何将文件的剩余内容读完?

我们可以通过通过循环保证文件读取完。

```
/**
     * 使用字节数组当缓冲
     * */
    private static void showContent7(String path) throws IOException {
        FileInputStream fis = new FileInputStream(path);
        byte[] byt = new byte[1024];
        int len = 0;
        while ((len = fis.read(byt)) != -1) {
            System.out.println(new String(byt, 0, len));
        }
    }
```

3.2. OutputStream

字节输出流
案例：
1，写代码实现把"Hello World!"写到"c:/a.txt"文件中。
a，c:/a.txt 不存在时，测试一下。
b，c:/a.txt 存在时，也测试一下。
要写两个版本：
a，使用 write(int b) 实现。
b，使用 write(byte[] b) 实现。
2，在已存在的 c:/a.txt 文本文件中追加内容：“Java IO”。
显然此时需要向指定文件中写入数据。
使用的就是可以操作文件的字节流对象。OutputStream。该类是抽象类，需要使用具体的实现类来创建对象查看 API 文档，找到了 OutputStream 的实现类 FileOutputStream 创建 FileOutputStream 流对象,必须指定数据要存放的目的地。通过构造函数的形式。创建流对象时，调用了系统底层的资源。在指定位置建立了数据存放的目的文件。

流程：
1：打开文件输出流，流的目的地是指定的文件
2：通过流向文件写数据
3：用完流后关闭流

3.2.1. 输出流写出方式 1：

使用 write(int b)方法,一次写出一个字节.
在 C 盘下创建 a.txt 文本文件

```
import java.io.FileOutputStream;
import java.io.IOException;

public class IoTest2 {
    public static void main(String[] args) throws IOException {
        String path = "c:\\a.txt";
        writeTxtFile(path);
    }

    private static void writeTxtFile(String path) throws IOException {
        // 1：打开文件输出流，流的目的地是指定的文件
        FileOutputStream fos = new FileOutputStream(path);

        // 2：通过流向文件写数据
        fos.write('j');
        fos.write('a');
```

```
        fos.write('v');
        fos.write('a');
        // 3:用完流后关闭流
        fos.close();

    }
}
```

当 c 盘下的 a.txt 不存在会怎么样？
测试：将 c 盘下的 a.txt 文件删除，发现当文件不存在时，会自动创建一个，但是创建不了多级目录。
注意：使用 write(int b)方法，虽然接收的是 int 类型参数，但是 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。

3.2.2. 输出流写出方式 2：

使用 write(byte[] b),就是使用缓冲.提高效率.
上述案例中的使用了 OutputStram 的 write 方法，一次只能写一个字节。成功的向文件中写入了内容。但是并不高效，如和提高效率呢？是否应该使用缓冲，根据字节输入流的缓冲原理，是否可以将数据保存中字节数组中。通过操作字节数组来提高效率。查找 API 文档，在 OutputStram 类中找到了 write(byte[] b)方法，将 b.length 个字节从指定的 byte 数组写入此输出流中。
如何将字节数据保存在字节数组中，以字符串为例，”hello , world” 如何转为字节数组。显然通过字符串的 getBytes 方法即可。

```
public class IoTest2 {
    public static void main(String[] args) throws IOException {
        String path = "c:\\a.txt";
        writeTxtFile(path);
    }

    private static void writeTxtFile(String path) throws IOException {
        // 1：打开文件输出流，流的目的地是指定的文件
        FileOutputStream fos = new FileOutputStream(path);

        // 2：通过流向文件写数据
        byte[] byt = "java".getBytes();
        fos.write(byt);
        // 3:用完流后关闭流
        fos.close();
    }
}
```

仔细查看 a.txt 文本文件发现上述程序每运行一次，老的内容就会被覆盖掉。，那么如何不覆盖已有信息，能够往 a.txt 里追加信息呢。查看 API 文档，发现 FileOutputStream 类中的构造方法中有一个构造可以实现追加的功能 FileOutputStream(File file, boolean append) 第二个参数，append - 如果为 true，则将字节写入文件末尾处，而不是写入文件开始处

```
private static void writeTxtFile(String path) throws IOException {
        // 1：打开文件输出流，流的目的地是指定的文件
        FileOutputStream fos = new FileOutputStream(path,true);

        // 2：通过流向文件写数据
        byte[] byt = "java".getBytes();
        fos.write(byt);
        // 3:用完流后关闭流
        fos.close();
    }
```

3.3. 字节流文件拷贝

3.3.1. 字节输入输出流综合使用

通过字节输出流向文件中写入一些信息，并使用字节输入流把文件中的信息显示到控制台上。

```
public class IoTest3 {
    public static void main(String[] args) throws IOException {
        String path = "c:\\b.txt";
        String content = "hello java";

        writeFile(path, content);

        readFile(path);
    }

    public static void writeFile(String path, String content)
            throws IOException {
        // 打开文件输出流
        FileOutputStream fos = new FileOutputStream(path);
        byte[] buffer = content.getBytes();
        // 向文件中写入内容
        fos.write(buffer);
        // 关闭流
        fos.close();
```

```
    }

    public static void readFile(String path) throws IOException {
        FileInputStream fis = new FileInputStream(path);
        byte[] byt = new byte[1024];
        int len = 0;
        while ((len = fis.read(byt)) != -1) {
            System.out.println(new String(byt, 0, len));
        }
        // 关闭流
        fos.close();

    }
}
```

注意输出流的细节：

这个输出流显然只适合小数据的写入，如果有大数据想要写入，我们的 byte 数组，该如何定义？

上述案例中我们将输入流和输出流进行和综合使用,如果尝试进输出流换成文本文件就可以实现文件的拷贝了.

什么是文件拷贝？很显然，先开一个输入流，将文件加载到流中，再开一个输出流，将流中数据写到文件中。就实现了文件的拷贝。

分析：
第一步：需要打开输入流和输出流
第二步：读取数据并写出数据
第三步：关闭流

```
public class IoTest3 {

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

        String srcPath = "c:\\a.txt";
        String destPath = "d:\\a.txt";
        copyFile(srcPath, destPath);
    }

    public static void copyFile(String srcPath, String destPath)
            throws IOException {

    }

}
```

3.3.2. 字节流拷贝文件实现 1

读一个字节写一个字节 read 和 write

```
public class IoTest3 {

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

        String srcPath = "c:\\a.txt";
        String destPath = "d:\\a.txt";
        copyFile(srcPath, destPath);
    }

    public static void copyFile(String srcPath, String destPath)
            throws IOException {
        // 打开输入流，输出流
        FileInputStream fis = new FileInputStream(srcPath);
        FileOutputStream fos = new FileOutputStream(destPath);

        // 读取和写入信息
        int len = 0;
        while ((len = fis.read()) != -1) {
            fos.write(len);
        }

        // 关闭流
        fis.close();
        fos.close();
    }

}
```

文本文件在计算机中是以二进制形式存在的,可以通过 io 流来拷贝,那么图片能不能拷贝呢?视频呢? 音频呢?

```
public class IoTest3 {

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

        String srcPath = "c:\\秋.jpg";
        String destPath = "d:\\秋.jpg";
        copyFile(srcPath, destPath);
    }

    public static void copyFile(String srcPath, String destPath)
```

```
            throws IOException {
        // 打开输入流，输出流
        FileInputStream fis = new FileInputStream(srcPath);
        FileOutputStream fos = new FileOutputStream(destPath);

        // 读取和写入信息
        int len = 0;
        while ((len = fis.read()) != -1) {
            fos.write(len);
        }

        // 关闭流
        fis.close();
        fos.close();
    }

}
```

测试统统通过，所以字节流可以操作所有的文件。只是发现程序很慢，需要很长时间。特别是拷贝音频和视频文件时。

为什么？因为每次读一个字节再写一个字节效率很低。很显然这样效率低下的操作不是我们想要的。有没有更快更好的方法呢，是否可以使用缓冲区来提高程序的效率呢。

3.3.3. 字节流拷贝文件实现 2；

使用字节数组作为缓冲区

```
public static void copyFile2(String srcPath, String destPath)
            throws IOException {
        // 打开输入流，输出流
        FileInputStream fis = new FileInputStream(srcPath);
        FileOutputStream fos = new FileOutputStream(destPath);

        // 读取和写入信息
        int len = 0;

        // 使用字节数组，当做缓冲区
        byte[] byt = new byte[1024];
        while ((len = fis.read(byt)) != -1) {
            fos.write(byt);
        }

        // 关闭流
        fis.close();
```

```
        fos.close();
    }
```

问题 1：使用缓冲(字节数组)拷贝数据,拷贝后的文件大于源文件.
测试该方法，拷贝文本文件，仔细观察发现和源文件不太一致。

打开文件发现拷贝后的文件和拷贝前的源文件不同,拷贝后的文件要比源文件多一些内容问题就在于我们使用的容器，这个容器我们是重复使用的，新的数据会覆盖掉老的数据，显然最后一次读文件的时候，容器并没有装满，出现了新老数据并存的情况。
所以最后一次把容器中数据写入到文件中就出现了问题。
如何避免？使用 FileOutputStream 的 write(byte[] b, int off, int len)
b 是容器，off 是从数组的什么位置开始，len 是获取的个数，容器用了多少就写出多少。

```
public static void copyFile2(String srcPath, String destPath)
            throws IOException {
        // 打开输入流，输出流
        FileInputStream fis = new FileInputStream(srcPath);
        FileOutputStream fos = new FileOutputStream(destPath);

        // 读取和写入信息
        int len = 0;

        // 使用字节数组，当做缓冲区
        byte[] byt = new byte[1024];
        while ((len = fis.read(byt)) != -1) {
            fos.write(byt, 0, len);
        }

        // 关闭流
        fis.close();
        fos.close();
    }
```

使用缓冲拷贝视频，可以根据拷贝的需求调整数组的大小，一般是 1024 的整数倍。发现使用缓冲后效率大大提高。

3.4. 字节流的异常处理

上述案例中所有的异常都只是进行了抛出处理，这样是不合理的。所以上述代码并不完善，因为异常没有处理。
当我们打开流,读和写,关闭流的时候都会出现异常,异常出现后,后面的代码都不会执行了。假设打开和关闭流出现了异常，那么显然 close 方法就不会再执行。那么会对程序有什么影响？
案例：

```
public class IoTest4 {
```

```
    public static void main(String[] args) throws IOException,
            InterruptedException {
        String path = "c:\\b.txt";
        readFile(path);
    }

    private static void readFile(String path) throws IOException,
            InterruptedException {
        FileInputStream fis = new FileInputStream(path);
        byte[] byt = new byte[1024];
        int len = fis.read(byt);
        System.out.println(new String(byt, 0, len));
        // 让程序睡眠，无法执行到close方法。
        Thread.sleep(1000 * 10);
        fis.close();
    }
}
```

在执行该程序的同时我们尝试去删除 b.txt 文件。如果在该程序没有睡醒的话，我们是无法删除 b.txt 文件的。因为 b.txt 还被该程序占用着，这是很严重的问题，所以一定要关闭流。

目前我们是抛出处理，一旦出现了异常，close 就没有执行，也就没有释放资源。那么为了保证 close 的执行该如何处理呢。

那么就需要使用 try{} catch(){}finally{}语句。try 中放入可能出现异常的语句，catch 是捕获异常对象，fianlly 是一定要执行的代码

```
public class IoTest4 {
    public static void main(String[] args) throws IOException,
            InterruptedException {
        String path = "c:\\b.txt";
        readFile(path);
    }

    private static void readFile(String path) {
        FileInputStream fis = null;
        try {
            fis = new FileInputStream(path);
            byte[] byt = new byte[1024];
            int len = fis.read(byt);
            System.out.println(new String(byt, 0, len));
        } catch (IOException e) {
            // 抛出运行时异常
            throw new RuntimeException(e);
        } finally {
            // 把close方法放入finally中保证一定会执行
```

```
            // 先判断是否空指针
            if (fis != null) {
                try {
                    fis.close();
                } catch (Exception e) {
                    throw new RuntimeException(e);
                }

            }

        }

    }
}
```

文件拷贝的异常处理：

```
public static void copyFile(String srcPath, String destPath) {

        FileInputStream fis = null;
        FileOutputStream fos = null;
        try {
            fis = new FileInputStream(srcPath);
            fos = new FileOutputStream(destPath);

            byte[] byt = new byte[1024 * 1024];
            int len = 0;
            while ((len = fis.read(byt)) != -1) {

                fos.write(byt, 0, len);
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    throw new RuntimeException(e);
```

```
                }
            }

        }

    }
```

注意：

在最后的 close 代码中可能会有问题，两个 close，如果第一个 close 方法出现了异常，并抛出了运行时异常，那么程序还是停止了。下面的 close 方法就没有执行到。

那么为了保证 close 的执行，将第二个放到 fianlly 中即可。

```
public static void copyFile(String srcPath, String destPath) {

        FileInputStream fis = null;
        FileOutputStream fos = null;
        try {
            fis = new FileInputStream(srcPath);
            fos = new FileOutputStream(destPath);

            byte[] byt = new byte[1024 * 1024];
            int len = 0;
            while ((len = fis.read(byt)) != -1) {

                fos.write(byt, 0, len);
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        } finally {

            try {
                if (fis != null) {
                    fis.close();
                }
            } catch (IOException e) {
                throw new RuntimeException(e);
            } finally {
                if (fos != null) {
                    try {
                        fos.close();
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                }

            }
```

```
        }

    }
```

3.5. 字节缓冲流

3.5.1. 缓冲流

上述程序中我们为了提高流的使用效率,自定义了字节数组,作为缓冲区.Java 其实提供了专门的字节流缓冲来提高效率.

BufferedInputStream 和 BufferedOutputStream

BufferedOutputStream 和 BufferedOutputStream 类可以通过减少读写次数来提高输入和输出的速度。它们内部有一个缓冲区，用来提高处理效率。查看 API 文档，发现可以指定缓冲区的大小。其实内部也是封装了字节数组。没有指定缓冲区大小，默认的字节是 8192。

显然缓冲区输入流和缓冲区输出流要配合使用。首先缓冲区输入流会将读取到的数据读入缓冲区，当缓冲区满时，或者调用 flush 方法，缓冲输出流会将数据写出。

注意：当然使用缓冲流来进行提高效率时，对于小文件可能看不到性能的提升。但是文件稍微大一些的话，就可以看到实质的性能提升了。

```
public class IoTest5 {
    public static void main(String[] args) throws IOException {
        String srcPath = "c:\\a.mp3";
        String destPath = "d:\\copy.mp3";
        copyFile(srcPath, destPath);
    }

    public static void copyFile(String srcPath, String destPath)
            throws IOException {
        // 打开输入流，输出流
        FileInputStream fis = new FileInputStream(srcPath);
        FileOutputStream fos = new FileOutputStream(destPath);

        // 使用缓冲流
        BufferedInputStream bis = new BufferedInputStream(fis);
        BufferedOutputStream bos = new BufferedOutputStream(fos);

        // 读取和写入信息
        int len = 0;

        while ((len = bis.read()) != -1) {
            bos.write(len);
```

```
        }

        // 关闭流
        bis.close();
        bos.close();
}

}
```

4. 字符流

计算机并不区分二进制文件与文本文件。所有的文件都是以二进制形式来存储的，因此，从本质上说，所有的文件都是二进制文件。所以字符流是建立在字节流之上的，它能够提供字符层次的编码和解码。例如，在写入一个字符时，Java 虚拟机会将字符转为文件指定的编码（默认是系统默认编码），在读取字符时，再将文件指定的编码转化为字符。

常见的码表如下：

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。又称 Latin-1(拉丁编码)或"西欧语言"。ASCII 码是包含的仅仅是英文字母，并且没有完全占满 256 个编码位置，所以它以 ASCII 为基础，在空置的 0xA0-0xFF 的范围内，加入 192 个字母及符号，藉以供使用变音符号的拉丁字母语言使用。从而支持德文，法文等。因而它依然是一个单字节编码，只是比 ASCII 更全面。

GB2312：　英文占一个字节，中文占两个字节.中国的中文编码表。

GBK：　　中国的中文编码表升级，融合了更多的中文文字符号。

Unicode：　国际标准码规范，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。

UTF-8：　　最多用三个字节来表示一个字符。

（我们以后接触最多的是 iso8859-1、gbk、utf-8）

查看上述码表后，很显然中文的'中'在 iso8859-1 中是没有对映的编码的。或者一个字符在 2 中码表中对应的编码不同，例如有一些字在不同的编码中是有交集的，例如 bjg5 和 gbk 中的汉字简体和繁体可能是一样的，就是有交集，但是在各自码表中的数字不一样。

例如

使用 gbk 将中文保存在计算机中，

　　中　国

对映　100　200　　如果使用 big5 打开

可能　？　...

不同的编码对映的是不一样的。

很显然，我们使用什么样的编码写数据，就需要使用什么样的编码来对数据。

ISO8859-1：一个字节
GBK： 两个字节包含了英文字符和扩展的中文 ISO8859-1+中文字符
UTF-8 万国码，推行的。是 1~3 个字节不等长。英文存的是 1 个字节，中文存的是 3 个字节，是为了节省空间。

那么我们之前学习的流称之为字节流，以字节为单位进行操作之情的操作全是英文，如果想要操作中文呢？
测试：将指定位置的文件通过字节流读取到控制台

```
public class TestIo {
    public static void main(String[] args) throws IOException {
        String path = "c:\\a.txt";
        writFileTest();
        readFileByInputStream(path);
    }

    private static void readFileByInputStream(String path) throws
IOException {
        FileInputStream fis = new FileInputStream(path);

        int len = 0;
        while ((len = fis.read()) != -1) {
            System.out.print((char) len);
        }
    }

    private static void writFileTest() throws FileNotFoundException,
            IOException {
        // 创建文件对象
        File file = new File("c:\\a.txt");
        // 创建文件输出流
        FileOutputStream fos = new FileOutputStream(file);
        fos.write("中国".getBytes());
        fos.close();
    }
}
```

发现控制台输出的信息：
???ú 是这样的东西，打开 a.txt 文本发现汉字”中国”确实写入成功。
那么说明使用字节流处理中文有问题。
仔细分析，我们的 FileInputStream 输入流的 read() 一次是读一个字节的，返回的是一个 int 显然进行了自动类型提升。那么我们来验证一下”中国”对应的字节是什么
使用："中国".getBytes() 即可得到字符串对应的字节数组。是[-42, -48, -71, -6]
同样，将 read 方法返回值直接强转为 byte ，发现结果也是-42, -48, -71, -6 。

代码：

```
public class TestIo {
    public static void main(String[] args) throws IOException {
        String path = "c:\\a.txt";
        writFileTest();
        readFileByInputStream(path);
        //查看中国对应的编码
        System.out.println(Arrays.toString("中国".getBytes()));
    }

    private static void readFileByInputStream(String path) throws
IOException {
        FileInputStream fis = new FileInputStream(path);
        int len = 0;
        while ((len = fis.read()) != -1) {
            System.out.println((byte)len);
        }
    }

    private static void writFileTest() throws FileNotFoundException,
            IOException {
        // 创建文件对象
        File file = new File("c:\\a.txt");
        // 创建文件输出流
        FileOutputStream fos = new FileOutputStream(file);
        fos.write("中国\r\n".getBytes());
        fos.close();
    }

}
```

那么中国 对应的是-42, -48, -71, -6 是 4 个字节。 那就是一个中文占 2 个字节，（这个和编码是有关系的）

很显然，我们的中文就不能够再一个字节一个字节的读了。所以字节流处理字符信息时并不方便那么就出现了字符流。

字节流是 字符流是以字符为单位。

体验字符流：

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

        String path = "c:\\a.txt";
        readFileByReader(path);
    }
private static void readFileByReader(String path) throws IOException {
        FileReader fr = new FileReader(path);
```

```
        int len = 0;
        while ((len = fr.read()) != -1) {
            System.out.print((char) len);
        }
    }
```

总结：字符流就是：字节流 + 编码表，为了更便于操作文字数据。字符流的抽象基类：Reader ， Writer。
由这些类派生出来的子类名称都是以其父类名作为子类名的后缀，如 FileReader、FileWriter。

4.1. Reader

方法：
1，int read()：
读取一个字符。返回的是读到的那个字符。如果读到流的末尾，返回-1.
2，int read(char[])：
将读到的字符存入指定的数组中，返回的是读到的字符个数，也就是往数组里装的元素的个数。如果读到流的末尾，返回-1.
3，close()
读取字符其实用的是 window 系统的功能，就希望使用完毕后，进行资源的释放
由于 Reader 也是抽象类，所以想要使用字符输入流需要使用 Reader 的实现类。查看 API 文档。找到了 FileReader。
1，用于读取文本文件的流对象。
2，用于关联文本文件。
构造函数：在读取流对象初始化的时候，必须要指定一个被读取的文件。
如果该文件不存在会发生 FileNotFoundException.

```
public class IoTest1_Reader {

    public static void main(String[] args) throws Exception {
        String path = "c:/a.txt";
        // readFileByInputStream(path);
        readFileByReader(path);
    }

    /**
     * 使用字节流读取文件内容
     *
     * @param path
     */
    public static void readFileByInputStream(String path) throws
Exception {
        InputStream in = new FileInputStream(path);
```

```
        int len = 0;
        while ((len = in.read()) != -1) {
            System.out.print((char) len);
        }

        in.close();
    }

    /**
     * 使用字符流读取文件内容
     */
    public static void readFileByReader(String path) throws Exception {
        Reader reader = new FileReader(path);
        int len = 0;
        while ((len = reader.read()) != -1) {
            System.out.print((char) len);
        }

        reader.close();
    }

}
```

4.2. Writer

Writer 中的常见的方法：
1，write(ch)： 将一个字符写入到流中。
2，write(char[])： 将一个字符数组写入到流中。
3，write(String)： 将一个字符串写入到流中。
4，flush()：刷新流，将流中的数据刷新到目的地中，流还存在。
5，close()：关闭资源：在关闭前会先调用 flush()，刷新流中的数据去目的地。然流关闭。
发现基本方法和 OutputStream 类似，有 write 方法，功能更多一些。可以接收字符串。同样道理 Writer 是抽象类无法创建对象。查阅 API 文档，找到了 Writer 的子类 FileWriter
1：将文本数据存储到一个文件中。

```
public class IoTest2_Writer {

    public static void main(String[] args) throws Exception {
        String path = "c:/ab.txt";

        writeToFile(path);
    }
```

```
    /**
     * 写指定数据到指定文件中
     *
     */
    public static void writeToFile(String path) throws Exception {
        Writer writer = new FileWriter(path);
        writer.write('中');
        writer.write("世界".toCharArray());
        writer.write("中国");

        writer.close();
    }
}
```

2：追加文件：

默认的 FileWriter 方法新值会覆盖旧值，想要实现追加功能需要使用如下构造函数创建输出流 append 值为 true 即可。

```
FileWriter(String fileName, boolean append)
FileWriter(File file, boolean append)
```

3:flush 方法

如果使用字符输出流,没有调用 close 方法,会发生什么?

```
private static void writeFileByWriter(File file) throws IOException {
        FileWriter fw = new FileWriter(file);
        fw.write('新');
        fw.flush();
        fw.write("中国".toCharArray());
        fw.write("世界你好!!!".toCharArray());
        fw.write("明天");
        // 关闭流资源
        //fw.close();
    }
```

程序执行完毕打开文件,发现没有内容写入.原来需要使用 flush 方法. 刷新该流的缓冲。为什么只要指定 claose 方法就不用再 flush 方法,因为 close 也调用了 flush 方法.

4.3. 字符流拷贝文件

一个文本文件中有中文有英文字母，有数字。想要把这个文件拷贝到别的目录中。我们可以使用字节流进行拷贝，使用字符流呢？肯定也是可以的。

4.3.1. 字符流拷贝文件实现 1

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

```
        String path1 = "c:/a.txt";
        String path2 = "c:/b.txt";

        copyFile(path1, path2);
    }

    /**
     * 使用字符流拷贝文件
     */
    public static void copyFile(String path1, String path2) throws
Exception {
        Reader reader = new FileReader(path1);
        Writer writer = new FileWriter(path2);

        int ch = -1;
        while ((ch = reader.read()) != -1) {
            writer.write(ch);
        }

        reader.close();
        writer.close();
    }
```

但是这个一次读一个字符就写一个字符，效率不高。把读到的字符放到字符数组中，再一次性的写出。

4.3.2. 字符流拷贝文件实现 2

```
public static void main(String[] args) throws Exception {
        String path1 = "c:/a.txt";
        String path2 = "c:/b.txt";

        copyFile(path1, path2);
    }

public static void copyFile3(String path1, String path2) throws Exception
{
        Reader reader = new FileReader(path1);
        Writer writer = new FileWriter(path2);

        int ch = -1;
        char [] arr=new char[1024];
        while ((ch = reader.read(arr)) != -1) {
            writer.write(arr,0,ch);
```

```
        }

        reader.close();
        writer.close();
    }
```

字节流可以拷贝视频和音频等文件，那么字符流可以拷贝这些吗？
经过验证拷贝图片是不行的。发现丢失了信息，为什么呢？

计算机中的所有信息都是以二进制形式进行的存储（1010）图片中的也都是二进制在读取文件的时候字符流自动对这些二进制按照码表进行了编码处理，但是图片本来就是二进制文件，不需要进行编码。有一些巧合在码表中有对应，就可以处理，并不是所有的二进制都可以找到对应的。信息就会丢失。所以字符流只能拷贝以字符为单位的文本文件（以 ASCII 码为例是 127 个，并不是所有的二进制都可以找到对应的 ASCII，有些对不上的，就会丢失信息。）

4.4. 字符流的异常处理

```
public static void main(String[] args) throws Exception {
        String path1 = "c:/a.txt";
        String path2 = "c:/b.txt";

        copyFile2(path1, path2);
    }

/**
     * 使用字符流拷贝文件，有完善的异常处理
     */
    public static void copyFile2(String path1, String path2) {
        Reader reader = null;
        Writer writer = null;
        try {
            // 打开流
            reader = new FileReader(path1);
            writer = new FileWriter(path2);

            // 进行拷贝
            int ch = -1;
            while ((ch = reader.read()) != -1) {
                writer.write(ch);
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
```

```
        } finally {
            // 关闭流，注意一定要能执行到close()方法，所以都要放到finally代码块
中
            try {
                if (reader != null) {
                    reader.close();
                }
            } catch (Exception e) {
                throw new RuntimeException(e);
            } finally {
                try {
                    if (writer != null) {
                        writer.close();
                    }
                } catch (Exception e) {
                    throw new RuntimeException(e);
                }
            }
        }
    }
```

4.5. 字符流的缓冲区

查看 Reader 发现 Reader,操作的是字符,我们就不需要进行编码解码操作,由字符流读到二进制,自动进行解码得到字符,写入字符自动编码成二进制.

Reader 有一个子类 BufferedReader。子类继承父类显然子类可以重写父类的方法,也可以增加自己的新方法。例如一次读一行就是常用的操作.那么 BufferedReader 类就提供了这个方法,可以查看 readLine()方法具备 一次读取一个文本行的功能。很显然,该子类可以对功能进行增强。

体验 BufferedReader

```
public class IoTest_BufferedReader {
    public static void main(String[] args) throws IOException {
        readFile("c:\\a.txt");
    }

    private static void readFile(String path) throws IOException {
        Reader read = new FileReader(path);

        BufferedReader br = new BufferedReader(read);

        String line = null;
        while ((line = br.readLine()) != null) {
```

```
            System.out.println(line);
        }

    }
}
```

注意：
在使用缓冲区对象时，要明确，缓冲的存在是为了增强流的功能而存在，所以在建立缓冲区对象时，要先有流对象存在.
缓冲区的出现提高了对流的操作效率。原理：其实就是将数组进行封装。
使用字符流缓冲区拷贝文本文件.

```
public class Demo7 {
    public static void main(String[] args) throws IOException {
        // 关联源文件
        File srcFile = new File("c:\\linux大纲.txt");
        // 关联目标文件
        File destFile = new File("d:\\linux大纲.txt");
        // 实现拷贝
        copyFile(srcFile, destFile);

    }

    private static void copyFile(File srcFile, File destFile)
            throws IOException {
        // 创建字符输入流
        FileReader fr = new FileReader(srcFile);
        // 创建字符输出流
        FileWriter fw = new FileWriter(destFile);

        // 字符输入流的缓冲流
        BufferedReader br = new BufferedReader(fr);
        // 字符输出流的缓冲流
        BufferedWriter bw = new BufferedWriter(fw);

        String line = null;
        // 一次读取一行
        while ((line = br.readLine()) != null) {
            // 一次写出一行.
            bw.write(line);
            // 刷新缓冲
            bw.flush();
            // 进行换行,由于readLine方法默认没有换行.需要手动换行
            bw.newLine();
        }
```

```
        // 关闭流
        br.close();
        bw.close();
    }
}
```

4.6. 装饰器模式

需求：想要在读取的文件的每一行添加行号。

```
public class IoTest7_BufferedReader {

    public static void main(String[] args) throws IOException {
        readFile("c:\\a.txt");
    }

    private static void readFile(String path) throws IOException {
        Reader read = new FileReader(path);

        BufferedReader br = new BufferedReader(read);
        int count = 0;
        String line = null;
        while ((line = br.readLine()) != null) {
            count++;
            System.out.println(count+":"+line);
        }

    }
}
```

很容易的就可以实现。如果每次使用 BufferedReader 输出时都需要显示行号呢？每次都加？很显然,我们的 BufferedReader 继承了 Reader 对父类进行了功能的增强,那么我们也可以继承 BufferedReader 重写该类的 readLine 方法,进行功能的增强.

```
public class IoTest_BufferedReader {
    public static void main(String[] args) throws IOException {
        readFile("c:\\a.txt");
    }

    private static void readFile(String path) throws IOException {
        Reader read = new FileReader(path);

        BufferedReader br = new MyBufferedReader(read);
```

```
        String line = null;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }

    }
}

class MyBufferedReader extends BufferedReader {
    public MyBufferedReader(Reader read) {
        super(read);
    }

    int count;

    @Override
    public String readLine() throws IOException {
        String line = super.readLine();
        if (line != null) {
            count++;
            return count + ":" + line;

        } else {
            return null;
        }

    }
}
```

需求:
要在输出的一行前加上引号
可以再定义一个 BufferedReader 的子类,继承 BufferedReader 增强功能.

```
public class IoTest_BufferedReader {
    public static void main(String[] args) throws IOException {
        readFile("c:\\a.txt");
    }

    private static void readFile(String path) throws IOException {
        Reader read = new FileReader(path);
        BufferedReader br = new MyQutoBufferedReader(read);
        int count = 0;
        String line = null;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
            count++;
```

```
        }

    }
}


// quotation 引号
class MyQutoBufferedReader extends BufferedReader {

    public MyQutoBufferedReader(Reader reader) {
        super(reader);
    }

    public String readLine() throws IOException {
        String line = super.readLine();
        if (line != null) {

            return "\"" + line + "\"";

        } else {
            return null;
        }

    }
}
```

需求三：
既想要显示行号又想要显示引号
发现,就需要再定义子类,发现这样比较麻烦，代码臃肿.而且代码重复.
可以换一种方式.如下：
其实就是一个新类要对原有类进行功能增强.
1．在增强类中维护一个被增强的父类引用变量
2．在增强类的构造函数中初始化 1 中的变量
3．创建需要增强的方法，在刚方法中调用被被增强类的方法，并加以增强。

```
public class IoTest_BufferedReader {
    public static void main(String[] args) throws IOException {
        readFile("c:\\a.txt");
    }

    private static void readFile(String path) throws IOException {
        Reader read = new FileReader(path);
        BufferedReader bufferedReader = new BufferedReader(read);
        BufferedReader br = new MyQutoBufferedReader2(bufferedReader);
        br = new MyLineBufferedReader2(br);
```

```
        String line = null;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
    }
}

// quotation 引号
class MyQutoBufferedReader2 extends BufferedReader {
    private BufferedReader bufferedReader;

    public MyQutoBufferedReader2(BufferedReader bufferedReader) {
        super(bufferedReader);
        this.bufferedReader = bufferedReader;
    }

    public String readLine() throws IOException {
        String line = super.readLine();
        if (line != null) {

            return "\"" + line + "\"";

        } else {
            return null;
        }

    }
}

class MyLineBufferedReader2 extends BufferedReader {
    private BufferedReader bufferedReader;

    public MyLineBufferedReader2(BufferedReader bufferedReader) {
        super(bufferedReader);
        this.bufferedReader = bufferedReader;
    }

    int count;

    @Override
    public String readLine() throws IOException {
        String line = super.readLine();
        if (line != null) {
            count++;
```

```
            return count + ":" + line;

        } else {
            return null;
        }

    }
}
```

这就是装饰器模式

装饰器模式：

使用分层对象来动态透明的向单个对象中添加责任（功能）。

装饰器指定包装在最初的对象周围的所有对象都具有相同的基本接口。

某些对象是可装饰的，可以通过将其他类包装在这个可装饰对象的四周，来将功能分层。

装饰器必须具有和他所装饰的对象相同的接口。

JavaIO 中的应用：

Java I/O 类库需要多种不同的功能组合，所以使用了装饰器模式。

FilterXxx 类是 JavaIO 提供的装饰器基类，即我们要想实现一个新的装饰器，就要继承这些类。

装饰器与继承：

问题：

修饰模式做的增强功能按照继承的特点也是可以实现的，为什么还要提出修饰设计模式呢？

继承实现的增强类和修饰模式实现的增强类有何区别？

继承实现的增强类：

优点：代码结构清晰，而且实现简单

缺点：对于每一个的需要增强的类都要创建具体的子类来帮助其增强，这样会导致继承体系过于庞大。

修饰模式实现的增强类：

优点：内部可以通过多态技术对多个需要增强的类进行增强

缺点：需要内部通过多态技术维护需要增强的类的实例。进而使得代码稍微复杂。

5. 其他流

5.1. 序列流

也称为合并流。

5.1.1. SequenceInputStream

序列流，对多个流进行合并。

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。

注意：

构造函数

SequenceInputStream(InputStream s1, InputStream s2)

SequenceInputStream(InputStream s1, InputStream s2)

合并两个流

使用构造函数 SequenceInputStream(InputStream s1, InputStream s2)

```
private static void testSequenceInputStream() throws IOException {
        FileInputStream fis1 = new FileInputStream("c:\\a.txt");
        FileInputStream fis2 = new FileInputStream("c:\\b.txt");

        SequenceInputStream s1 = new SequenceInputStream(fis1, fis2);
        int len = 0;
        byte[] byt = new byte[1024];

        FileOutputStream fos = new FileOutputStream("c:\\z.txt");

        while ((len = s1.read(byt)) != -1) {
            fos.write(byt, 0, len);
        }
        s1.close();
    }
```

合并多个流：

```
public static void testSequenceInputStream() throws Exception {
        InputStream in1 = new FileInputStream("c:/a.txt");
        InputStream in2 = new FileInputStream("c:/b.txt");
        InputStream in3 = new FileInputStream("c:/c.txt");

        LinkedHashSet<InputStream> set = new
LinkedHashSet<InputStream>();
        set.add(in1);
        set.add(in2);
        set.add(in3);
        final Iterator<InputStream> iter = set.iterator();
```

```
        SequenceInputStream sin = new SequenceInputStream(
                new Enumeration<InputStream>() {
                    @Override
                    public boolean hasMoreElements() {
                        return iter.hasNext();
                    }

                    @Override
                    public InputStream nextElement() {
                        return iter.next();
                    }
                });

        FileOutputStream out = new FileOutputStream("c:/z.txt");

        for (int b = -1; (b = sin.read()) != -1;) {
            out.write(b);
        }
        sin.close();
        out.close();
    }
```

案例:将 map3 歌曲文件进行切割拷贝,并合并.

```
public class Demo2 {
    public static void main(String[] args) throws IOException {

        split(new File("c:\\a.mp3"), 10, new File("c:\\"));
        System.out.println("切割完毕");

        LinkedHashSet<InputStream> hs = new
LinkedHashSet<InputStream>();
        hs.add(new FileInputStream(new File("c:\\part.1.mp3")));
        hs.add(new FileInputStream(new File("c:\\part.2.mp3")));
        hs.add(new FileInputStream(new File("c:\\part.3.mp3")));
        hs.add(new FileInputStream(new File("c:\\part.4.mp3")));
        merage(hs, new File("c:\\merage.mp3"));
        System.out.println("合并完毕");
    }

    private static void merage(LinkedHashSet<InputStream> hs, File
dest)
            throws IOException {

        final Iterator<InputStream> it = hs.iterator();
        FileOutputStream fos = new FileOutputStream(dest);
```

```
        SequenceInputStream seq = new SequenceInputStream(
                new Enumeration<InputStream>() {

                    @Override
                    public boolean hasMoreElements() {

                        return it.hasNext();
                    }

                    @Override
                    public InputStream nextElement() {
                        return it.next();
                    }
                });
        byte[] byt = new byte[1024 * 1024];
        int len = 0;
        while ((len = seq.read(byt)) != -1) {
            fos.write(byt, 0, len);
        }
        seq.close();
        fos.close();
    }

    // 1. 切割文件
    /*
     * 切割文件,切割份数, 切割后保存路径
     */
    private static void split(File src, int count, File dir) throws
IOException {
        FileInputStream fis = new FileInputStream(src);
        FileOutputStream fos = null;
        byte[] byt = new byte[1024 * 1024];
        int len = 0;
        for (int i = 1; i <= count; i++) {
            len = fis.read(byt);
            if (len != -1) {
                fos = new FileOutputStream(dir + "part." + i + ".mp3");
                fos.write(byt, 0, len);
            }

            // fos.close();

        }
        fis.close();
```

```
    }
}
```

5.2. 对象的序列化

当创建对象时,程序运行时它就会存在,但是程序停止时,对象也就消失了.但是如果希望对象在程序不运行的情况下仍能存在并保存其信息,将会非常有用,对象将被重建并且拥有与程序上次运行时拥有的信息相同。可以使用对象的序列化。

对象的序列化： 将内存中的对象直接写入到文件设备中

对象的反序列化： 将文件设备中持久化的数据转换为内存对象

基本的序列化由两个方法产生：一个方法用于序列化对象并将它们写入一个流，另一个方法用于读取流并反序列化对象。

```
ObjectOutput
        writeObject(Object obj)
        将对象写入底层存储或流。
ObjectInput
        readObject()
        读取并返回对象。
```

5.2.1. ObjectOutputStream

5.2.2. ObjectInputStream

由于上述 ObjectOutput 和 ObjectInput 是接口，所以需要使用具体实现类。

```
ObjectOutput
    ObjectOutputStream 被写入的对象必须实现一个接口:Serializable
                       否则会抛出：NotSerializableException
ObjectInput
    ObjectInputStream     该方法抛出异常：ClassNotFountException
```

ObjectOutputStream 和 ObjectInputStream 对象分别需要字节输出流和字节输入流对象来构建对象。也就是这两个流对象需要操作已有对象将对象进行本地持久化存储。

案例：

序列化和反序列化 Cat 对象。

```
public class Demo3 {
    public static void main(String[] args) throws IOException,
            ClassNotFoundException {
```

```
        Cat cat = new Cat("tom", 3);
        FileOutputStream fos = new FileOutputStream(new
File("c:\\Cat.txt"));
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(cat);
        System.out.println(cat);
        oos.close();
        // 反序列化
        FileInputStream fis = new FileInputStream(new
File("c:\\Cat.txt"));
        ObjectInputStream ois = new ObjectInputStream(fis);
        Object readObject = ois.readObject();
        Cat cat2 = (Cat) readObject;
        System.out.println(cat2);
        fis.close();
}

class Cat implements Serializable {
    public String name;
    public int age;

    public Cat() {

    }

    public Cat(String name, int age) {

        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Cat [name=" + name + ", age=" + age + "]";
    }

}
```

例子关键点：

1. 声明 Cat 类实现了 Serializable 接口。是一个标示器，没有要实现的方法。
2. 新建 Cat 对象。
3. 新建字节流对象（FileOutputStream）进序列化对象保存在本地文件中。
4. 新建 ObjectOutputStream 对象，调用 writeObject 方法序列化 Cat 对象。
5. writeObject 方法会执行两个工作：序列化对象，然后将序列化的对象写入文件中。

6．反序列化就是调用 ObjectInputStream 的 readObject()方法。
7．异常处理和流的关闭动作要执行。

5.2.3. Serializable:

类通过实现 java.io.Serializable 接口以启用其序列化功能。未实现此接口的类将无法使其任何状态序列化或反序列化。可序列化类的所有子类型本身都是可序列化的。序列化接口没有方法或字段，仅用于标识可序列化的语义。
所以需要被序列化的类必须是实现 Serializable 接口，该接口中没有描述任何的属性和方法，称之为标记接口。

如果对象没有实现接口 Serializable，在进行序列化时会抛出：NotSerializableException 异常。

注意：
保存一个对象的真正含义是什么？如果对象的实例变量都是基本数据类型，那么就非常简单。但是如果实例变量是包含对象的引用，会怎么样？保存的会是什么？很显然在 Java 中保存引用变量的实际值没有任何意义，因为 Java 引用的值是通过 JVM 的单一实例的上下文中才有意义。通过序列化后，尝试在 JVM 的另一个实例中恢复对象，是没有用处的。
如下：
首先建立一个 Dog 对象，也建立了一个 Collar 对象。Dog 中包含了一个 Collar(项圈)现在想要保存 Dog 对象，但是 Dog 中有一个 Collar，意味着保存 Dog 时也应该保存 Collar。假如 Collar 也包含了其他对象的引用，那么会发生什么？意味着保存一个 Dog 对象需要清楚的知道 Dog 对象的内部结构。会是一件很麻烦的事情。

Java 的序列化机制可以解决该类问题，当序列化一个对象时，Java 的序列化机制会负责保存对象的所有关联的对象（就是对象图），反序列化时，也会恢复所有的相关内容。
本例中：如果序列化 Dog 会自动序列化 Collar。但是，只有实现了 Serializable 接口的类才可以序列化。如果只是 Dog 实现了该接口，而 Collar 没有实现该接口。会发生什么？
Dog 类和 Collar 类

```
import java.io.Serializable;

public class Dog implements Serializable {
    private Collar collar;
    private String name;

    public Dog(Collar collar, String name) {

        this.collar = collar;
        this.name = name;
    }

    public Collar getCollar() {
        return collar;
    }
```

```
}

class Collar {
    private int size;

    public int getSize() {
        return size;
    }

    public Collar(int size) {
        this.size = size;
    }

}
```

序列化

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class Demo4 {
    public static void main(String[] args) throws IOException {
        Collar coll = new Collar(10);
        Dog dog = new Dog(coll, "旺财");

        FileOutputStream fis = new FileOutputStream(new
File("c:\\dog.txt"));
        ObjectOutputStream os = new ObjectOutputStream(fis);
        os.writeObject(dog);
    }
}
```

执行程序，出现了运行时异常。

```
Exception in thread "main" java.io.NotSerializableException: Collar
```

所以我们也必须将 Dog 中使用的 Collar 序列化。但是如果我们无法访问 Collar 的源代码，或者无法使 Collar 可序列化，如何处理？

两种解决方法：

一：继承 Collar 类，使子类可序列化

但是：如果 Collar 是 final 类，就无法继承了。并且，如果 Collar 引用了其他非序列化对象，也无法解决该问题。

transient

此时就可以使用 transient 修饰符，可以将 Dog 类中的成员变量标识为 transient 那么在序列化 Dog 对象时，序列化就会跳过 Collar。

```
public class Demo4 {
    public static void main(String[] args) throws IOException,
            ClassNotFoundException {
        Collar coll = new Collar(10);
        Dog dog = new Dog(coll, "旺财");
        System.out.println(dog.getCollar().getSize());

        FileOutputStream fis = new FileOutputStream(new
File("c:\\dog.txt"));
        ObjectOutputStream os = new ObjectOutputStream(fis);
        os.writeObject(dog);

        // 反序列化
        FileInputStream fos = new FileInputStream(new
File("c:\\dog.txt"));
        ObjectInputStream ois = new ObjectInputStream(fos);
        Object readObject = ois.readObject();
        Dog dog2 = (Dog) readObject;
        // Collar未序列化。
        dog2.getCollar().getSize();
    }
}
```

这样我们具有一个序列化的 Dog 和非序列化的 Collar。
此时反序列化 Dog 后，访问 Collar，就会出现运行时异常

```
10
Exception in thread "main" java.lang.NullPointerException
```

注意：序列化不适用于静态变量，因为静态变量并不属于对象的实例变量的一部分。静态变量随着类的加载而加载，是类变量。由于序列化只适用于对象。

基本数据类型可以被序列化

```
public class Demo5 {
    public static void main(String[] args) throws IOException {
        // 创建序列化流对象
        FileOutputStream fis = new FileOutputStream(new
File("c:\\basic.txt"));
        ObjectOutputStream os = new ObjectOutputStream(fis);
        // 序列化基本数据类型
        os.writeDouble(3.14);
        os.writeBoolean(true);
        os.writeInt(100);
```

```
        os.writeInt(200);

        // 关闭流
        os.close();

        // 反序列化
        FileInputStream fos = new FileInputStream(new
File("c:\\basic.txt"));
        ObjectInputStream ois = new ObjectInputStream(fos);

        System.out.println(ois.readDouble());
        System.out.println(ois.readBoolean());
        System.out.println(ois.readInt());
        System.out.println(ois.readInt());

        fos.close();
    }
}
```

serialVersionUID
用于给类指定一个 UID。该 UID 是通过类中的可序列化成员的数字签名运算出来的一个 long 型的值。
只要是这些成员没有变化，那么该值每次运算都一样。
该值用于判断被序列化的对象和类文件是否兼容。
如果被序列化的对象需要被不同的类版本所兼容。可以在类中自定义 UID。
定义方式：static final long serialVersionUID = 42L;

5.3. Properties.

可以和流相关联的集合对象 Properties.

```
Map
    |--Hashtable
        |--Properties
```

Properties:该集合不需要泛型，因为该集合中的键值对都是 String 类型。
1，存入键值对：setProperty(key,value);
2，获取指定键对应的值：value getProperty(key);
3，获取集合中所有键元素：
Enumeration propertyNames();
在jdk1.6版本给该类提供一个新的方法。
Set<String> stringPropertyNames();

4，列出该集合中的所有键值对，可以通过参数打印流指定列出到的目的地。
list(PrintStream);
list(PrintWriter);
例：list(System.out):将集合中的键值对打印到控制台。
list(new PrintStream("prop.txt")):将集合中的键值对存储到prop.txt文件中。
5，可以将流中的规则数据加载进行集合，并称为键值对。
load(InputStream):
jdk1.6版本。提供了新的方法。
load(Reader):
注意：流中的数据要是"键=值" 的规则数据。
6，可以将集合中的数据进行指定目的的存储。
store(OutputStram,String comment)方法。
jdk1.6版本。提供了新的方法。
store(Writer ,String comment):
使用该方法存储时，会带着当时存储的时间。
注意：
Properties只加载key=value这样的键值对，与文件名无关，注释使用#
练习：记录一个程序运行的次数，当满足指定次数时，该程序就不可以再继续运行了。
通常可用于软件使用次数的限定。

```
public static void sysPropList() throws IOException {
        Properties prop = System.getProperties();

        // prop.list(System.out);// 目的是控制台。
        // 需求是：将jvm的属性信息存储到一个文件中。
        prop.list(new PrintStream("java.txt"));
    }

    public static void sysProp() {
        Properties prop = System.getProperties();

        Set<String> keys = prop.stringPropertyNames();

        for (String key : keys) {
            System.out.println(key + ":" + prop.getProperty(key));
        }
    }
```

Properties 类与配置文件
Map
    |--Hashtable
        |--Properties
注意：是一个 Map 集合，该集合中的键值对都是字符串。该集合通常用于对键值对形式的配置文件进行操作.
配置文件：将软件中可变的部分数据可以定义到一个文件中，方便以后更改，该文件称之为

配置文件。
优势：提高代码的维护性。
Properties： 该类是一个 Map 的子类，提供了可以快速操作配置文件的方法
load() ： 将文件设备数据装载为 Map 集合数据
get(key)： 获取 Map 中的数据
getProperty()获取 Map 中的数据特有方法
案例：

```
/*
     * 将配置文件中的数据通过流加载到集合中。
     */
    public static void loadFile() throws IOException {
        // 1,创建Properties(Map)对象
        Properties prop = new Properties();

        // 2.使用流加载配置文件。
        FileInputStream fis = new FileInputStream("c:\\qq.txt");

        // 3。使用Properties 对象的load方法将流中数据加载到集合中。
        prop.load(fis);

        // 遍历该集合
        Set<Entry<Object, Object>> entrySet = prop.entrySet();
        Iterator<Entry<Object, Object>> it = entrySet.iterator();
        while (it.hasNext()) {
            Entry<Object, Object> next = it.next();
            Object key = next.getKey();
            Object value = next.getValue();
        }
        // 通过键获取指定的值
        Object object = prop.get("jack");
        System.out.println(object);

        // 通过键修改值
        prop.setProperty("jack", "888888");

        // 将集合中的数据写入到配置文件中。
        FileOutputStream fos = new FileOutputStream("c:\\qq.txt");

        // 注释：
        prop.store(fos, "yes,qq");

        fos.close();
        fis.close();
```

```
    }
```

获取记录程序运行次数：

```
public class Demo6 {
    public static void main(String[] args) throws IOException {
        int count = 0;
        Properties pro = new Properties();

        File file = new File("c:\\count.ini");
        FileInputStream fis = null;
        if (!file.exists()) {
            file.createNewFile();
        }
        fis = new FileInputStream(file);
        pro.load(fis);
        String str = pro.getProperty("count");
        if (str != null) {
            count = Integer.parseInt(str);
        }
        if (count == 3) {
            System.out.println("使用次数已到，请付费");
            System.exit(0);
        }

        count++;
        System.out.println("欢迎使用本软件" + "你已经使用了：" + count + "
次");

        pro.setProperty("count", count + "");
        FileOutputStream fos = new FileOutputStream(new
File("c:\\count.ini"));
        pro.store(fos, "请保护知识产权");

        fis.close();
        fos.close();

    }
}
```

5.4. 打印流

PrintStream 可以接受文件和其他字节输出流，所以打印流是对普通字节输出流的增强，其中定义了很多的重载的 print()和 println()，方便输出各种类型的数据。

5.4.1. PrintStream

PrintWriter
1，打印流。
PrintStream:
是一个字节打印流，System.out对应的类型就是PrintStream。
它的构造函数可以接收三种数据类型的值。
1，字符串路径。
2，File对象。
3，OutputStream。

```
public static void main(String[] args) throws IOException {
    PrintStream ps = System.out;

    // 普通write方法需要调用flush或者close方法才会在控制台显示
    // ps.write(100);
    // ps.close();

    // 不换行打印
    ps.print(100);
    ps.print('a');
    ps.print(100.5);
    ps.print("世界");
    ps.print(new Object());
    System.out.println("---------------");
    // 换行
    ps.println(100);
    ps.println('a');
    ps.println(100.5);
    ps.println("世界");
    ps.println(new Object());

    // 重定向打印流
    PrintStream ps2 = new PrintStream(new File("c:\\a.txt"));
    System.setOut(ps2);
    // 换行
    ps2.println(100);
```

```
        ps2.println('a');
        ps2.println(100.5);
        ps2.println("世界");
        ps2.println(new Object());

        // printf(); 格式化
        ps2.printf("%d,%f,%c,%s", 100, 3.14, '中', "世界你好!!!");
        ps2.printf("%4s和%8s 打价格战", "京东", "苏宁");


    }   }
```

注意：打印流的三种方法

void print(数据类型 变量)

println(数据类型 变量)

printf(String format, Object... args)

可以自定数据格式

print 和println方法的区别在于,一个换行一个不换行

print 方法和write方法的却别在于,print提供自动刷新.

普通的write方法需要调用flush或者close方法才可以看到数据.

JDK1.5之后Java对PrintStream进行了扩展，增加了格式化输出方式，可以使用printf()重载方法直接格式化输出。但是在格式化输出的时候需要指定输出的数据类型格式。

No.	字符	描述
1	%s	表示内容为字符串
2	%d	表示内容为整数
3	%f	表示内容为小数
4	%c	表示内容为字符

5.4.2. PrintWriter

是一个字符打印流。构造函数可以接收四种类型的值。

1，字符串路径。

2，File对象。

对于1，2类型的数据，还可以指定编码表。也就是字符集。

3，OutputStream

4，Writer

对于3，4类型的数据，可以指定自动刷新。

注意：该自动刷新值为true时，只有三个方法可以用：println,printf,format.

如果想要既有自动刷新，又可执行编码。如何完成流对象的包装？

```
PrintWrter pw =
new PrintWriter(new OutputSteamWriter(new
FileOutputStream("a.txt"),"utf-8"),true);
```

如果想要提高效率。还要使用打印方法。

```
PrintWrter pw =
newPrintWriter(new  BufferdWriter(new OutputSteamWriter(
newFileOutputStream("a.txt"),"utf-8")),true);
```

```
public static void testPrintWriter() throws Exception {
        PrintWriter pw = new PrintWriter("c:/b.txt", "gbk");

        // pw.append("xxx");
        // pw.println(55);
        // pw.println('c');
        // pw.printf("%.1s与%4s打价格战, %c", "京东","苏宁", 'a');

        pw.close();

    }
```

Scanner

```
public static void testScanner() throws Exception {
        // Scanner scanner = new Scanner(new File("c:/test.txt"));
        Scanner scanner = new Scanner(System.in);

        System.out.println(scanner.nextInt());
        System.out.println(scanner.nextBoolean());

        scanner.close();
    }
```

5.5. 操作数组的流对象

5.5.1. 操作字节数组

ByteArrayInputStream

以及 ByteArrayOutputStream

```
toByteArray();
toString();
writeTo(OutputStream);
```

```
public static void testByteArrayInputStream() throws Exception {
        InputStream in = new ByteArrayInputStream(new byte[] { 65, 66,
```

```
67 });
        ByteArrayOutputStream out = new ByteArrayOutputStream();

        for (int b = -1; (b = in.read()) != -1;) {
            out.write(b);
        }

        in.close();
        out.close();

        System.out.println(Arrays.toString(out.toByteArray()));
        System.out.println(out);
    }
```

5.5.2. 操作字符数组

CharArrayReader
CharArrayWriter
对于这些流，源是内存。目的也是内存。
而且这些流并未调用系统资源。使用的就是内存中的数组。
所以这些在使用的时候不需要close。
操作数组的读取流在构造时，必须要明确一个数据源。所以要传入相对应的数组。
对于操作数组的写入流，在构造函数可以使用空参数。因为它内置了一个可变长度数组作为缓冲区。

```
public static void testCharArrayReader() throws Exception {
        CharArrayReader reader = new CharArrayReader(new char[] { 'A',
'b', 'c' });
        CharArrayWriter writer = new CharArrayWriter();

        for (int b = -1; (b = reader.read()) != -1;) {
            writer.write(b);
        }

        reader.close();
        writer.close();

        System.out.println(writer.toCharArray());
    }
```

这几个流的出现其实就是通过流的读写思想在操作数组。
类似的对象同理：
StringReader

StringWriter。

```
public static void testStringReader() throws Exception {
        StringReader reader = new StringReader("test 中国");
        StringWriter writer = new StringWriter();

        for (int b = -1; (b = reader.read()) != -1;) {
            writer.write(b);
        }

        reader.close();
        writer.close();

        System.out.println(writer.toString());
    }
```

5.6. 操作基本数据类型的流对象

5.6.1. DataInputStream

以及 DataOutputStream

查看 API 文档 DataInputStream 的信息。发现从底层输入流中读取基本 Java 数据类型。查看方法,有读一个字节,读一个 char 读一个 double 的方法,

DataInputStream 从数据流读取字节,并将它们转换为正确的基本数据类型值或字符串。该流有操作基本数据类型的方法.
有读的,那么必定有对应的写的就是 DataOutputStream 将基本类型的值或字符串转换为字节，并且将字节输出到数据流。
DataInputStream 类继承 FilterInputStream 类，并实现了 DataInput 接口。
DataOutputStream
类继承 FilterOutputStream 并实现了 DataOutput 接口。
例如：

DataInputStream
操作基本数据类型的方法：
int readInt():一次读取四个字节，并将其转成int值。
boolean readBoolean():一次读取一个字节。
short readShort();
long readLong();
剩下的数据类型一样。
String readUTF():按照 utf-8 修改版读取字符。注意，它只能读 writeUTF()写入的字符数据。
DataOutputStream

DataOutputStream(OutputStream)：
操作基本数据类型的方法：
writeInt(int)：一次写入四个字节。
注意和 write(int)不同。write(int)只将该整数的最低一个 8 位写入。剩余三个 8 位丢弃。
writeBoolean(boolean);
writeShort(short);
writeLong(long);
剩下是数据类型也也一样。
writeUTF(String)：按照 utf-8 修改版将字符数据进行存储。只能通过 readUTF 读取。

测试： DataOutputStream
使用 DataOutputStream 写数据文件。

```
public static void testDataInputStream() throws Exception {
        DataOutputStream out = new DataOutputStream(new
FileOutputStream(
                "c:/a.txt"));

        out.writeBoolean(true);
        out.writeByte(15); // 0x05 1 个字节
        out.writeBytes("abc"); // 0x 0041 2个字节
        out.writeChar('X'); // ??
        out.writeChars("xyz");
        out.writeLong(111);
        out.writeUTF("中国");

        out.close();

        DataInputStream in = new DataInputStream(
                new FileInputStream("c:/a.txt"));
        System.out.println(in.readBoolean());
        System.out.println(in.readByte());

        System.out.println(in.readByte());
        System.out.println(in.readByte());
        System.out.println(in.readByte());

        System.out.println(in.readChar());

        System.out.println(in.readChar());
        System.out.println(in.readChar());
        System.out.println(in.readChar());
```

```
        System.out.println(in.readLong());

        System.out.println(in.readUTF());
        in.close();
    }
```

6. 编码

什么是编码？

计算机中存储的都是二进制，但是要显示的时候，就是我们看到的却可以有中国 ，a 1 等字符

计算机中是没有存储字符的，但是我们却看到了。计算机在存储这些信息的时候，根据一个有规则的编号，当用户输入 a 有 a 对映的编号，就将这个编号存进计算机中这就是编码。

计算机只能识别二进制数据。

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。

例如：汉字 中

有一种编码：

中字在 utf 8 中对映的编码

utf-8 -->100

在 gbk 中呢？有可能就不是 100 了

gbk --> 150

很显然同一个信息在不同的编码中对映的数字也不同，

不同的国家和地区使用的码表是不同的，

gbk 是中国大陆

bjg5 是台湾同胞中的繁体字。所以如果给 big5 一个简体字是不认识的。

还有 ASCII 美国标准信息交换码

6.1. 码表

常见的码表如下：

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。又称 Latin-1(拉丁编码)或“西欧语言”。ASCII 码是包含的仅仅是英文字母，并且没有完全占满 256 个编码位置，所以它以 ASCII 为基础，在空置的 0xA0-0xFF 的范围内，加入 192 个字母及符号，藉以供使用变音符号的拉丁字母语言使用。从而支持德文，法文等。因而它依然是一个单字节编码，只是比 ASCII 更全面。

GB2312: 中国的中文编码表。

GBK: 中国的中文编码表升级，融合了更多的中文文字符号。

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。
UTF-8：　　最多用三个字节来表示一个字符。
（我们以后接触最多的是 iso8859-1、gbk、utf-8）

查看上述码表后，很显然中文的'中'在 iso8859-1 中是没有对映的编码的。或者一个字符在 2 中码表中对应的编码不同，例如有一些字在不同的编码中是有交集的，例如 bjg5 和 gbk 中的汉字简体和繁体可能是一样的，就是有交集，但是在各自码表中的数字不一样。

例如
使用 gbk 将中文保存在计算机中，
　　中　国
对映　100　200　　如果使用 big5 打开
可能　？　...
不同的编码对映的是不一样的。
很显然，我们使用什么样的编码写数据，就需要使用什么样的编码来对数据。

ISO8859-1：一个字节
GBK：　两个字节包含了英文字符和扩展的中文　　ISO8859-1+中文字符
UTF-8 万国码，推行的。是 1~3 个字节不等长。英文存的是 1 个字节，中文存的是 3 个字节，是为了节省空间。

6.2. 编码：

字符串---》字节数组
String 类的 getBytes() 方法进行编码，将字符串，转为对映的二进制，并且这个方法可以指定编码表。如果没有指定码表，该方法会使用操作系统默认码表。
注意：中国大陆的 Windows 系统上默认的编码一般为 GBK。在 Java 程序中可以使用 System.getProperty("file.encoding")方式得到当前的默认编码。

6.3. 解码：

字节数组---》字符串
String 类的构造函数完成。
String(byte[]　bytes)　使用系统默认码表
String(byte[],charset)指定码表
注意：我们使用什么字符集（码表）进行编码，就应该使用什么字符集进行解码，否则很有可能出现乱码（兼容字符集不会）。

```
// 编码操作与解码操作。
public static void main(String[] args) throws Exception {
    String value = System.getProperty("file.encoding");
    System.out.println("系统默认的编码为 " + value);
```

```
        String str = "中";

        // 编码操作
        byte[] bytes = str.getBytes();
        byte[] bytes2 = str.getBytes("gbk");// d6d0
        byte[] bytes3 = str.getBytes("utf-8");// e4b8ad

        System.out.println(Arrays.toString(bytes)); // [-42, -48]
        System.out.println(Arrays.toString(bytes2));// [-42, -48]
        System.out.println(Arrays.toString(bytes3));// [-28, -72, -83]

        // 解码操作
        // 编码gbk,解码utf-8乱码。
        String str2 = new String(bytes2, "utf-8");
        System.out.println(str2);

        // 编码utf-8 解码gbk，乱码
        str2 = new String(bytes3, "gbk");
        System.out.println(str2);
        // gbk兼容gb2312所以，没有问题。
        str = new String("中国".getBytes("gb2312"), "gbk");
        System.out.println(str);
    }
```

存文件时可以使用各种编码，但是解码的时候要对映的采用相同的解码方式。
我们的字符流自动的做了编码和解码的工作，写一个中文，字符流进行了编码，存到了计算机中读到了一个字符，字符流进行了解码，我们可以看到字符。因为文件存的都是二进制。但是拷贝图片时，是纯二进制，不是有意义的字符，所以码表无法转换。
字符流的弊端：

一：无法拷贝图片和视频。

二：拷贝文件使用字节流而不使用字符流，因为字符流读文件涉及到解码，会先解码，写文件的时候又涉及到编码，这些操作多余，而且读和写的码表不对应还容易引发问题。例如 FileReader 读文件，我们没有指定编码时，默认是按照系统编码 gbk 进行操作，如果读到 utf-8 的文件也是按照 gbk 编码进行解码，那就会出现问题。

6.4. 字节流读取中文

```
public class TestIo {
    public static void main(String[] args) throws IOException {
        readFileByInputStream2("c:\\a.txt");
    }
```

```
    private static void readFileByInputStream2(String path) throws
IOException {
        FileInputStream fis = new FileInputStream(path);
        int len = 0;

        while ((len = fis.read()) != -1) {
            System.out.print((char) len);
        }

    }
}
```

这个方法读取文本文件，中文是无法正确显示的。
很显然这些字节需要解码，可以将字节输入流读取的信息保存在字节数组中，指定对应的码表进行解码即可。

```
public class TestIo {
    public static void main(String[] args) throws IOException {
        readFileByInputStream("c:\\a.txt");
    }

    private static void readFileByInputStream(String path) throws
IOException {
        FileInputStream fis = new FileInputStream(path);
        int len = 0;
        byte[] buffer = new byte[1024];
        while ((len = fis.read(buffer)) != -1) {
            System.out.println(new String(buffer, 0, len, "gbk"));
        }

    }
}
```

注意：如果指定的编码表和解码表不对应就会出现问题

```
public class TestIo {
    public static void main(String[] args) throws IOException {
        // 该文件默认是gbk编码
        readFileByInputStream("c:\\a.txt");
    }

    private static void readFileByInputStream(String path) throws
IOException {
        FileInputStream fis = new FileInputStream(path);
        int len = 0;
```

```
        byte[] buffer = new byte[1024];
        while ((len = fis.read(buffer)) != -1) {
            // 使用utf-8 解码，解错。
            System.out.println(new String(buffer, 0, len, "utf-8"));
        }

    }
}
```

6.5. 字节流写出中文

需要编码，可以指定码表。就需要自己把字符串进行编码操作后，把得到的二进制内容通过字节流写入到文件中
使用 String 的 getBytes 方法，无参数的会使用系统默认的码表进行编码，也可以指定码表
系统默认编码

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

        String path = "c:\\test.txt";
        writeFileByOutputStream(path, "世界你好");
        readFileByInputStream(path);
    }

    private static void writeFileByOutputStream(String path, String
content)
            throws IOException {
        FileOutputStream fos = new FileOutputStream(path);

        // 把字符串进行编码操作，系统默认编码
        byte[] bytes = content.getBytes();
        // 内容通过字节流写入到文件中。
        fos.write(bytes);
        fos.close();
    }

    private static void readFileByInputStream(String path) throws
IOException {
        FileInputStream fis = new FileInputStream(path);
        int len = 0;
        byte[] buffer = new byte[1024];
```

```
        while ((len = fis.read(buffer)) != -1) {
            // 二进制解码，使用系统默认编码
            System.out.println(new String(buffer, 0, len));
        }

    }
}
```

使用 utf-8 进行编码

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

        String path = "c:\\test.txt";
        writeFileByOutputStream(path, "世界你好");
        readFileByInputStream(path);
    }

    private static void writeFileByOutputStream(String path, String
content)
            throws IOException {
        FileOutputStream fos = new FileOutputStream(path);

        // 把字符串进行编码操作
        byte[] bytes = content.getBytes("utf-8");
        // 内容通过字节流写入到文件中。
        fos.write(bytes);
        fos.close();
    }

    private static void readFileByInputStream(String path) throws
IOException {
        FileInputStream fis = new FileInputStream(path);
        int len = 0;
        byte[] buffer = new byte[1024];

        while ((len = fis.read(buffer)) != -1) {
            // 二进制解码，使用系统默认编码
            System.out.println(new String(buffer, 0, len,"utf-8"));
        }

    }
}
```

在明白了字节流也可以正确的处理中文字符之后，就应该明白字符流其实就是字节流在加上系统默认的码表。自动进行了编码和解码的操作。底层还是使用字节流读取文件。通过转换流的学习就可以明白这些道理。

6.6. 转换流

InputStreamReader

查看 API 文档，发现是字节流通向字符流的桥梁。查看构造，可以传递字节流，可以指定编码，该流可以实现什么功能？很显然可以包装我们的字节流，自动的完成节流编码和解码的工作。该流是一个 Reader 的子类，是字符流的体系。所以将转换流称之为字节流和字符流之间的桥梁。

InputStreamReader 是字节流通向字符流的桥梁

测试 InputStreamReader:

第一步：需要专门新建以 GBK 编码的文本文件。为了便于标识，我们命名为 gbk.txt 和以 UFT-8 编码的文本文件,命名为 utf.txt

第二步：分别写入汉字”中国”

第三步：编写测试方法，用 InputStreamReader 分别使用系统默认编码,GBK,UTF-8 编码读取文件.

```
public class Demo4 {
    public static void main(String[] args) throws IOException {
        File file = new File("c:\\a.txt");
        File fileGBK = new File("c:\\gbk.txt");
        File fileUTF = new File("c:\\utf.txt");
        // 默认编码
        testReadFile(file);
        // 传入gbk编码文件,使用gbk解码
        testReadFile(fileGBK, "gbk");
        // 传入utf-8文件,使用utf-8解码
        testReadFile(fileUTF, "utf-8");

    }

    // 该方法中nputStreamReader使用系统默认编码读取文件.
    private static void testReadFile(File file) throws
            IOException {
        FileInputStream fis = new FileInputStream(file);
        InputStreamReader ins = new InputStreamReader(fis);
        int len = 0;
        while ((len = ins.read()) != -1) {
```

```
            System.out.print((char) len);
        }
        ins.close();
        fis.close();
    }

    // 该方法使用指定编码读取文件
    private static void testReadFile(File file, String encod)
            throws IOException {
        FileInputStream fis = new FileInputStream(file);
        InputStreamReader ins = new InputStreamReader(fis, encod);
        int len = 0;
        while ((len = ins.read()) != -1) {
            System.out.print((char) len);
        }
        ins.close();
        }
}
```

注意：码表不对应

分别测试：

使用系统默认编码读取utf-8编码文件

使用utf-8编码读取gbk编码文件

使用"gbk"编码读取utf-8文件.

发现都会出现乱码的问题.

```
        // 使用系统默认编码读取utf-8
        testReadFile(fileUTF);
        // 传入gbk编码文件,使用utf-8解码
        testReadFile(fileGBK, "utf-8");
        // 传入utf-8文件,使用"gbk解码
        testReadFile(fileUTF, "gbk");
```

类 OutputStreamWriter

OutputStreamWriter

有了 InputStreamReader 可以转换 InputStream

那么其实还有 OutputStreamWriter 可以转换 OutputStream

OutputStreamWriter 是字符流通向字节流的桥梁

测试 OutputStreamWriter

一：分别使用 OutputStreamWriter 使用系统默认编码,GBK,UTF-8 相对应的默认编码文件,GBK 编码文件,UTF-8 编码文件中写出汉字"中国".

二：在使用上述案例中的 readFile 方法传入相对应码表读取.

```
public class TestIo {
    public class Demo4 {
    public static void main(String[] args) throws IOException {
```

```
        File file = new File("c:\\a.txt");
        File fileGBK = new File("c:\\gbk.txt");
        File fileUTF = new File("c:\\utf.txt");

        // 写入
        // 使用系统默认码表写入
        testWriteFile(file);
        // 使用gbk编码向gbk文件写入信息
        testWriteFile(fileGBK, "gbk");
        // 使用utf-8向utf-8文件中写入信息
        testWriteFile(fileUTF, "utf-8");

        // 读取
        // 默认编码
        testReadFile(file);
        // 传入gbk编码文件,使用gbk解码
        testReadFile(fileGBK, "gbk");
        // 传入utf-8文件,使用utf-8解码
        testReadFile(fileUTF, "utf-8");

    }

    // 使用系统码表将信息写入到文件中
    private static void testWriteFile(File file) throws IOException {
        FileOutputStream fos = new FileOutputStream(file);
        OutputStreamWriter ops = new OutputStreamWriter(fos);
        ops.write("中国");
        ops.close();
    }

    // 使用指定码表,将信息写入到文件中
    private static void testWriteFile(File file, String encod)
            throws IOException {
        FileOutputStream fos = new FileOutputStream(file);
        OutputStreamWriter ops = new OutputStreamWriter(fos, encod);
        ops.write("中国");
        ops.close();
    }

    // 该方法中nputStreamReader使用系统默认编码读取文件.
    private static void testReadFile(File file) throws IOException {
        FileInputStream fis = new FileInputStream(file);
        InputStreamReader ins = new InputStreamReader(fis);
        int len = 0;
```

```
        while ((len = ins.read()) != -1) {
            System.out.print((char) len);
        }
        ins.close();

    }

    // 该方法适合用指定编码读取文件
    private static void testReadFile(File file, String encod)
            throws IOException {
        FileInputStream fis = new FileInputStream(file);
        InputStreamReader ins = new InputStreamReader(fis, encod);
        int len = 0;
        while ((len = ins.read()) != -1) {
            System.out.print((char) len);
        }

        ins.close();
    }

}
```

注意：码表不对应的问题

分别测试：

向 GBK 文件中写入 utf-8 编码的信息

向 utf 文件中写入 gbk 编码的信息

发现文件都有问题,无法正常的读取了.

```
public static void main(String[] args) throws IOException {
        File file = new File("c:\\a.txt");
        File fileGBK = new File("c:\\gbk.txt");
        File fileUTF = new File("c:\\utf.txt");

        // 写入
        // // 使用系统默认码表写入
        // testWriteFile(file);
        // // 使用gbk编码向gbk文件写入信息
        // testWriteFile(fileGBK, "gbk");
        // // 使用utf-8向utf-8文件中写入信息
        // testWriteFile(fileUTF, "utf-8");

        testWriteFile(fileGBK);
        // 向GBK文件中写入utf-8编码的信息
        testWriteFile(fileGBK, "utf-8");
```

```
        // 向utf文件中写入gbk编码的信息
        testWriteFile(fileUTF, "gbk");

        // 读取
        // 默认编码
        testReadFile(file);
        // 传入gbk编码文件,使用gbk解码
        testReadFile(fileGBK, "gbk");
        // 传入utf-8文件,使用utf-8解码
        testReadFile(fileUTF, "utf-8");

    }
```

InputStreamReader：字节到字符的桥梁。
OutputStreamWriter：字符到字节的桥梁。

它们有转换作用，而本身又是字符流。所以在构造的时候，需要传入字节流对象进来。
构造函数：
InputStreamReader(InputStream)
通过该构造函数初始化，使用的是本系统默认的编码表 GBK。
InputStreamReader(InputStream,String charSet)
通过该构造函数初始化，可以指定编码表。
OutputStreamWriter(OutputStream)
通过该构造函数初始化，使用的是本系统默认的编码表 GBK。
OutputStreamWriter(OutputStream,String charSet)
通过该构造函数初始化，可以指定编码表。
注意：
操作文件的字符流对象是转换流的子类。

```
Reader
      |--InputStreamReader
            |--FileReader
Writer
      |--OutputStreamWriter
            |--FileWriter
```

注意：
在使用 FileReader 操作文本数据时，该对象使用的是默认的编码表。
如果要使用指定编码表时，必须使用转换流。

如果系统默认编码是 GBK 的：
FileReader fr = new FileReader("a.txt");//操作a.txt的中的数据使用的本系统默认的GBK。
操作a.txt中的数据使用的也是本系统默认的GBK。
InputStreamReader isr = new InputStreamReader(new

FileInputStream("a.txt"));
这两句的代码的意义相同。
但是：如果a.txt中的文件中的字符数据是通过utf-8的形式编码。使用FileReader就无能为力，那么在读取时，就必须指定编码表。那么转换流必须使用。
InputStreamReader isr =
new InputStreamReader(new FileInputStream("a.txt"),"utf-8");

7. 递归

递归做为一种算法在程序设计语言中广泛应用。是指函数/过程/子程序在运行过程中直接或间接调用自身而产生的重入现象。

（自己调用自己，有结束条件）

注意：递归时一定要明确结束条件。

数学中递归运算.

对于任何正整数 N ,N! (读作 N 的阶乘)的值定义为 1-N(包括 N)的所有的整数的成绩.因此 3! 就是 3!=3*2*1 =6;

5! 定义为 5!=5*4*3*2*1=120

那么整数 N 的阶乘 N! 可以表示为

1!=1

N!=N*(N-1)! for N>1

若果 N 等于 1 那么 1 的继承就是 1,其他所有 N! =N*(N-1)!,例如:50!=50*49!

49!=49*48! 48!=48*47! 一直持续到 1 出现.

如何使用 Java 程序计算阶乘?

```
public static long recursion(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * recursion(n - 1);
    }
}
```

7.1. 案例：

1，列出指定目录中所有的子孙文件与子孙目录名，只需要列出名称即可。

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。

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，效果如下所示：
|--src

```
|   |--cn
|   |   |--itcast
|   |   |   |--a_helloworld
|   |   |   |   |--HelloWorld.java
|   |   |   |--b_for
|   |   |   |   |--ForTest.java
|   |   |   |--c_api
|   |   |   |   |--Student.java
|--bin
|   |--cn
|   |   |--itcast
|   |   |   |--i_exception
|   |   |   |   |--ExceptionTest.class
|   |   |   |--h_linecount
|   |   |   |   |--LineCounter3.class
|   |   |   |   |--LineCounter2.class
|   |   |   |   |--LineCounter.class
|--lib
|   |--commons-io.jar
```

答案：

案例一：

```
// 1，列出指定目录中所有的子孙文件与子孙目录名，只需要列出名称即可。
    private static void listFile(File file) {

        File[] listFiles = file.listFiles();

        for (File f : listFiles) {
            if (f.isFile()) {
                System.out.println(f.getName());
            } else if (f.isDirectory()) {
                System.out.println(f.getName());
                listFile(f);
            }

        }
    }
public static void main(String[] args) {
        File file = new File("c:\\abc");
         listFile(file);
    }
```

案例二

```
// 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：
    private static void listFile2(File file, String str) {
```

```
        File[] listFiles = file.listFiles();

        for (int i = 0; i < listFiles.length; i++) {
            File f = listFiles[i];
            System.out.println(str + f.getName());

            if (f.isDirectory()) {
                listFile2(f, str + "-");
            }
        }
    }
public static void main(String[] args) {
        File file = new File("c:\\abc");
        String str = "-";
        listFile2(file, str);
```

案例三：

```
// 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
    private static void listFile3(File file, String str) {

        File[] listFiles = file.listFiles();

        for (File f : listFiles) {
            System.out.println(str + f.getName());
            if (f.isDirectory()) {
                listFile3(f, "|  " + str);
            }

        }
    }
public static void main(String[] args) {
        File file = new File("c:\\abc");
        file = new File("c:\\day18ide");
        file = new File("c:\\MyIo");
        str = "|-";
        listFile3(file, str);
```

7.2.练习：

1，删除一个非空的目录。
2，移动一个非空的目录到另一个地方（剪切）。
3，把 File 类中的重要方法设计代码测试一遍。

```
// 1，删除一个非空的目录。并加强健壮性
    private static void deleteFile(File file) {
        if (!file.exists()) {
            System.out.println("路径不存在");
            return;
        }
        if (!file.isDirectory()) {
            System.out.println("不是目录");
            return;
        }
        // 如果当前目录中有子目录和文件,先删除子目录和文件
        File[] listFiles = file.listFiles();
        for (File f : listFiles) {
            if (f.isFile()) {
                f.delete();
            } else if (f.isDirectory()) {
                deleteFile(f);
            }
        }
        // 删除当前目录
        file.delete();

    }
```

0o
练习 2:
使用 File 类的 renameTo 方法和递归实现非空目录的剪切.

```
public static void main(String[] args) throws IOException {
        // 重命名文件(成功)
        // File src = new File("c:\\aaa.txt");
        // File dest = new File("c:\\bbb.txt");
        // src.renameTo(dest);

        // //移动文件(成功)
        // File src = new File("c:\\aaa.txt");
        // File dest = new File("d:\\aaa.txt");
        // src.renameTo(dest);
```

```
        // 移动一个空目录(失败)
        // File src = new File("c:\\aaa");
        // File dest = new File("d:\\aaa");
        // System.out.println(src.renameTo(dest));

        // 使用File类和递归实现文件的剪切.
        File src = new File("c:\\abc");
        File dest = new File("d:\\");
        cutFile(src, dest);

    }

    // 移动一个非空的目录到另一个地方（剪切）。
    private static void cutFile(File srcDir, File dest) throws
IOException {
        if (!srcDir.exists() || !dest.exists()) {
            System.out.println("指定的源目录或者目标目录不存在");
            return;
        }
        if (!srcDir.isDirectory() || !dest.isDirectory()) {
            System.out.println("指定的源目录或者目标目录不是目录");
            return;
        }

        // 得到源目录名
        String srcDirName = srcDir.getName(); // abc
        // 根据源目录名创建新目录名
        File destDir = new File(dest + srcDirName); // d:\\abc dest 为
父路径
                                                    // srcDirName 为子路径
        // 创建目标目录
        destDir.mkdir();

        // 遍历源目录
        File[] listFiles = srcDir.listFiles();

        for (File f : listFiles) {
            // 如果是子源文件,使用renameTo方法,移动至目标目录中(该方法同时会删
除源目录中的文件)
            if (f.isFile()) {
                f.renameTo(new File(destDir, f.getName())); // 指定目标
文件的父目录,文件名(根据源文件名生成).
            } else if (f.isDirectory()) {
                // 如果是子目录,执行重复动作. 将源子目录 , 目标目录(父目录+//)
```

```
                cutFile(f, new File(destDir, File.separator)); // 指定源
目录,指定目的路径d:\\abc\\
            }
        }
        // 删除源目录
        srcDir.delete();

    }
```