IO流

1、本章内容

1. File类
2. 输入输出流
3. 字节流的实现
4. 过滤流
5. 内存流
6. 字符流
7. 随机文件访问
8. 序列化
9. 新IO

2、File类

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（持久化数据：永久的保存程序中创建的数据 - 文件、数据库）。

这些文件可以移动、传送、亦可以被其他程序使用。将数据存储在文件中，所以我们需要学习一个和文件有密切关系的类，叫做File类，File类声明在java.io包下。File类：文件和文件目录路径的抽象表示形式，与平台无关。

File类关心的是在磁盘上文件的存储，File类的一个对象，描述的是一个文件或文件夹（文件夹也可以称为目录）。

想要在Java程序中表示一个真实存在的文件或目录，那么必须有一个File对象，但是Java程序中的一个File对象，可能没有一个真实存在的文件或目录。File对象可以作为参数传递给流的构造器。

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（比如：新建、删除、重命名、修改时间、文件大小等方法），但是并未涉及到写入或读取文件内容的操作。如果需要读取或写入文件内容，必须使用IO流来完成。

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。

File类的用处

1、一个File对象可以代表一个文件夹或文件。
2、一个File对象也可以代表一个不存在的文件夹或文件。
3、构建一个File对象，不会在机器上创建一个文件。

```
/*
* 想要在Java程序中表示一个真实存在的文件或目录，那么必须有一个File对象，
* 但是Java程序中的一个File对象，可能没有一个真实存在的文件或目录。
* D:\io -> \在java里面表示转义符
*/
@Test
public void test1() {
    // 想要通过一个File对象来表示d:/io文件夹和d:/io/hello.txt文件
```

```
    File file1 = new File("d:\\io");

    File file2 = new File("d:\\io\\hello.txt");

    File file3 = new File("d:\\hello\\测试.txt");

    // 判断文件或目录是否存在
    System.out.println("file1对象对应的文件夹是否存在:" + file1.exists()); // true
    System.out.println("file2对象对应的文件是否存在:" + file2.exists()); // true
    System.out.println("file3对象对应的文件是否存在:" + file3.exists()); // false

    System.out.println("----------------------------------------");
    System.out.println("file1对象对应的是文件夹吗?" + file1.isDirectory()); // true
    System.out.println("file1对象对应的是文件吗?" + file1.isFile()); // false

    System.out.println("----------------------------------------");
    System.out.println("file2对象对应的是文件夹吗?" + file2.isDirectory()); //
false
    System.out.println("file2对象对应的是文件吗?" + file2.isFile()); // true

    System.out.println("----------------------------------------");
    System.out.println("file3对象对应的是文件夹吗?" + file3.isDirectory()); //
false
    System.out.println("file3对象对应的是文件吗?" + file3.isFile()); // false

    System.out.println("----------------------------------------");
    System.out.println("file2对象的内容长度:" + file2.length());
    System.out.println("file2对象的最后修改时间:" + new
Date(file2.lastModified()));
}
```

File类字段

```
static String pathSeparator 与系统有关的路径分隔符，为了方便，它被表示为一个字符串。

static char pathSeparatorChar    与系统有关的路径分隔符。

static String separator 与系统有关的默认名称分隔符，为了方便，它被表示为一个字符串。

static char separatorChar    与系统有关的默认名称分隔符。
```

```
@Test
public void test2() {
    // static String pathSeparator 与系统有关的路径分隔符，为了方便，它被表示为一个字符
串。
    System.out.println("与系统有关的路径分隔符字符串:" + File.pathSeparator); // ;

    // static char pathSeparatorChar     与系统有关的路径分隔符。
    System.out.println("与系统有关的路径分隔符字符:" + File.pathSeparatorChar); // ;

    // static String separator  与系统有关的默认名称分隔符，为了方便，它被表示为一个字符
串。
    System.out.println("与系统有关的默认名称分隔符字符串:" + File.separator); // \

    // static char separatorChar     系统有关的默认名称分隔符。
```

```
    System.out.println("与系统有关的默认名称分隔符字符:" + File.separatorChar); // \

}
```

File类构造函数

```
    File(File parent, String child) 根据 parent 抽象路径名和 child 路径名字符串创建一个新 File 实例。

    File(String pathname)   通过将给定路径名字符串转换为抽象路径名来创建一个新 File 实例。
        以pathname为路径创建File对象，可以是绝对路径或者相对路径。

    File(String parent, String child)   根据 parent 路径名字符串和 child 路径名字符串创建一个新 File 实例。

    File(URI uri)   通过将给定的 file: URI 转换为一个抽象路径名来创建一个新的 File 实例。

    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。
        创建File对象需要导包，import java.io.File。
        File对象没有无参数构造，创建对象需要传参。
        File类的对象，既可以代表文件也可以代表文件夹。
```

示例代码：

```
@Test
public void test5() {
   // File(String pathname)    通过将给定路径名字符串转换为抽象路径名来创建一个新 File 实例。
   // pathname可以是相对路径,也可以是绝对路径
   File file1 = new File("d:/io");
   System.out.println("file1对应的文件夹是否存在:" + file1.exists());

   System.out.println("-----------------------------------");
   // File(File parent, String child)  根据 parent 抽象路径名和 child 路径名字符串创建一个新 File 实例。
   File file2 = new File(file1, "hello.txt"); // d:/io  hello.txt ->d:/io/hello.txt
   System.out.println("file2对应的文件夹是否存在:" + file2.exists());

   System.out.println("-----------------------------------");
   // File(String parent, String child)    根据 parent 路径名字符串和 child 路径名字符串创建一个新 File 实例。
   File file3 = new File("d:/io", "hello.txt");
   System.out.println("file3对应的文件夹是否存在:" + file3.exists());
}
```

File类中常用的方法

1 创建

boolean createNewFile()　　当且仅当不存在具有此抽象路径名指定名称的文件时，不可分地创建一个新的空文件。

static File createTempFile(String prefix, String suffix)　　在默认临时文件目录中创建一个空文件，使用给定前缀和后缀生成其名称。

static File createTempFile(String prefix, String suffix, File directory)　在指定的目录中创建一个新的空文件，使用给定的前缀和后缀字符串生成其名称。

boolean mkdir()　　创建此抽象路径名指定的目录。

boolean mkdirs()　　创建此抽象路径名指定的目录，包括所有必需但不存在的父目录。

boolean renameTo(File dest)　　重新命名此抽象路径名表示的文件。

注意事项：如果你创建文件或者文件目录没有写盘符路径，那么，默认在项目路径下。

示例代码：

```
@Test
public void test6() throws IOException {
    File file1 = new File("d:/abc");
    // boolean createNewFile()  当且仅当不存在具有此抽象路径名指定名称的文件时，不可分地创建一个新的空文件。
    // file1.createNewFile();  // 在d盘下面创建的是abc文件,只是这个文件没有扩展名，不会给你创建文件夹;
    file1 = new File("d:/io/new.txt");
    boolean isNew = file1.createNewFile();
    if (isNew) {
        System.out.println("文件创建成功!");
    } else {
        System.out.println("文件创建失败!");
    }

    // static File createTempFile(String prefix, String suffix) 在默认临时文件目录中创建一个空文件，使用给定前缀和后缀生成其名称。
    File tempFile1 = File.createTempFile("Java", ".yyds");
    System.out.println(tempFile1); // File类重写了toString，会把文件的路径给反馈出来

    // static File createTempFile(String prefix, String suffix, File directory) 在指定的目录中创建一个新的空文件，使用给定的前缀和后缀字符串生成其名称。
    File directory = new File("d:/io");
    File tempFile2 = File.createTempFile("Java", ".yyds", directory);
    System.out.println(tempFile2);

    // boolean mkdir()       创建此抽象路径名指定的目录。
    /*File file2 = new File("d:/测试1");
        boolean isMk1 = file2.mkdir();
        if(isMk1){
            System.out.println("文件夹创建成功!");
        }else{
            System.out.println("文件夹创建失败!");
        }*/

    // 只能创建一级目录，如果有多级目录，创建失败
    File file3 = new File("d:/测试2/abc");
```

```
    // boolean isMk2 = file3.mkdir();

    // boolean mkdirs() 创建此抽象路径名指定的目录，包括所有必需但不存在的父目录。
    boolean isMk2 = file3.mkdirs();
    if (isMk2) {
        System.out.println("文件夹创建成功!");
    } else {
        System.out.println("文件夹创建失败!");
    }

    // boolean renameTo(File dest)       重新命名此抽象路径名表示的文件。
    // 情况一: 如果原文件对象和参数文件对象位于同一级目录,那么就是改名操作;
    /*File filex = new File("d:/io/hello.txt");
        File filey = new File("d:/io/HelloWorld.txt");
        filex.renameTo(filey);*/

    // 情况二: 如果原文件对象和参数文件对象不位于同一级目录,那么是文件移动操作;
    File filey = new File("d:/io/HelloWorld.txt");
    File filez = new File("d:/测试2/pp.txt");
    filey.renameTo(filez);
}
```

2 删除

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

    void deleteOnExit()      在虚拟机终止时，请求删除此抽象路径名表示的文件或目录。

    删除注意事项:
        Java中的删除不走回收站。
        要删除一个文件目录，请注意该文件目录内不能包含文件或者文件目录。
```

示例代码:

```
@Test
public void test7() throws InterruptedException {
    // xx.txt不存在,删除失败
    File file1 = new File("d:/io/xx.txt");

    // a.txt存在,存在可以删除
    file1 = new File("d:/io/a.txt"); // 文件被删除,不走回收站

    //  boolean delete()     删除此抽象路径名表示的文件或目录。
    boolean isDel = file1.delete();
    if (isDel) {
        System.out.println("文件删除成功!");
    } else {
        System.out.println("文件删除失败!");
    }

    // io目录下面不为空，删除失败
    file1 = new File("d:/io");
    System.out.println("是否删除了io目录:" + file1.delete());

    // void 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。
    file1 = new File("d:/io/b.txt");
```

```
    file1.deleteOnExit();

    System.out.println("deleteOnExit方法之后的语句");
    Thread.sleep(3000);

}

// 删除一个文件下夹所有的文件夹
File delfile=new File(str1);
File[] files=delfile.listFiles();
for(int i=0;i<files.length;i++){
    if(files[i].isDirectory()){
        files[i].delete();
    }
}
```

3 判断

```
    boolean canExecute()     测试应用程序是否可以执行此抽象路径名表示的文件。

    boolean canRead()    测试应用程序是否可以读取此抽象路径名表示的文件。

    boolean canWrite()       测试应用程序是否可以修改由此抽象路径名表示的文件。

    int compareTo(File pathname)     按字母顺序比较两个抽象路径名。

    boolean equals(Object obj)       测试此抽象路径名与给定对象是否相等。

    boolean exists()         测试此抽象路径名表示的文件或目录是否存在。

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

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

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

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

        Copied!
```

示例代码：

```
@Test
public void test8() {
    File file1 = new File("d:/io/hello.txt"); // 存在该文件
    File file2 = new File("d:/io/world.txt"); // 存在该文件,但是该文件只读
    File file3 = new File("d:/io/new.txt"); // 存在该文件,但是该文件隐藏
    File file4 = new File("d:/io/haha.txt"); // 不存在该文件

    // boolean canExecute() 测试应用程序是否可以执行此抽象路径名表示的文件。只要文件存在,
都是可以执行的
    System.out.println("file1是否可执行:" + file1.canExecute());// true
    System.out.println("file2是否可执行:" + file2.canExecute());// true
    System.out.println("file3是否可执行:" + file3.canExecute());// true
    System.out.println("file4是否可执行:" + file4.canExecute());// false
```

```
    System.out.println("---------------------------------------");
    // boolean canRead()     测试应用程序是否可以读取此抽象路径名表示的文件。只要文件存在，都是可读
    System.out.println("file1是否可读:" + file1.canRead());// true
    System.out.println("file2是否可读:" + file2.canRead());// true
    System.out.println("file3是否可读:" + file3.canRead());// true
    System.out.println("file4是否可读:" + file4.canRead());// false

    System.out.println("---------------------------------------");
    // boolean canWrite()        测试应用程序是否可以修改由此抽象路径名表示的文件。只要文件存在和非只读，都是可以写的
    System.out.println("file1是否可写:" + file1.canWrite());// true
    System.out.println("file2是否可写:" + file2.canWrite());// false
    System.out.println("file3是否可写:" + file3.canWrite());// true
    System.out.println("file4是否可写:" + file4.canWrite());// false

    System.out.println("---------------------------------------");
    // int compareTo(File pathname)      按字母顺序比较两个抽象路径名。
    // boolean equals(Object obj)        测试此抽象路径名与给定对象是否相等。
    // boolean exists()          测试此抽象路径名表示的文件或目录是否存在。
    System.out.println("file1是否存在:" + file1.exists());// true
    System.out.println("file2是否存在:" + file2.exists());// true
    System.out.println("file3是否存在:" + file3.exists());// true
    System.out.println("file4是否存在:" + file4.exists());// false

    System.out.println("---------------------------------------");
    // boolean isAbsolute()      测试此抽象路径名是否为绝对路径名。
    System.out.println("file1是否为绝对路径名:" + file1.isAbsolute());// true
    System.out.println("file2是否为绝对路径名:" + file2.isAbsolute());// true
    System.out.println("file3是否为绝对路径名:" + file3.isAbsolute());// true
    System.out.println("file4是否为绝对路径名:" + file4.isAbsolute());// true
    File file5 = new File("config/db.properties");
    System.out.println("file5是否为绝对路径名:" + file4.isAbsolute());// false

    System.out.println("---------------------------------------");
    // boolean isDirectory()     测试此抽象路径名表示的文件是否是一个目录。
    // boolean isFile() 测试此抽象路径名表示的文件是否是一个标准文件。
    // boolean isHidden()    测试此抽象路径名指定的文件是否是一个隐藏文件。
    System.out.println("file1是否是隐藏文件:" + file1.isHidden());// false
    System.out.println("file2是否是隐藏文件:" + file2.isHidden());// false
    System.out.println("file3是否是隐藏文件:" + file3.isHidden());// true
    System.out.println("file4是否是隐藏文件:" + file4.isHidden());// false
}
```

4 获取

File getAbsoluteFile()　返回此抽象路径名的绝对路径名形式。

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

File getCanonicalFile()      返回此抽象路径名的规范形式。

String getCanonicalPath()    返回此抽象路径名的规范路径名字符串。

long getFreeSpace()      返回此抽象路径名指定的分区中未分配的字节数。
```

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

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。

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

String getPath()　将此抽象路径名转换为一个路径名字符串。

long getTotalSpace()　返回此抽象路径名指定的分区大小。

long getUsableSpace()　返回此抽象路径名指定的分区上可用于此虚拟机的字节数。

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

long length()　返回由此抽象路径名表示的文件的长度。

String[] list()　返回一个字符串数组，这些字符串指定此抽象路径名表示的目录中的文件和目录。

String[] list(FilenameFilter filter)　返回一个字符串数组，这些字符串指定此抽象路径名表示的目录中满足指定过滤器的文件和目录。

File[] listFiles()　返回一个抽象路径名数组，这些路径名表示此抽象路径名表示的目录中的文件。

File[] listFiles(FileFilter filter)　返回抽象路径名数组，这些路径名表示此抽象路径名表示的目录中满足指定过滤器的文件和目录。

File[] listFiles(FilenameFilter filter) 返回抽象路径名数组，这些路径名表示此抽象路径名表示的目录中满足指定过滤器的文件和目录。

static File[] listRoots()　列出可用的文件系统根。

```
@Test
public void test9() {
    // 相对路径的写法
    File file1 = new File("config/db.properties");
    System.out.println("file1的抽象路径表现形式:" + file1);

    // File getAbsoluteFile()　返回此抽象路径名的绝对路径名形式。
    File file2 = file1.getAbsoluteFile(); // 获取file1的绝对路径,然后通过绝对路径构建了一个对象;
    System.out.println("file2的抽象路径表现形式:" + file2);

    // equals() 比较的是两个对象的抽象路径名;
    System.out.println(file1.equals(file2)); // false

    // String getAbsolutePath()　返回此抽象路径名的绝对路径名字符串。
    System.out.println("file1的绝对路径:" + file1.getAbsolutePath());

    // long getFreeSpace()　返回此抽象路径名指定的分区中未分配的字节数
    System.out.println("file1所在分区中未分配的字节数:" + file1.getFreeSpace());

    // String getName() 返回由此抽象路径名表示的文件或目录的名称。
    System.out.println("file1的文件或文件名:" + file1.getName()); // db.properties
    System.out.println("file2的文件或文件名:" + file2.getName()); // db.properties
```

```
    // 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。
    System.out.println("file1的父路径:" + file1.getParent()); // config
    System.out.println("file2的父路径:" + file2.getParent()); // config

    // File getParentFile()      返回此抽象路径名父目录的抽象路径名；如果此路径名没有指定父目录，则返回 null。
    // String getPath() 将此抽象路径名转换为一个路径名字符串。
    System.out.println("file1路径名字符串:" + file1.getPath());
    System.out.println("file2路径名字符串:" + file2.getPath());

    // long getTotalSpace() 返回此抽象路径名指定的分区大小。
    System.out.println("file1所在分区的大小:" + file1.getTotalSpace());

    // long lastModified()  返回此抽象路径名表示的文件最后一次被修改的时间。
    // long length()     返回由此抽象路径名表示的文件的长度。
    System.out.println("file1的长度：" + file1.length());

}

@Test
public void test10() {
    File file = new File("d:/io");

    // String[] list()       返回一个字符串数组，这些字符串指定此抽象路径名表示的目录中的文件和目录。

    // 遍历一：打印当前目录的所有文件和文件夹
    /*String[] list = file.list();
        for (String str : list) {
            System.out.println(str);
        }*/

    System.out.println(file + "下面的所有的子文件和文件夹:");
    traversalFiles("d:/io");

    System.out.println("------------------------------");
    System.out.println("io文件夹下面的所有txt文件:");
    String[] list = file.list();
    for (String str : list) {
        if (str.endsWith(".txt")) {
            System.out.println(str);
        }
    }

    System.out.println("------------------------------");
    System.out.println("io文件夹下面的所有txt文件:");
    // String[] list(FilenameFilter filter) 返回一个字符串数组，这些字符串指定此抽象路径名表示的目录中满足指定过滤器的文件和目录。
    String[] list1 = file.list(new FilenameFilter() {
        @Override
        public boolean accept(File dir, String name) {
            return name.endsWith(".txt");
        }
    });

    System.out.println(Arrays.asList(list1));
```

```
    // File[] listFiles()          返回一个抽象路径名数组，这些路径名表示此抽象路径名表示的目录中的文件。
    // 和list方法类似，但是返回值是File数组

    // static File[] listRoots()     列出可用的文件系统根。
    File[] files = File.listRoots();
    System.out.println(Arrays.asList(files));
}

/*
     * 遍历所有的文件
     */
public void traversalFiles(String pathName) {
    // 先判定参数路径为null的问题
    if (pathName == null) {
        System.out.println("路径名不能为null");
        return;
    }

    File file = new File(pathName);

    // 文件是否存在 和 不能为null
    if (file.exists()) {
        if (file.isDirectory()) {
            System.out.println(file.getName() + "是文件夹");
            // 获取指定文件夹下面的所有的文件String
            /*String[] list = file.list();
                for (String str : list) {
                    traversalFiles(file.getAbsolutePath()+"/"+str);
                }*/

            File[] list = file.listFiles();
            for (File childFile : list) {
                traversalFiles(childFile.getAbsolutePath());
            }
        } else {
            System.out.println("\t" + file.getName());
        }
    } else {
        System.out.println("文件路径不存在!");
    }
}
```

3、输入输出流

标准I/O：Java程序可通过命令行参数与外界进行简短的信息交换，同时，也规定了与标准输入、输出设备，如键盘、显示器进行信息交换的方式。而通过文件可以与外界进行任意数据形式的信息交换。

System.in和System.out分别代表了系统标准的输入和输出设备。

默认输入设备是：键盘，输出设备是：显示器。

System.in的类型是InputStream。

System.out的类型是PrintStream，其是OutputStream的子类/FilterOutputStream 的子类。

重定向：通过System类的setIn，setOut方法对默认设备进行改变。

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class StandardIOTest {
    public static void main(String[] args) {

        System.out.println("请输入信息(退出输入e或exit):");
        // 把"标准"输入流(键盘输入)这个字节流包装成字符流,再包装成缓冲流
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        String s = null;
        try {
            while ((s = br.readLine()) != null) { // 读取用户输入的一行数据 --> 阻塞
程序
                if ("e".equalsIgnoreCase(s) || "exit".equalsIgnoreCase(s)) {
                    System.out.println("安全退出!!");
                    break;
                }
                // 将读取到的整行字符串转成大写输出
                System.out.println("-->:" + s.toUpperCase());
                System.out.println("继续输入信息：");
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (br != null) {
                    br.close(); // 关闭过滤流时,会自动关闭它包装的底层节点流
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

    }
}
```

4、字节流

字节流：每次读取（写出）一个字节，当传输的资源文件有中文时，就会出现乱码。

FileInputStream

```
    public class FileInputStream extends InputStream

    FileInputStream 从文件系统中的某个文件中获得输入字节。哪些文件可用取决于主机环境。

    FileInputStream 用于读取诸如图像数据之类的原始字节流。要读取字符流，请考虑使用 FileReader。
```

构造函数:

```
    FileInputStream(File file)  通过打开一个到实际文件的连接来创建一个 FileInputStream，该文件通过文件系统中的 File 对象 file 指定。

    FileInputStream(FileDescriptor fdObj)   通过使用文件描述符 fdObj 创建一个 FileInputStream，该文件描述符表示到文件系统中某个实际文件的现有连接。

    FileInputStream(String name)    通过打开一个到实际文件的连接来创建一个 FileInputStream，该文件通过文件系统中的路径名 name 指定。
```

成员方法:

```
    int available() 返回下一次对此输入流调用的方法可以不受阻塞地从此输入流读取（或跳过）的估计剩余字节数。

    void close()    关闭此文件输入流并释放与此流有关的所有系统资源。

    int read()  从此输入流中读取一个数据字节。

    int read(byte[] b)  从此输入流中将最多 b.length 个字节的数据读入一个 byte 数组中。

    int read(byte[] b, int off, int len)    从此输入流中将最多 len 个字节的数据读入一个 byte 数组中。

    long skip(long n)   从输入流中跳过并丢弃 n 个字节的数据。
```

```java
import java.io.File;
import java.io.FileInputStream;

public class FileInputStreamTest01 {
    public static void main(String[] args) throws Exception {
        //1、把要操作硬盘上的文件，需要把文件转化成对象来操作
        //a.txt的内容如下:abcdefghigklmn
        File file = new File("src/a.txt");

        //2、调用FileInputStream构造函数把file类传递进去
        FileInputStream fis = new FileInputStream(file);
        //构造函数不仅可以传递file类,也可以传递String类型的地址
        //FileInputStream fis1=new FileInputStream("src/a.txt");

        //3、调用read()方法读取字节
        int x = fis.read();//从此输入流中读取一个数据字节,返回值是int
        //这里对应的是97,指的是ascii
        System.out.println(x);//97
```

```
        //我们可以使用强转的方式转换为字符
        System.out.println((char) x);//a

        System.out.println("--------------------------");
        //如果想要读取所有,那么只需要使用循环读取即可
        //read()当读取到末尾的时候,返回值是-1
        while (x != -1) {
            char ch = (char) x;
            System.out.print(ch);
            //继续读取
            x = fis.read();//d
        }

        //4、close()关闭此文件输入流并释放与此流有关的所有系统资源
        fis.close();
    }
}
```

FileOutputStream

```
    public class FileOutputStream extends OutputStream
```

文件输出流是用于将数据写入 File 或 FileDescriptor 的输出流。文件是否可用或能否可以被创建取决于基础平台。特别是某些平台一次只允许一个 FileOutputStream（或其他文件写入对象）打开文件进行写入。在这种情况下，如果所涉及的文件已经打开，则此类中的构造方法将失败。

FileOutputStream 用于写入诸如图像数据之类的原始字节的流。要写入字符流，请考虑使用 FileWriter。

构造函数:

FileOutputStream(File file)　　创建一个向指定 File 对象表示的文件中写入数据的文件输出流。

FileOutputStream(File file, boolean append)　　创建一个向指定 File 对象表示的文件中写入数据的文件输出流。

FileOutputStream(FileDescriptor fdObj)　创建一个向指定文件描述符处写入数据的输出文件流，该文件描述符表示一个到文件系统中的某个实际文件的现有连接。

FileOutputStream(String name)　创建一个向具有指定名称的文件中写入数据的输出文件流。

FileOutputStream(String name, boolean append)　创建一个向具有指定 name 的文件中写入数据的输出文件流。

成员方法:

```
    void close()     关闭此文件输出流并释放与此流有关的所有系统资源。

    void write(int b)    将指定字节写入此文件输出流。

    void write(byte[] b)     将 b.length 个字节从指定 byte 数组写入此文件输出流中。

    void write(byte[] b, int off, int len)  将指定 byte 数组中从偏移量 off 开始的
len 个字节写入此文件输出流。
```

示例代码:

```
import java.io.FileOutputStream;

public class FileOutputStreamTest01 {
    public static void main(String[] args) throws Exception {
        //输入流，要从指定的文件读取内容到内存中，如果文件不存在，则报错
        //输出流，要从内存中写内容到指定文件，如果文件不存在，则创建一个新的文件
        FileOutputStream fos = new FileOutputStream("src/b.txt", true);

        //write(int b) 将指定字节写入此文件输出流。
        fos.write('a');
        //输出完毕之后,如果src下面有b.txt,那么就会向里面写入a,如果没有,则创建b.txt,并且写
入内容

        String str = "abcdefg";
        //把字符串转换为数组
        byte b[] = str.getBytes();
        //write(byte[] b) 将 b.length 个字节从指定 byte 数组写入此文件输出流中。
        fos.write(b);

        //close() 关闭此文件输出流并释放与此流有关的所有系统资源。
        fos.close();

        //流关闭之后,再次写入内容的时候,会把之前的内容给覆盖掉,那么是因为调用构造函数没传递
true参数
    }

}
```

5、字符流

字符流：每次读取（写出）一个字符，有中文时，使用该流就可以正确传输显示中文。

字符输入流 FileReader;
字符输出流 FileWriter;

字符流的操作和字节流一致，但是我们推荐在使用读取中文内容的时候使用字符流，因为字节流是一个个读取，那么在读取过程中，可能把一个字符拆分为两个字节来读取，可能会出现一定问题。

FileReader

FileReader 用来读取字符文件的便捷类。通常用来读取和中文相关的内容;

public class FileReader extends InputStreamReader
InputStreamReader 是字节流通向字符流的桥梁：它使用指定的 charset 读取字节并将其解码为字符。它使用的字符集可以由名称指定或显式给定，或者可以接受平台默认的字符集。

public class InputStreamReader extends Reader
Reader 用于读取字符流的抽象类。

我们从这样的体系结构可以看出来，这就是我们之前说的，字符流的底层还是来进行的字节读取而已;

构造函数:

FileReader(File file) 在给定从中读取数据的 File 的情况下创建一个新 FileReader。

FileReader(FileDescriptor fd) 在给定从中读取数据的 FileDescriptor 的情况下创建一个新 FileReader。

FileReader(String fileName) 在给定从中读取数据的文件名的情况下创建一个新 FileReader。

成员方法:

FileReader类没有定义相关成员方法，都是从InputStreamReader里面继承而来的方法;

void close() 关闭该流并释放与之关联的所有资源。

String getEncoding() 返回此流使用的字符编码的名称。

int read() 读取单个字符。

int read(char[] ch) 从此输入流中将最多 b.length 个字节的数据读入一个 char 数组中。

int read(char[] b, int off, int len) 从此输入流中将最多 len 个字节的数据读入一个 char 数组中。

FileWriter

FileWriter 用来写入字符文件的便捷类。

public class FileWriter extends OutputStreamWriter

OutputStreamWriter 是字符流通向字节流的桥梁：可使用指定的 charset 将要写入流中的字符编码成字节。它使用的字符集可以由名称指定或显式给定，否则将接受平台默认的字符集。

public class OutputStreamWriter extends Writer

构造函数：

FileWriter(File file)　根据给定的 File 对象构造一个 FileWriter 对象。

FileWriter(File file, boolean append)　根据给定的 File 对象构造一个 FileWriter 对象。

FileWriter(String fileName)　根据给定的文件名构造一个 FileWriter 对象。

FileWriter(String fileName, boolean append)　根据给定的文件名以及指示是否附加写入数据的 boolean 值来构造 FileWriter 对象。

成员方法：

FileWriter类没有定义相关成员方法，都是从OutputStreamWriter里面继承而来的方法；

void close()　关闭此流，但要先刷新它。

void flush()　刷新该流的缓冲。

String getEncoding()　返回此流使用的字符编码的名称。

void write(int c)　写入单个字符。

void write(String str)　写入字符串

void write(String str, int off, int len)　写入字符串的某一部分。

void write(char[] cbuf)　写入字符数组

void write(char[] cbuf, int off, int len)　写入字符数组的某一部分。

我们使用字符流,就是用来处理文字相关的文档；

6、内存流

为了提高数据读写的速度，Java API提供了带缓冲功能的流类，在使用这些流类时，会创建一个内部缓冲区数组，缺省使用8192个字节(8Kb)的缓冲区。

public class BufferedInputStream extends FilterInputStream {

```
        private static int DEFAULT_BUFFER_SIZE = 8192;
    }
```

缓冲流要“套接”在相应的节点流之上，不能独立存在，必须依托于另一个流（节点流）。

根据数据操作单位可以把缓冲流分为：

```
        BufferedInputStream 缓冲字节输入流；
BufferedOutputStream 缓冲字节输出流；

        BufferedReader  缓冲字符输入流；
        BufferedWriter  缓冲字符输出流；
```

写出数据时，先写入到其内部的缓冲区中，当缓冲区满了，会进行一次真实的写操作。

内存流读和写的原理

当读取数据时，数据按块读入缓冲区，其后的读操作则直接访问缓冲区。

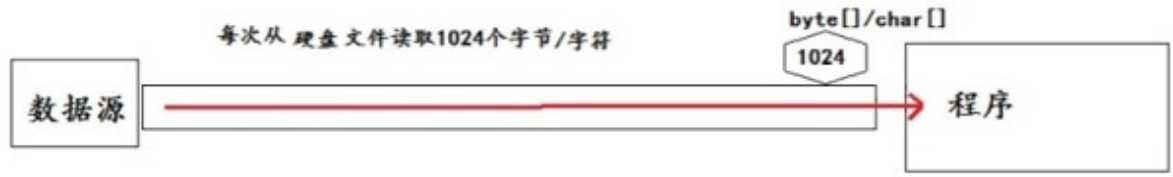
读取数据时，一次性尽可能多的读取字节，并缓存起来，这样实际上就是从缓冲区中读取数据，所以读取效率高。当缓冲区内容都读取完毕后，该流会再次尽可能多的读取字节缓存起来，等待再次被读取。

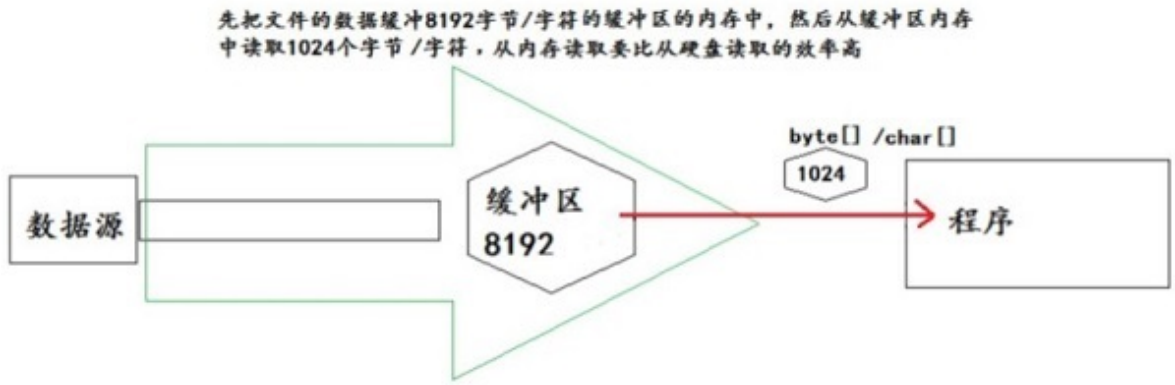
向流中写入字节时，不会直接写到文件，先写到缓冲区中直到缓冲区写满，才会把缓冲区中的数据一次性写到文件里。使用方法`flush()`可以强制将缓冲区的内容全部写入输出流。

关闭流的顺序和打开流的顺序相反。只要关闭最外层流即可，关闭最外层流也会相应关闭内层节点流。

`flush()`方法的使用：手动将`buffer`中内容写入文件。

如果是带缓冲区的流对象的`close()`方法，不但会关闭流，还会在关闭流之前刷新缓冲区，关闭后不能再写出。





内存流新增方法

```
    BufferedInputStream、BufferedOutputStream 没有新增的方法；

    BufferedReader新增方法：
        String readLine()   读取一个文本行。

    BufferedWriter新增方法：
        void newLine()  写入一个行分隔符。
```

示例代码

```
import java.io.*;

public class BufferedTest {
    public static void main(String[] args) {
        BufferedReader br = null;
        BufferedWriter bw = null;
        try {
            // 创建缓冲流对象：它是处理流，是对节点流的包装
            br = new BufferedReader(new FileReader("d:\\io\\source.txt"));
            bw = new BufferedWriter(new FileWriter("d:\\io\\dest.txt"));

            String str;
            while ((str = br.readLine()) != null) { // 一次读取字符文本文件的一行字符
                bw.write(str); // 一次写入一行字符串
                bw.newLine(); // 写入行分隔符
            }

            bw.flush(); // 刷新缓冲区
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            // 关闭IO流对象
            try {
                if (bw != null) {
                    bw.close(); // 关闭过滤流时,会自动关闭它所包装的底层节点流
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                if (br != null) {
                    br.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

7、过滤流

过滤流就是在节点流的基础上附加功能

处理类型	字符流	字节流
缓存	BufferedReader、BufferedWriter	BufferedInputStream、BuàeredOutputStream
过滤处理	FilterReader、FilterWriter	FilterInputStream、FilterOutputStream
桥接处理	InputStreamReader、OutputStreamWriter	
对象序列化处理		ObjectInputStream、ObjectOutputStream
数据转换		DataInputStream、DataOutputStream
行数统计	LineNumberReader	LineNumberInputStream
回滚处理	PushbackReader	PushbackInputStream
打印功能	PrintWriter	PrintStream

过滤流
FilterInputStream/FilterOutputStream和FilterReader/FilterWriter

```
 public class FilterInputStream extends InputStream { //典型的装饰模式
    protected volatile InputStream in; //被装饰目标
    protected FilterInputStream(InputStream in) { //通过构造器组装被装饰对象
        this.in = in;
    }
    public int read() throws IOException {//调用Filter中的read方法时实际操作是由被装饰对象实现 的
        return in.read();
    }
 }
```

所谓的过滤流实际上就是类似上面的加密处理，在输入之后（后置处理，被装饰对象先执行）或者输出之前（前置 处理，先处理然后被装饰对象执行）进行一下额外的处理，最终实际操作是调用被装饰对象的方法完成工作，依靠 这种装饰模式实现在节点流的基础上附加额外功能.当然也允许多个过滤流嵌套从而达到功能累加的目的
FilterInputStream实际上就是一个装饰抽象角色

自定义流实现循环加密:
读取数据不变:FileReader—BuàeredReader
写出数据自定义过滤流SecurityWriter(FilterWriter)

```
public class SecurityWriter extends FilterWriter {
    protected SecurityWriter(Writer out) {
        super(out);
    }
    public void write(int c) throws IOException {
        if (c >= 'a' && c <= 'z') {
            c = (c - 'a' + 13) % 26 + 'a';
        } else if (c >= 'A' && c <= 'Z') {
            c = (c - 'A' + 13) % 26 + 'A';
```

```
        }
        super.write(c);
    }
}

public class SecurityReader extends FilterReader {
    protected SecurityReader(Reader in) {
        super(in);
    }
    public int read() throws IOException {
        int c = super.read();
        if (c >= 'a' && c <= 'z') {
            c = (c - 'a' + 13) % 26 + 'a';
        } else if (c >= 'A' && c <= 'Z') {
            c = (c - 'A' + 13) % 26 + 'A';
        }
        return c;
    }
}
```

8、随机文件访问

RandomAccessFile 声明在java.io包下，但直接继承于java.lang.Object类。并且它实现了DataInput、DataOutput这两个接口，也就意味着这个类既可以读也可以写。

RandomAccessFile 类支持 “随机访问” 的方式，程序可以直接跳到文件的任意地方来读、写文件。
支持只访问文件的部分内容。
可以向已存在的文件后追加内容。

RandomAccessFile 对象包含一个记录指针，用以标示当前读写处的位置。

RandomAccessFile 类对象可以自由移动记录指针：
long getFilePointer()：获取文件记录指针的当前位置。
void seek(long pos)：将文件记录指针定位到 pos 位置。

构造器和访问模式：

构造器：
public RandomAccessFile(File file, String mode) 创建从中读取和向其中写入（可选）的随机访问文件流，该文件由 File 参数指定。

public RandomAccessFile(String name, String mode) 创建从中读取和向其中写入（可选）的随机访问文件流，该文件具有指定名称。

创建 RandomAccessFile 类实例需要指定一个 mode 参数，该参数指定 RandomAccessFile 的访问模式：
r: 以只读方式打开;
rw：打开以便读取和写入;
rwd:打开以便读取和写入；同步文件内容的更新;
rws:打开以便读取和写入；同步文件内容和元数据的更新;

如果模式为只读r。则不会创建文件，而是会去读取一个已经存在的文件，如果读取的文件不存在则会出现异常。 如果模式为rw读写。如果文件不存在则会去创建文件，如果存在则不会创建。

我们可以用RandomAccessFile这个类，来实现一个多线程断点下载的功能，用过下载工具的朋友们都知道，下载前都会建立两个临时文件，一个是与被下载文件大小相同的空文件，另一个是记录文件指针的位置文件，每次暂停的时候，都会保存上一次的指针，然后断点下载的时候，会继续从上一次的地方下载，从而实现断点下载或上传的功能，有兴趣的可以自己实现下。

实例代码：

```
import org.junit.Test;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;

/**
 * RandomAccessFile的使用
 * 1.RandomAccessFile直接继承于java.lang.Object类，实现了DataInput和DataOutput接口
 * 2.RandomAccessFile既可以作为一个输入流，又可以作为一个输出流
 *
 * 3.如果RandomAccessFile作为输出流时，写出到的文件如果不存在，则在执行过程中自动创建。
 *   如果写出到的文件存在，则会对原有文件内容进行覆盖。（默认情况下，从头覆盖）
 *
 * 4. 可以通过相关的操作，实现RandomAccessFile“插入”数据的效果
 */
public class RandomAccessFileTest {

    // RandomAccessFile实现文件读写
    @Test
    public void test1() {

        RandomAccessFile raf1 = null;
        RandomAccessFile raf2 = null;
        try {
            //1.
            raf1 = new RandomAccessFile(new File("爱情与友情.jpg"),"r");
            raf2 = new RandomAccessFile(new File("爱情与友情1.jpg"),"rw");
            //2.
            byte[] buffer = new byte[1024];
            int len;
            while((len = raf1.read(buffer)) != -1){
                raf2.write(buffer,0,len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //3.
            if(raf1 != null){
                try {
                    raf1.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }

            }
            if(raf2 != null){
                try {
                    raf2.close();
```

```
                } catch (IOException e) {
                    e.printStackTrace();
                }

            }
        }
    }

    // RandomAccessFile写入操作
    @Test
    public void test2() throws IOException {

        RandomAccessFile raf1 = new RandomAccessFile("hello.txt","rw");

        raf1.seek(3);//将指针调到角标为3的位置
        raf1.write("xyz".getBytes());//

        raf1.close();

    }
    /*
    使用RandomAccessFile实现数据的插入效果
     */
    @Test
    public void test3() throws IOException {

        RandomAccessFile raf1 = new RandomAccessFile("hello.txt","rw");

        raf1.seek(3);//将指针调到角标为3的位置
        //保存指针3后面的所有数据到StringBuilder中
        StringBuilder builder = new StringBuilder((int) new
File("hello.txt").length());
        byte[] buffer = new byte[20];
        int len;
        while((len = raf1.read(buffer)) != -1){
            builder.append(new String(buffer,0,len)) ;
        }
        //调回指针，写入“xyz”
        raf1.seek(3);
        raf1.write("xyz".getBytes());

        //将StringBuilder中的数据写入到文件中
        raf1.write(builder.toString().getBytes());

        raf1.close();
    }
}
```

9、序列化

ObjectInputStream和ObjectOutputSteam,用于存储和读取基本数据类型数据或对象的处理流。它的强大之处就是可以把Java中的对象写入到数据源中，也能把对象从数据源中还原回来。

序列化：用ObjectOutputStream类保存基本类型数据或对象的机制。

反序列化：用ObjectInputStream类读取基本类型数据或对象的机制。

ObjectOutputStream和ObjectInputStream不能序列化static和transient修饰的成员变量。

序列化概述

对象序列化机制允许把内存中的Java对象转换成平台无关的二进制流，从而允许把这种二进制流持久地保存在磁盘上，或通过网络将这种二进制流传输到另一个网络节点。当其它程序获取了这种二进制流，就可以恢复成原来的Java对象。

1、序列化：指允许把堆内存中的Java对象数据转换成平台无关的二进制流，从而允许把这种二进制流持久地存储到磁盘文件中，或通过网络将这种二进制流传输到另一个网络节点（网络传输）。这个过程称为序列化，通常是指将数据结构或对象转化成字节序列的过程。该字节序列包括该对象的数据、有关对象的类型的信息和存储在对象中数据的类型。

即将对象转化为二进制，用于保存，或者网络传输。

2、反序列化：把磁盘文件中的对象数据或者把网络节点上的对象数据，恢复成Java对象模型的过程。也就是将在序列化过程中所生成的字节序列转换成数据结构或者对象的过程。

与序列化相反，将二进制转化成对象。

序列化的好处在于可将任何实现了Serializable接口的对象转化为字节数据，使其在保存和传输时可被还原。

序列化是 RMI（Remote Method Invoke – 远程方法调用）过程的参数和返回值都必须实现的机制，而 RMI 是 JavaEE 的基础。因此序列化机制是JavaEE 平台的基础。

如果需要让某个对象支持序列化机制，则必须让对象所属的类及其属性是可序列化的，为了让某个类是可序列化的，该类必须实现如下两个接口之一。否则，会抛出NotSerializableException异常。

Serializable：类通过实现 java.io.Serializable 接口以启用其序列化功能（推荐）。

Externalizable：Externalizable 实例类的唯一特性是可以被写入序列化流中，该类负责保存和恢复实例内容。

通常建议：程序创建的每个JavaBean类都实现Serializeable接口。

Serializable接口面试题：

谈谈你对java.io.Serializable接口的理解，我们知道它用于序列化，是空方法接口，还有其它认识吗？

实现了Serializable接口的对象，可将它们转换成一系列字节，并可在以后完全恢复回原来的样子。这一过程亦可通过网络进行。这意味着序列化机制能自动补偿操作系统间的差异。换句话说，可以先在Windows机器上创建一个对象，对其序列化，然后通过网络发给一台Unix机器，然后在那里准确无误地重新“装配”。不必关心数据在不同机器上如何表示，也不必关心字节的顺序或者其他任何细节。

由于大部分作为参数的类如String、Integer等都实现了java.io.Serializable的接口，也可以利用多态的性质，作为参数使接口更灵活。

序列化实现的方式

如果需要将某个对象保存到磁盘上或者通过网络传输，那么这个类应该实现Serializable接口；
Serializable接口是一个标记接口，不用实现任何方法。一旦实现了此接口，该类的对象就是可序列化的。

凡是实现Serializable接口的类都有一个表示序列化版本标识符的静态变量：
private static final long serialVersionUID;
serialVersionUID用来表明类的不同版本间的兼容性。简言之，其目的是以序列化对象进行版本控制，有关各版本反序列化时是否兼容。
如果类没有显示定义这个静态常量，它的值是Java运行时环境根据类的内部细节自动生成的。若类的实例变量做了修改，serialVersionUID 可能发生变化。故建议，显式声明。

简单来说，Java的序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性的。在进行反序列化时，JVM会把传来的字节流中的serialVersionUID与本地相应实体类的serialVersionUID进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常。(InvalidCastException)

在Java的OutputStream类下面的子类 ObjectOutputStream 类就有对应的 writeObject(Object object) 其中要求对应的object实现了java的序列化的接口。

在使用tomcat开发JavaEE相关项目的时候，我们关闭tomcat后，相应的session中的对象就存储在了硬盘上，如果我们想要在tomcat重启的时候能够从tomcat上面读取对应session中的内容，那么保存在session中的内容就必须实现相关的序列化操作，还有jdbc加载驱动用的就是反序列化，将字符串变为对象；

普通序列化

若某个类实现了 Serializable 接口，该类的对象就是可序列化的：

序列化步骤：
步骤一：创建一个ObjectOutputStream输出流；
步骤二：调用ObjectOutputStream对象的writeObject(对象)方法输出可序列化对象。
注意写出一次，操作flush()一次。

```
import org.junit.Test;

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

public class WriteObject {

    /*
     * 使用ObjectOutputStream实现对象序列化
     *  1、序列号字符串对象
     *  2、序列号自定义对象
     */
    @Test
    public void testObjectOutputStream01() throws IOException {
        // 创建一个ObjectOutputStream输出流
        ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("object.dat"));
        // 将字符串对象序列化到文件
        oos.writeObject(new String("我爱北京天安门"));
        oos.flush();//刷新操作

        // 将自定义对象序列化到文件：如果Person没有实现Serializable接口，则抛出
java.io.NotSerializableException异常
```

```
        Person person = new Person("张三", 23);
        oos.writeObject(person);
        oos.flush();

        oos.close();
    }
}


import java.io.Serializable;

/**
 * Person需要满足如下的要求，方可序列化
 * 1.需要实现接口：Serializable
 * 2.当前类提供一个全局常量：serialVersionUID - 后面演示
 * 3.除了当前Person类需要实现Serializable接口之外，还必须保证其内部所有属性
 * 也必须是可序列化的。（默认情况下，基本数据类型可序列化）
 *
 * 补充：ObjectOutputStream和ObjectInputStream不能序列化static和transient修饰的成员变量
 */
public class Person implements Serializable {
    private String name;
    private int age;

    public Person() {
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

反序列化

反序列化步骤：
    步骤一：创建一个ObjectInputStream输入流；
    步骤二：调用ObjectInputStream对象的readObject()得到序列化的对象。

强调：如果某个类的属性不是基本数据类型或 String 类型，而是另一个引用类型，那么这个引用类型必须是可序列化的，否则拥有该类型的Field 的类也不能序列化。

我们将上面序列化到object.dat的字符串和person对象反序列化回来。

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class ReadObject {
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        //创建一个ObjectInputStream输入流
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("object.dat"));

        //反序列化字符串对象
        Object obj = ois.readObject();
        String str = (String) obj;

        //反序列化Person对象
        Person p = (Person) ois.readObject();

        System.out.println(str); // 我爱北京天安门
        System.out.println(p);// Person{name='张三', age=23}

        ois.close();
    }
}

import java.io.Serializable;

public class Person implements Serializable {
    private String name;
    private int age;

    public Person() {
        System.out.println("反序列化,你调用我了吗? -- 无参构造函数");
    }

    public Person(String name, int age) {
        System.out.println("反序列化,你调用我了吗? -- 有参构造函数");
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
```

```
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

10、新IO

一、什么是NIO

- Java NIO全称java non-blocking IO， 是指JDK提供的新API。从JDK1.4开始，Java提供了一系列改进的输入/输出的新特性，被统称为NIO(即New IO)，是同步非阻塞的
- NIO有三大核心部分: **Channel(通道)， Buffer(缓冲区),Selector(选择器)**
- NIO是面向缓冲区，或者面向块编程的。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动，这就增加了处理过程中的灵活性，使用它可以提供非阻塞式的高伸缩性网络。

二、NIO 与BIO 模型对比

> BIO 是同步阻塞IO,服务器的模式是一个线程处理一个请求，当无响应时，会阻塞线程

> NIO 同步非阻塞IO,会有一个Selector管理多个线程，当有事件发生后，进行处理、不会发生阻塞

三、NIO 与BIO的差异

1、BIO 以流的方式处理数据,而NIO以块的方式处理数据,块I/O 的效率比流I/O高很多
2、BIO 是阻塞的，NIO则是非阻塞的
3、 BIO基 于字节流和字符流进行操作，而NIO 基于Channel(通道)和Buffer(缓冲区)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector(选择器)用于监听多个通道的事件(比如:连接请求，数据到达等)，因此使用单个线程就可以监听多个客户端通道

四、相关API

NIO的API位于java.nio及其子包下。下面是常用的Java新IO中的接口或类：

Path接口：

Path接口代表一条路径（既可是文件的路径，也可是目录的路径）。

Paths类：

Paths是一个工具类，负责把String包装为Path。
一句话：Path接口代表一条路径；Paths是Path的工具类。

由于Path接口没有实现类，故需要通过Paths类的get方法来获得一个Path对象。

Files类：

Files是做文件IO的工具类。基本上文件复制、移动、删除、读取、写入、重命名、隐藏、创建、创建快捷方式……等都只要一个方法即可，使用起来非常简单方便。

下面的小例子测试了Files工具类的新建文件和复制文件的功能：

```
import java.io.IOException;
import java.nio.file.*;

public class FilesTest
{
    public static void main(String[] args) throws IOException
    {
        if (!Files.exists(Paths.get("filesTest.txt")))
        {
            //新建一个文件
            //createFile (Path path, FileAttribute <?> ... attrs):
            //在当前路径下创建一个新的空文件，如果文件已经存在则失败。
            Files.createFile(Paths.get("filesTest.txt"));
        }

        //新建一个临时文件
        //createTempFile(Path dir, String prefix, String suffix, FileAttribute<?
>... attrs):
        //在指定目录中创建一个新的空文件(临时文件)，使用给定的前缀和后缀字符串生成其名称。
        Files.createTempFile(Paths.get("./filesTest"), "tempFile_", ".txt");


        if (!Files.exists(Paths.get("test1_copy.txt")))
        {
            //复制一个文件
            Files.copy(Paths.get("test1.txt"), Paths.get("test1_copy.txt"));
        }
    }
}
```

下面的小例子测试了Files工具类的读取文件的功能：

```
import java.io.IOException;
import java.nio.file.*;

public class FilesReadTest
{
    public static void main(String[] args) throws IOException
    {
        //readAllLines(Path path): 从一个文件中读取所有行。(返回一个List<String>)
        Files.readAllLines(Paths.get("FilesReadTest.java")).
            forEach(System.out::println);
    }
}
```

下面的小例子测试了Files工具类的写文件的功能：

```
import java.io.IOException;
import java.nio.file.*;

public class FilesWriteTest
{
    public static void main(String[] args) throws IOException,
NoSuchFileException
    {

        //write(Path path, byte[] bytes, OpenOption... options):
        //写bytes到一个文件。 (默认情况下，该方法会创建一个新文件或覆盖现有文件)
        Files.write(Paths.get("filesWriteTest.txt"), "Hello
World!\n".getBytes());

        //StandardOpenOption.APPEND:追加内容进一个已经存在的文件
        Files.write(Paths.get("filesWriteTest.txt"),
            "This is the content appended!\n".getBytes(),
StandardOpenOption.APPEND);

    }
}
```