File类、IO流

Part1： File类&递归

- [ ] 能够说出File对象的创建方式
- [ ] 能够说出File类获取名称的方法名称
- [ ] 能够说出File类获取绝对路径的方法名称
- [ ] 能够说出File类获取文件大小的方法名称
- [ ] 能够说出File类判断是否是文件的方法名称
- [ ] 能够说出File类判断是否是文件夹的方法名称
- [ ] 能够辨别相对路径和绝对路径
- [ ] 能够遍历文件夹
- [ ] 能够解释递归的含义
- [ ] 能够使用递归的方式计算5的阶乘
- [ ] 能够说出使用递归会内存溢出隐患的原因

第一章 File类

1.1 概述

`java.io.File` 类是文件和目录路径名的抽象表示，主要用于文件和目录的创建、查找和删除等操作。

1.2 构造方法

- `public File(String pathname)` ：通过将给定的**路径名字符串**转换为抽象路径名来创建新的File实例。
- `public File(String parent, String child)` ：从**父路径名字符串和子路径名字符串**创建新的 File实例。
- `public File(File parent, String child)` ：从**父抽象路径名和子路径名字符串**创建新的File实例。
- 构造举例，代码如下：

```
// 文件路径名
String pathname = "D:\\aaa.txt";
File file1 = new File(pathname);

// 文件路径名
String pathname2 = "D:\\aaa\\bbb.txt";
File file2 = new File(pathname2);
```

```
// 通过父路径和子路径字符串
 String parent = "d:\\aaa";
 String child = "bbb.txt";
 File file3 = new File(parent, child);

// 通过父级File对象和子路径字符串
File parentDir = new File("d:\\aaa");
String child = "bbb.txt";
File file4 = new File(parentDir, child);
```

> 小贴士：
>
> 1. 一个File对象代表硬盘中实际存在的一个文件或者目录。
> 2. 无论该路径下是否存在文件或者目录，都不影响File对象的创建。

1.3 常用方法

获取功能的方法

- `public String getAbsolutePath()` ：返回此File的绝对路径名字符串。
- `public String getPath()` ：将此File转换为路径名字符串。
- `public String getName()` ：返回由此File表示的文件或目录的名称。
- `public long length()` ：返回由此File表示的文件的长度。

  方法演示，代码如下：

```
public class FileGet {
    public static void main(String[] args) {
        File f = new File("d:/aaa/bbb.java");
        System.out.println("文件绝对路径:"+f.getAbsolutePath());
        System.out.println("文件构造路径:"+f.getPath());
        System.out.println("文件名称:"+f.getName());
        System.out.println("文件长度:"+f.length()+"字节");

        File f2 = new File("d:/aaa");
        System.out.println("目录绝对路径:"+f2.getAbsolutePath());
        System.out.println("目录构造路径:"+f2.getPath());
        System.out.println("目录名称:"+f2.getName());
        System.out.println("目录长度:"+f2.length());
    }
}
输出结果：
文件绝对路径:d:\aaa\bbb.java
文件构造路径:d:\aaa\bbb.java
文件名称:bbb.java
文件长度:636字节

目录绝对路径:d:\aaa
```

```
目录构造路径:d:\aaa
目录名称:aaa
目录长度:4096
```

> API中说明：length()，表示文件的长度。但是File对象表示目录，则返回值未指定。

绝对路径和相对路径

- **绝对路径**：从盘符开始的路径，这是一个完整的路径。
- **相对路径**：相对于项目目录的路径，这是一个便捷的路径，开发中经常使用。

```
public class FilePath {
    public static void main(String[] args) {
        // D盘下的bbb.java文件
        File f = new File("D:\\bbb.java");
        System.out.println(f.getAbsolutePath());

    // 项目下的bbb.java文件
        File f2 = new File("bbb.java");
        System.out.println(f2.getAbsolutePath());
    }
}
输出结果:
D:\bbb.java
D:\idea_project_test4\bbb.java
```

判断功能的方法

- `public boolean exists()` ： 此File表示的文件或目录是否实际存在。
- `public boolean isDirectory()` ： 此File表示的是否为目录。
- `public boolean isFile()` ： 此File表示的是否为文件。

方法演示，代码如下：

```
public class FileIs {
    public static void main(String[] args) {
        File f = new File("d:\\aaa\\bbb.java");
        File f2 = new File("d:\\aaa");
        // 判断是否存在
        System.out.println("d:\\aaa\\bbb.java 是否存在:"+f.exists());
        System.out.println("d:\\aaa 是否存在:"+f2.exists());
        // 判断是文件还是目录
        System.out.println("d:\\aaa 文件?:"+f2.isFile());
        System.out.println("d:\\aaa 目录?:"+f2.isDirectory());
    }
}
输出结果:
d:\aaa\bbb.java 是否存在:true
d:\aaa 是否存在:true
```

```
d:\aaa 文件?:false
d:\aaa 目录?:true
```

创建删除功能的方法

- `public boolean createNewFile()`：当且仅当具有该名称的文件尚不存在时，创建一个新的空文件。
- `public boolean delete()`：删除由此File表示的文件或目录。
- `public boolean mkdir()`：创建由此File表示的目录。
- `public boolean mkdirs()`：创建由此File表示的目录，包括任何必需但不存在的父目录。

方法演示，代码如下：

```
public class FileCreateDelete {
    public static void main(String[] args) throws IOException {
        // 文件的创建
        File f = new File("aaa.txt");
        System.out.println("是否存在:"+f.exists()); // false
        System.out.println("是否创建:"+f.createNewFile()); // true
        System.out.println("是否存在:"+f.exists()); // true

      // 目录的创建
        File f2= new File("newDir");
        System.out.println("是否存在:"+f2.exists());// false
        System.out.println("是否创建:"+f2.mkdir()); // true
        System.out.println("是否存在:"+f2.exists());// true

    // 创建多级目录
        File f3= new File("newDira\\newDirb");
        System.out.println(f3.mkdir());// false
        File f4= new File("newDira\\newDirb");
        System.out.println(f4.mkdirs());// true

        // 文件的删除
        System.out.println(f.delete());// true

        // 目录的删除
        System.out.println(f2.delete());// true
        System.out.println(f4.delete());// false
    }
}
```

> API中说明：delete方法，如果此File表示目录，则目录必须为空才能删除。

1.4 目录的遍历

- `public String[] list()`：返回一个String数组，表示该File目录中的所有子文件或目录。
- `public File[] listFiles()`：返回一个File数组，表示该File目录中的所有的子文件或目录。

```
public class FileFor {
    public static void main(String[] args) {
        File dir = new File("d:\\java_code");

        //获取当前目录下的文件以及文件夹的名称。
    String[] names = dir.list();
    for(String name : names){
      System.out.println(name);
    }
        //获取当前目录下的文件以及文件夹对象，只要拿到了文件对象，那么就可以获取更多信息
        File[] files = dir.listFiles();
        for (File file : files) {
            System.out.println(file);
        }
    }
}
```

> 小贴士：
>
> 调用listFiles方法的File对象，表示的必须是实际存在的目录，否则返回null，无法进行遍历。

第二章 递归

2.1 概述

- **递归**：指在当前方法内调用自己的这种现象。
- **递归的分类:**
  - 递归分为两种，直接递归和间接递归。
  - 直接递归称为方法自身调用自己。
  - 间接递归可以A方法调用B方法，B方法调用C方法，C方法调用A方法。
- **注意事项**：
  - 递归一定要有条件限定，保证递归能够停止下来，否则会发生栈内存溢出。
  - 在递归中虽然有限定条件，但是递归次数不能太多。否则也会发生栈内存溢出。
  - 构造方法,禁止递归

```
public class Demo01DiGui {
  public static void main(String[] args) {
    // a();
    b(1);
  }

  /*
   * 3.构造方法,禁止递归
   * 编译报错:构造方法是创建对象使用的,不能让对象一直创建下去
   */
  public Demo01DiGui() {
    //Demo01DiGui();
```

```
  }


  /*
   * 2.在递归中虽然有限定条件，但是递归次数不能太多。否则也会发生栈内存溢出。
   * 4993
   *  Exception in thread "main" java.lang.StackOverflowError
   */
  private static void b(int i) {
    System.out.println(i);
    //添加一个递归结束的条件,i==5000的时候结束
    if(i==5000){
      return;//结束方法
    }
    b(++i);
  }

  /*
   * 1.递归一定要有条件限定，保证递归能够停止下来，否则会发生栈内存溢出。 Exception in
thread "main"
   * java.lang.StackOverflowError
   */
  private static void a() {
    System.out.println("a方法");
    a();
  }
}
```

2.2 递归累加求和

计算1 ~ n的和

分析：num的累和 = num + (num-1)的累和，所以可以把累和的操作定义成一个方法，递归调用。

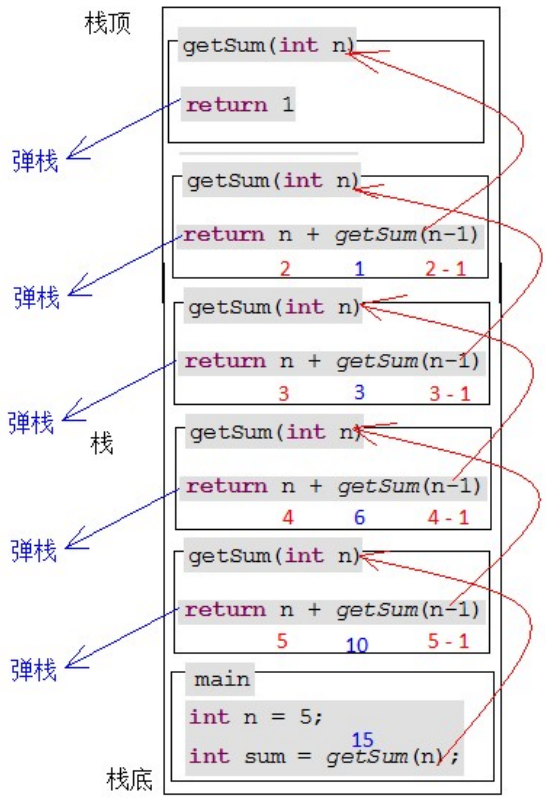
实现代码：

```
public class DiGuiDemo {
  public static void main(String[] args) {
    //计算1~num的和，使用递归完成
    int num = 5;
        // 调用求和的方法
    int sum = getSum(num);
        // 输出结果
    System.out.println(sum);

  }
    /*
      通过递归算法实现.
      参数列表:int
```

```
        返回值类型：int
    */
  public static int getSum(int num) {
        /*
            num为1时,方法返回1,
            相当于是方法的出口,num总有是1的情况
        */
    if(num == 1){
      return 1;
    }
        /*
          num不为1时,方法返回 num +(num-1)的累和
          递归调用getSum方法
        */
    return num + getSum(num-1);
  }
}
```

代码执行图解

```
public class DiGuiDemo {
    public static void main(String[] args) {
        //计算1~num的和，使用递归完成
        int n = 5;
        int sum = getSum(n);
        System.out.println(sum);

    }
    public static int getSum(int n) {
        if(n == 1){
            return 1;
        }
        return n + getSum(n-1);
    }
}
```



> 小贴士：递归一定要有条件限定，保证递归能够停止下来，次数不要太多，否则会发生栈内存溢出。

2.3 递归求阶乘

- **阶乘**：所有小于及等于该数的正整数的积。

```
n的阶乘：n! = n * (n-1) *...* 3 * 2 * 1
```

分析：这与累和类似,只不过换成了乘法运算，学员可以自己练习，需要注意阶乘值符合int类型的范围。

```
推理得出：n! = n * (n-1)!
```

代码实现：

```
public class DiGuiDemo {
    //计算n的阶乘，使用递归完成
    public static void main(String[] args) {
        int n = 3;
        // 调用求阶乘的方法
        int value = getValue(n);
        // 输出结果
        System.out.println("阶乘为:"+ value);
    }
  /*
     通过递归算法实现.
     参数列表:int
     返回值类型: int
    */
    public static int getValue(int n) {
        // 1的阶乘为1
        if (n == 1) {
            return 1;
        }
        /*
          n不为1时,方法返回 n! = n*(n-1)!
          递归调用getValue方法
        */
        return n * getValue(n - 1);
    }
}
```

2.4 递归打印多级目录

分析：多级目录的打印，就是当目录的嵌套。遍历之前，无从知道到底有多少级目录，所以我们还是要使用递归实现。

代码实现：

```
public class DiGuiDemo2 {
    public static void main(String[] args) {
        // 创建File对象
        File dir  = new File("D:\\aaa");
        // 调用打印目录方法
        printDir(dir);
    }
```

```
    public static void  printDir(File dir) {
        // 获取子文件和目录
        File[] files = dir.listFiles();
        // 循环打印
        /*
          判断:
          当是文件时,打印绝对路径.
          当是目录时,继续调用打印目录的方法,形成递归调用.
        */
        for (File file : files) {
        // 判断
            if (file.isFile()) {
                // 是文件,输出文件绝对路径
                System.out.println("文件名:"+ file.getAbsolutePath());
            } else {
                // 是目录,输出目录绝对路径
                System.out.println("目录:"+file.getAbsolutePath());
                // 继续遍历,调用printDir,形成递归
                printDir(file);
            }
        }
    }
}
```

第三章 综合案例

3.1 文件搜索

搜索`D:\aaa`目录中的`.java`文件。

分析:

1. 目录搜索，无法判断多少级目录，所以使用递归，遍历所有目录。
2. 遍历目录时，获取的子文件，通过文件名称，判断是否符合条件。

代码实现:

```
public class DiGuiDemo3 {
    public static void main(String[] args) {
        // 创建File对象
        File dir  = new File("D:\\aaa");
        // 调用打印目录方法
        printDir(dir);
    }

    public static void printDir(File dir) {
        // 获取子文件和目录
        File[] files = dir.listFiles();
```

```
        // 循环打印
        for (File file : files) {
            if (file.isFile()) {
                // 是文件，判断文件名并输出文件绝对路径
                if (file.getName().endsWith(".java")) {
                    System.out.println("文件名:" + file.getAbsolutePath());
                }
            } else {
                // 是目录，继续遍历,形成递归
                printDir(file);
            }
        }
    }
}
```

3.2 文件过滤器优化

`java.io.FileFilter`是一个接口，是File的过滤器。 该接口的对象可以传递给File类的`listFiles(FileFilter)` 作为参数， 接口中只有一个方法。

`boolean accept(File pathname)`：测试pathname是否应该包含在当前File目录中，符合则返回true。

分析：

1. 接口作为参数，需要传递子类对象，重写其中方法。我们选择匿名内部类方式，比较简单。
2. `accept`方法，参数为File，表示当前File下所有的子文件和子目录。保留住则返回true，过滤掉则返回false。保留规则：
    1. 要么是.java文件。
    2. 要么是目录，用于继续遍历。
3. 通过过滤器的作用，`listFiles(FileFilter)`返回的数组元素中，子文件对象都是符合条件的，可以直接打印。

代码实现：

```
public class DiGuiDemo4 {
    public static void main(String[] args) {
        File dir = new File("D:\\aaa");
        printDir2(dir);
    }

    public static void printDir2(File dir) {
        // 匿名内部类方式,创建过滤器子类对象
        File[] files = dir.listFiles(new FileFilter() {
            @Override
            public boolean accept(File pathname) {
                return
pathname.getName().endsWith(".java")||pathname.isDirectory();
```

```
            }
        });
        // 循环打印
        for (File file : files) {
            if (file.isFile()) {
                System.out.println("文件名:" + file.getAbsolutePath());
            } else {
                printDir2(file);
            }
        }
    }
}
```

3.3 Lambda优化

分析： `FileFilter`是只有一个方法的接口，因此可以用lambda表达式简写。

lambda格式：

```
()->{ }
```

代码实现：

```
public static void printDir3(File dir) {
    // lambda的改写
    File[] files = dir.listFiles(f ->{
        return f.getName().endsWith(".java") || f.isDirectory();
    });

  // 循环打印
    for (File file : files) {
        if (file.isFile()) {
            System.out.println("文件名:" + file.getAbsolutePath());
        } else {
          printDir3(file);
        }
    }
}
```

Part2：字节流、字符流

- [ ] 能够说出IO流的分类和功能
- [ ] 能够使用字节输出流写出数据到文件
- [ ] 能够使用字节输入流读取数据到程序

- [ ] 能够理解读取数据read(byte[])方法的原理
- [ ] 能够使用字节流完成文件的复制
- [ ] 能够使用FileWirter写数据到文件
- [ ] 能够说出FileWriter中关闭和刷新方法的区别
- [ ] 能够使用FileWriter写数据的5个方法
- [ ] 能够使用FileWriter写数据实现换行和追加写
- [ ] 能够使用FileReader读数据
- [ ] 能够使用FileReader读数据一次一个字符数组
- [ ] 能够使用Properties的load方法加载文件中配置信息

第四章 IO概述

4.1 什么是IO

生活中，你肯定经历过这样的场景。当你编辑一个文本文件，忘记了`ctrl+s`，可能文件就白白编辑了。当你电脑上插入一个U盘，可以把一个视频，拷贝到你的电脑硬盘里。那么数据都是在哪些设备上的呢？键盘、内存、硬盘、外接设备等等。

我们把这种数据的传输，可以看做是一种数据的流动，按照流动的方向，以内存为基准，分为`输入input`和`输出output`，即流向内存是输入流，流出内存的输出流。

Java中I/O操作主要是指使用`java.io`包下的内容，进行输入、输出操作。**输入**也叫做**读取**数据，**输出**也叫做作**写出**数据。

4.2 IO的分类

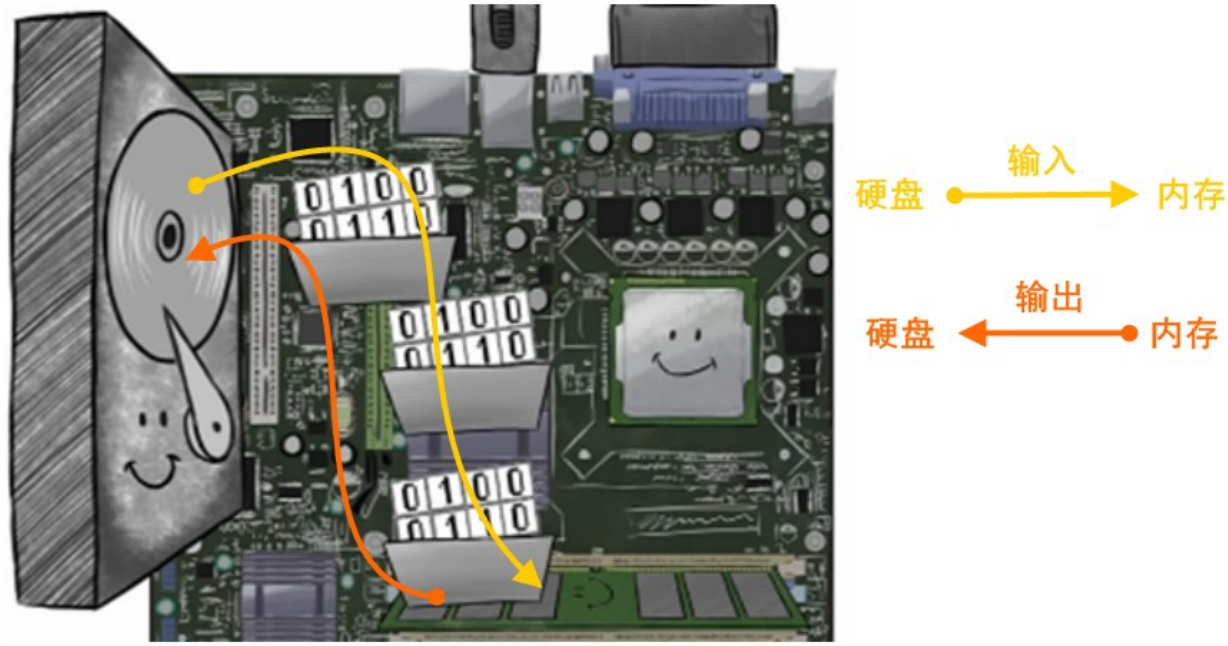
根据数据的流向分为：**输入流**和**输出流**。

- **输入流**：把数据从`其他设备`上读取到`内存`中的流。
- **输出流**：把数据从`内存`中写出到`其他设备`上的流。

格局数据的类型分为：**字节流**和**字符流**。

- **字节流**：以字节为单位，读写数据的流。
- **字符流**：以字符为单位，读写数据的流。

4.3 IO的流向说明图解



4.4 顶级父类们

	输入流	输出流
字节流	字节输入流 **InputStream**	字节输出流 **OutputStream**
字符流	字符输入流 **Reader**	字符输出流 **Writer**

第五章 字节流

5.1 一切皆为字节

一切文件数据(文本、图片、视频等)在存储时，都是以二进制数字的形式保存，都一个一个的字节，那么传输时一样如此。所以，字节流可以传输任意文件数据。在操作流的时候，我们要时刻明确，无论使用什么样的流对象，底层传输的始终为二进制数据。

5.2 字节输出流【OutputStream】

`java.io.OutputStream`抽象类是表示字节输出流的所有类的超类，将指定的字节信息写出到目的地。它定义了字节输出流的基本共性功能方法。

- `public void close()` ： 关闭此输出流并释放与此流相关联的任何系统资源。
- `public void flush()` ： 刷新此输出流并强制任何缓冲的输出字节被写出。
- `public void write(byte[] b)`： 将 b.length字节从指定的字节数组写入此输出流。
- `public void write(byte[] b, int off, int len)` ： 从指定的字节数组写入 len字节，从偏移量 off开始输出到此输出流。
- `public abstract void write(int b)` ： 将指定的字节输出流。

> 小贴士：
>
> close方法，当完成流的操作时，必须调用此方法，释放系统资源。

5.3 FileOutputStream类

`OutputStream`有很多子类，我们从最简单的一个子类开始。

`java.io.FileOutputStream`类是文件输出流，用于将数据写出到文件。

构造方法

- `public FileOutputStream(File file)`：创建文件输出流以写入由指定的 File对象表示的文件。
- `public FileOutputStream(String name)`： 创建文件输出流以指定的名称写入文件。

当你创建一个流对象时，必须传入一个文件路径。该路径下，如果没有这个文件，会创建该文件。如果有这个文件，会清空这个文件的数据。

- 构造举例，代码如下：

```
public class FileOutputStreamConstructor throws IOException {
    public static void main(String[] args) {
      // 使用File对象创建流对象
        File file = new File("a.txt");
        FileOutputStream fos = new FileOutputStream(file);

        // 使用文件名称创建流对象
        FileOutputStream fos = new FileOutputStream("b.txt");
    }
}
```

写出字节数据

1. **写出字节**：`write(int b)` 方法，每次可以写出一个字节数据，代码使用演示：

```
public class FOSWrite {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileOutputStream fos = new FileOutputStream("fos.txt");
        // 写出数据
        fos.write(97); // 写出第1个字节
        fos.write(98); // 写出第2个字节
        fos.write(99); // 写出第3个字节
        // 关闭资源
        fos.close();
    }
}
```

输出结果：

```
abc
```

> 小贴士：
>
> 1. 虽然参数为int类型四个字节，但是只会保留一个字节的信息写出。
> 2. 流操作完毕后，必须释放系统资源，调用close方法，千万记得。

2. **写出字节数组**：`write(byte[] b)`，每次可以写出数组中的数据，代码使用演示：

```
public class FOSWrite {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileOutputStream fos = new FileOutputStream("fos.txt");
        // 字符串转换为字节数组
        byte[] b = "黑马程序员".getBytes();
        // 写出字节数组数据
        fos.write(b);
        // 关闭资源
        fos.close();
    }
}
输出结果：
黑马程序员
```

3. **写出指定长度字节数组**：`write(byte[] b, int off, int len)` ,每次写出从off索引开始，len个字节，代码使用演示：

```
public class FOSWrite {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileOutputStream fos = new FileOutputStream("fos.txt");
        // 字符串转换为字节数组
        byte[] b = "abcde".getBytes();
    // 写出从索引2开始，2个字节。索引2是c，两个字节，也就是cd。
        fos.write(b,2,2);
        // 关闭资源
        fos.close();
    }
}
输出结果：
cd
```

数据追加续写

经过以上的演示，每次程序运行，创建输出流对象，都会清空目标文件中的数据。如何保留目标文件中数据，还能继续添加新数据呢？

- `public FileOutputStream(File file, boolean append)`： 创建文件输出流以写入由指定的 File对象表示的文件。
- `public FileOutputStream(String name, boolean append)`： 创建文件输出流以指定的名称写入文件。

这两个构造方法，参数中都需要传入一个boolean类型的值，`true` 表示追加数据，`false` 表示清空原有数据。这样创建的输出流对象，就可以指定是否追加续写了，代码使用演示：

```
public class FOSWrite {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileOutputStream fos = new FileOutputStream("fos.txt", true);
        // 字符串转换为字节数组
        byte[] b = "abcde".getBytes();
    // 写出从索引2开始，2个字节。索引2是c，两个字节，也就是cd。
        fos.write(b);
        // 关闭资源
        fos.close();
    }
}
文件操作前：cd
文件操作后：cdabcde
```

写出换行

Windows系统里，换行符号是`\r\n` 。把

以指定是否追加续写了，代码使用演示：

```
public class FOSWrite {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileOutputStream fos = new FileOutputStream("fos.txt");
        // 定义字节数组
        byte[] words = {97,98,99,100,101};
        // 遍历数组
        for (int i = 0; i < words.length; i++) {
            // 写出一个字节
            fos.write(words[i]);
            // 写出一个换行, 换行符号转成数组写出
            fos.write("\r\n".getBytes());
        }
        // 关闭资源
        fos.close();
    }
}

输出结果：
a
b
c
d
e
```

> - 回车符`\r`和换行符`\n`：
>   - 回车符：回到一行的开头（return）。
>   - 换行符：下一行（newline）。
> - 系统中的换行：
>   - Windows系统里，每行结尾是`回车+换行`，即`\r\n`；
>   - Unix系统里，每行结尾只有`换行`，即`\n`；
>   - Mac系统里，每行结尾是`回车`，即`\r`。从 Mac OS X开始与Linux统一。

5.4 字节输入流【InputStream】

`java.io.InputStream`抽象类是表示字节输入流的所有类的超类，可以读取字节信息到内存中。它定义了字节输入流的基本共性功能方法。

- `public void close()`：关闭此输入流并释放与此流相关联的任何系统资源。
- `public abstract int read()`： 从输入流读取数据的下一个字节。
- `public int read(byte[] b)`： 从输入流中读取一些字节数，并将它们存储到字节数组 b中 。

> 小贴士：
>
> close方法，当完成流的操作时，必须调用此方法，释放系统资源。

5.5 FileInputStream类

`java.io.FileInputStream`类是文件输入流，从文件中读取字节。

构造方法

- `FileInputStream(File file)`： 通过打开与实际文件的连接来创建一个 FileInputStream ，该文件由文件系统中的 File对象 file命名。
- `FileInputStream(String name)`： 通过打开与实际文件的连接来创建一个 FileInputStream ，该文件由文件系统中的路径名 name命名。

当你创建一个流对象时，必须传入一个文件路径。该路径下，如果没有该文件,会抛出`FileNotFoundException` 。

- 构造举例，代码如下：

```
public class FileInputStreamConstructor throws IOException{
    public static void main(String[] args) {
      // 使用File对象创建流对象
        File file = new File("a.txt");
        FileInputStream fos = new FileInputStream(file);

        // 使用文件名称创建流对象
        FileInputStream fos = new FileInputStream("b.txt");
    }
}
```

读取字节数据

1. **读取字节**：`read`方法，每次可以读取一个字节的数据，提升为int类型，读取到文件末尾，返回`-1`，代码使用演示：

```
public class FISRead {
    public static void main(String[] args) throws IOException{
        // 使用文件名称创建流对象
        FileInputStream fis = new FileInputStream("read.txt");
        // 读取数据，返回一个字节
        int read = fis.read();
        System.out.println((char) read);
        read = fis.read();
        System.out.println((char) read);
        read = fis.read();
        System.out.println((char) read);
        read = fis.read();
        System.out.println((char) read);
        read = fis.read();
        System.out.println((char) read);
        // 读取到末尾,返回-1
        read = fis.read();
        System.out.println( read);
    // 关闭资源
        fis.close();
    }
}
输出结果:
a
b
c
d
e
-1
```

循环改进读取方式，代码使用演示：

```
public class FISRead {
    public static void main(String[] args) throws IOException{
        // 使用文件名称创建流对象
        FileInputStream fis = new FileInputStream("read.txt");
        // 定义变量，保存数据
        int b ;
        // 循环读取
        while ((b = fis.read())!=-1) {
            System.out.println((char)b);
        }
    // 关闭资源
        fis.close();
    }
```

```
}
输出结果:
a
b
c
d
e
```

> 小贴士:
>
> 1. 虽然读取了一个字节，但是会自动提升为int类型。
> 2. 流操作完毕后，必须释放系统资源，调用close方法，千万记得。

2. **使用字节数组读取**：`read(byte[] b)`，每次读取b的长度个字节到数组中，返回读取到的有效字节个数，读取到末尾时，返回`-1`，代码使用演示：

```
public class FISRead {
    public static void main(String[] args) throws IOException{
        // 使用文件名称创建流对象.
        FileInputStream fis = new FileInputStream("read.txt"); // 文件中为abcde
        // 定义变量，作为有效个数
        int len ;
        // 定义字节数组，作为装字节数据的容器
        byte[] b = new byte[2];
        // 循环读取
        while (( len= fis.read(b))!=-1) {
            // 每次读取后,把数组变成字符串打印
            System.out.println(new String(b));
        }
    // 关闭资源
        fis.close();
    }
}

输出结果:
ab
cd
ed
```

错误数据`d`，是由于最后一次读取时，只读取一个字节`e`，数组中，上次读取的数据没有被完全替换，所以要通过`len`，获取有效的字节，代码使用演示：

```
public class FISRead {
    public static void main(String[] args) throws IOException{
        // 使用文件名称创建流对象.
        FileInputStream fis = new FileInputStream("read.txt"); // 文件中为abcde
        // 定义变量，作为有效个数
        int len ;
        // 定义字节数组，作为装字节数据的容器
```

```
        byte[] b = new byte[2];
        // 循环读取
        while (( len= fis.read(b))!=-1) {
            // 每次读取后,把数组的有效字节部分，变成字符串打印
            System.out.println(new String(b, 0, len));//  len 每次读取的有效字节个
数
        }
    // 关闭资源
        fis.close();
    }
}
```

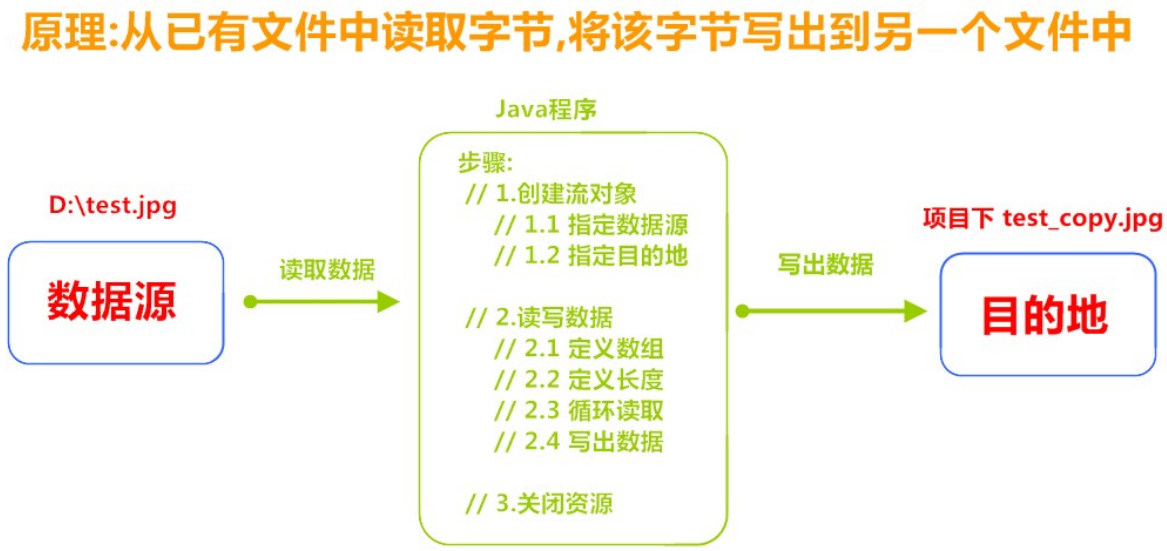
输出结果：

```
ab
cd
e
```

> 小贴士：
>
> 使用数组读取，每次读取多个字节，减少了系统间的IO操作次数，从而提高了读写的效率，建议开发中使用。

5.6 字节流练习：图片复制

复制原理图解



案例实现

复制图片文件，代码使用演示：

```
public class Copy {
    public static void main(String[] args) throws IOException {
        // 1.创建流对象
        // 1.1 指定数据源
```

```
        FileInputStream fis = new FileInputStream("D:\\test.jpg");
        // 1.2 指定目的地
        FileOutputStream fos = new FileOutputStream("test_copy.jpg");

        // 2.读写数据
        // 2.1 定义数组
        byte[] b = new byte[1024];
        // 2.2 定义长度
        int len;
        // 2.3 循环读取
        while ((len = fis.read(b))!=-1) {
            // 2.4 写出数据
            fos.write(b, 0 , len);
        }

        // 3.关闭资源
        fos.close();
        fis.close();
    }
}
```

> 小贴士：
>
> 流的关闭原则：先开后关，后开先关。

第六章 字符流

当使用字节流读取文本文件时，可能会有一个小问题。就是遇到中文字符时，可能不会显示完整的字符，那是因为一个中文字符可能占用多个字节存储。所以Java提供一些字符流类，以字符为单位读写数据，专门用于处理文本文件。

6.1 字符输入流【Reader】

`java.io.Reader`抽象类是表示用于读取字符流的所有类的超类，可以读取字符信息到内存中。它定义了字符输入流的基本共性功能方法。

- `public void close()` ：关闭此流并释放与此流相关联的任何系统资源。
- `public int read()`： 从输入流读取一个字符。
- `public int read(char[] cbuf)`： 从输入流中读取一些字符，并将它们存储到字符数组 cbuf 中 。

6.2 FileReader类

`java.io.FileReader`类是读取字符文件的便利类。构造时使用系统默认的字符编码和默认字节缓冲区。

> 小贴士：
>
> 1. 字符编码：字节与字符的对应规则。Windows系统的中文编码默认是GBK编码表。
>
>   idea中UTF-8

> 2. 字节缓冲区：一个字节数组，用来临时存储字节数据。

构造方法

- `FileReader(File file)`： 创建一个新的 FileReader ，给定要读取的File对象。
- `FileReader(String fileName)`： 创建一个新的 FileReader ，给定要读取的文件的名称。

当你创建一个流对象时，必须传入一个文件路径。类似于FileInputStream 。

- 构造举例，代码如下：

```
public class FileReaderConstructor throws IOException{
    public static void main(String[] args) {
      // 使用File对象创建流对象
        File file = new File("a.txt");
        FileReader fr = new FileReader(file);

        // 使用文件名称创建流对象
        FileReader fr = new FileReader("b.txt");
    }
}
```

读取字符数据

1. **读取字符**：`read`方法，每次可以读取一个字符的数据，提升为int类型，读取到文件末尾，返回`-1`，循环读取，代码使用演示：

```
public class FRRead {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileReader fr = new FileReader("read.txt");
        // 定义变量，保存数据
        int b ;
        // 循环读取
        while ((b = fr.read())!=-1) {
            System.out.println((char)b);
        }
    // 关闭资源
        fr.close();
    }
}
输出结果:
黑
马
程
序
员
```

> 小贴士：虽然读取了一个字符，但是会自动提升为int类型。

2. **使用字符数组读取**：`read(char[] cbuf)`，每次读取b的长度个字符到数组中，返回读取到的有效字符个数，读取到末尾时，返回`-1`，代码使用演示：

```
public class FRRead {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileReader fr = new FileReader("read.txt");
        // 定义变量，保存有效字符个数
        int len ;
        // 定义字符数组，作为装字符数据的容器
         char[] cbuf = new char[2];
        // 循环读取
        while ((len = fr.read(cbuf))!=-1) {
            System.out.println(new String(cbuf));
        }
    // 关闭资源
        fr.close();
    }
}
输出结果：
黑马
程序
员序
```

获取有效的字符改进，代码使用演示：

```
public class FISRead {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileReader fr = new FileReader("read.txt");
        // 定义变量，保存有效字符个数
        int len ;
        // 定义字符数组，作为装字符数据的容器
        char[] cbuf = new char[2];
        // 循环读取
        while ((len = fr.read(cbuf))!=-1) {
            System.out.println(new String(cbuf,0,len));
        }
      // 关闭资源
        fr.close();
    }
}

输出结果：
黑马
程序
员
```

6.3 字符输出流【Writer】

`java.io.Writer`抽象类是表示用于写出字符流的所有类的超类，将指定的字符信息写出到目的地。它定义了字节输出流的基本共性功能方法。

- `void write(int c)` 写入单个字符。
- `void write(char[] cbuf)`写入字符数组。
- `abstract  void write(char[] cbuf, int off, int len)`写入字符数组的某一部分,off数组的开始索引,len写的字符个数。
- `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,off字符串的开始索引,len写的字符个数。
- `void flush()`刷新该流的缓冲。
- `void close()` 关闭此流，但要先刷新它。

6.4 FileWriter类

`java.io.FileWriter`类是写出字符到文件的便利类。构造时使用系统默认的字符编码和默认字节缓冲区。

构造方法

- `FileWriter(File file)`： 创建一个新的 FileWriter，给定要读取的File对象。
- `FileWriter(String fileName)`： 创建一个新的 FileWriter，给定要读取的文件的名称。

当你创建一个流对象时，必须传入一个文件路径，类似于FileOutputStream。

- 构造举例，代码如下：

```
public class FileWriterConstructor {
    public static void main(String[] args) throws IOException {
      // 使用File对象创建流对象
        File file = new File("a.txt");
        FileWriter fw = new FileWriter(file);

        // 使用文件名称创建流对象
        FileWriter fw = new FileWriter("b.txt");
    }
}
```

基本写出数据

写出字符：`write(int b)` 方法，每次可以写出一个字符数据，代码使用演示：

```
public class FWWrite {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileWriter fw = new FileWriter("fw.txt");
        // 写出数据
```

```
        fw.write(97); // 写出第1个字符
        fw.write('b'); // 写出第2个字符
        fw.write('C'); // 写出第3个字符
        fw.write(30000); // 写出第4个字符，中文编码表中30000对应一个汉字。

        /*
        【注意】关闭资源时,与FileOutputStream不同。
         如果不关闭,数据只是保存到缓冲区，并未保存到文件。
        */
        // fw.close();
    }
}
```

输出结果：

abC田

> 小贴士：
>
> 1. 虽然参数为int类型四个字节，但是只会保留一个字符的信息写出。
> 2. 未调用close方法，数据只是保存到了缓冲区，并未写出到文件中。

关闭和刷新

因为内置缓冲区的原因，如果不关闭输出流，无法写出字符到文件中。但是关闭的流对象，是无法继续写出数据的。如果我们既想写出数据，又想继续使用流，就需要`flush`方法了。

- `flush`：刷新缓冲区，流对象可以继续使用。
- `close`:先刷新缓冲区，然后通知系统释放资源。流对象不可以再被使用了。

代码使用演示：

```
public class FWWrite {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileWriter fw = new FileWriter("fw.txt");
        // 写出数据，通过flush
        fw.write('刷'); // 写出第1个字符
        fw.flush();
        fw.write('新'); // 继续写出第2个字符，写出成功
        fw.flush();

        // 写出数据，通过close
        fw.write('关'); // 写出第1个字符
        fw.close();
        fw.write('闭'); // 继续写出第2个字符,【报错】java.io.IOException: Stream
closed
        fw.close();
    }
}
```

> 小贴士：即便是flush方法写出了数据，操作的最后还是要调用close方法，释放系统资源。

写出其他数据

1. **写出字符数组**：`write(char[] cbuf)` 和 `write(char[] cbuf, int off, int len)`，每次可以写出字符数组中的数据，用法类似FileOutputStream，代码使用演示：

```
public class FWWrite {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileWriter fw = new FileWriter("fw.txt");
        // 字符串转换为字节数组
        char[] chars = "黑马程序员".toCharArray();

        // 写出字符数组
        fw.write(chars); // 黑马程序员

    // 写出从索引2开始，2个字节。索引2是'程'，两个字节，也就是'程序'。
        fw.write(b,2,2); // 程序

        // 关闭资源
        fos.close();
    }
}
```

2. **写出字符串**：`write(String str)` 和 `write(String str, int off, int len)`，每次可以写出字符串中的数据，更为方便，代码使用演示：

```
public class FWWrite {
    public static void main(String[] args) throws IOException {
        // 使用文件名称创建流对象
        FileWriter fw = new FileWriter("fw.txt");
        // 字符串
        String msg = "黑马程序员";

        // 写出字符数组
        fw.write(msg); //黑马程序员

    // 写出从索引2开始，2个字节。索引2是'程'，两个字节，也就是'程序'。
        fw.write(msg,2,2);  // 程序

        // 关闭资源
        fos.close();
    }
}
```

3. **续写和换行**：操作类似于FileOutputStream。

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

```
        // 使用文件名称创建流对象，可以续写数据
        FileWriter fw = new FileWriter("fw.txt", true);
        // 写出字符串
        fw.write("黑马");
        // 写出换行
        fw.write("\r\n");
        // 写出字符串
      fw.write("程序员");
        // 关闭资源
        fw.close();
    }
}
输出结果：
黑马
程序员
```

> 小贴士：字符流，只能操作文本文件，不能操作图片，视频等非文本文件。
>
> 当我们单纯读或者写文本文件时 使用字符流 其他情况使用字节流

第七章 IO异常的处理

7.1 JDK7前处理

之前的入门练习，我们一直把异常抛出，而实际开发中并不能这样处理，建议使用`try...catch...finally`代码块，处理异常部分，代码使用演示：

```
public class HandleException1 {
    public static void main(String[] args) {
        // 声明变量
        FileWriter fw = null;
        try {
            //创建流对象
            fw = new FileWriter("fw.txt");
            // 写出数据
            fw.write("黑马程序员"); //黑马程序员
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (fw != null) {
                    fw.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

7.2 JDK7的处理(扩展知识点了解内容)

还可以使用JDK7优化后的`try-with-resource`语句，该语句确保了每个资源在语句结束时关闭。所谓的资源（resource）是指在程序完成后，必须关闭的对象。

格式：

```
try (创建流对象语句，如果多个,使用';'隔开) {
  // 读写数据
} catch (IOException e) {
  e.printStackTrace();
}
```

代码使用演示：

```
public class HandleException2 {
    public static void main(String[] args) {
        // 创建流对象
        try ( FileWriter fw = new FileWriter("fw.txt"); ) {
            // 写出数据
            fw.write("黑马程序员"); //黑马程序员
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

7.3 JDK9的改进(扩展知识点了解内容)

JDK9中`try-with-resource`的改进，对于**引入对象**的方式，支持的更加简洁。被引入的对象，同样可以自动关闭，无需手动close，我们来了解一下格式。

改进前格式：

```
// 被final修饰的对象
final Resource resource1 = new Resource("resource1");
// 普通对象
Resource resource2 = new Resource("resource2");
// 引入方式：创建新的变量保存
try (Resource r1 = resource1;
     Resource r2 = resource2) {
     // 使用对象
}
```

改进后格式：

```
// 被final修饰的对象
final Resource resource1 = new Resource("resource1");
// 普通对象
Resource resource2 = new Resource("resource2");

// 引入方式：直接引入
try (resource1; resource2) {
     // 使用对象
}
```

改进后，代码使用演示：

```
public class TryDemo {
    public static void main(String[] args) throws IOException {
        // 创建流对象
        final  FileReader fr  = new FileReader("in.txt");
        FileWriter fw = new FileWriter("out.txt");
        // 引入到try中
        try (fr; fw) {
            // 定义变量
            int b;
            // 读取数据
            while ((b = fr.read())!=-1) {
              // 写出数据
              fw.write(b);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

第八章 属性集

8.1 概述

`java.util.Properties` 继承于 `Hashtable` ，来表示一个持久的属性集。它使用键值结构存储数据，每个键及其对应值都是一个字符串。该类也被许多Java类使用，比如获取系统属性时， `System.getProperties` 方法就是返回一个 `Properties` 对象。

8.2 Properties类

构造方法

- `public Properties()` :创建一个空的属性列表。

基本的存储方法

- `public Object setProperty(String key, String value)`： 保存一对属性。
- `public String getProperty(String key)`：使用此属性列表中指定的键搜索属性值。
- `public Set<String> stringPropertyNames()`：所有键的名称的集合。

```
public class ProDemo {
    public static void main(String[] args) throws FileNotFoundException {
        // 创建属性集对象
        Properties properties = new Properties();
        // 添加键值对元素
        properties.setProperty("filename", "a.txt");
        properties.setProperty("length", "209385038");
        properties.setProperty("location", "D:\\a.txt");
        // 打印属性集对象
        System.out.println(properties);
        // 通过键,获取属性值
        System.out.println(properties.getProperty("filename"));
        System.out.println(properties.getProperty("length"));
        System.out.println(properties.getProperty("location"));

        // 遍历属性集,获取所有键的集合
        Set<String> strings = properties.stringPropertyNames();
        // 打印键值对
        for (String key : strings ) {
            System.out.println(key+" -- "+properties.getProperty(key));
        }
    }
}
输出结果:
{filename=a.txt, length=209385038, location=D:\a.txt}
a.txt
209385038
D:\a.txt
filename -- a.txt
length -- 209385038
location -- D:\a.txt
```

与流相关的方法

- `public void load(InputStream inStream)`： 从字节输入流中读取键值对。

参数中使用了字节输入流，通过流对象，可以关联到某文件上，这样就能够加载文本中的数据了。文本数据格式:

```
filename=a.txt
length=209385038
location=D:\a.txt
```

加载代码演示:

```
public class ProDemo2 {
    public static void main(String[] args) throws FileNotFoundException {
        // 创建属性集对象
        Properties pro = new Properties();
        // 加载文本中信息到属性集
        pro.load(new FileInputStream("read.txt"));
        // 遍历集合并打印
        Set<String> strings = pro.stringPropertyNames();
        for (String key : strings ) {
            System.out.println(key+" -- "+pro.getProperty(key));
        }
     }
}
```

输出结果：

```
filename -- a.txt
length -- 209385038
location -- D:\a.txt
```

> 小贴士：文本中的数据，必须是键值对形式，可以使用空格、等号、冒号等符号分隔。

Part3：缓冲流、转换流、序列化流、打印流

- [ ] 能够使用字节缓冲流读取数据到程序
- [ ] 能够使用字节缓冲流写出数据到文件
- [ ] 能够明确字符缓冲流的作用和基本用法
- [ ] 能够使用缓冲流的特殊功能
- [ ] 能够阐述编码表的意义
- [ ] 能够使用转换流读取指定编码的文本文件
- [ ] 能够使用转换流写入指定编码的文本文件
- [ ] 能够说出打印流的特点
- [ ] 能够使用序列化流写出对象到文件
- [ ] 能够使用反序列化流读取文件到程序中

第九章 缓冲流

昨天学习了基本的一些流，作为IO流的入门，今天我们要见识一些更强大的流。比如能够高效读写的缓冲流，能够转换编码的转换流，能够持久化存储对象的序列化流等等。这些功能更为强大的流，都是在基本的流对象基础之上创建而来的，就像穿上铠甲的武士一样，相当于是对基本流对象的一种增强。

9.1 概述

缓冲流,也叫高效流，是对4个基本的`FileXxx`流的增强，所以也是4个流，按照数据类型分类：

- **字节缓冲流**：`BufferedInputStream`，`BufferedOutputStream`

- **字符缓冲流**：`BufferedReader`，`BufferedWriter`

缓冲流的基本原理，是在创建流对象时，会创建一个内置的默认大小的缓冲区数组，通过缓冲区读写，减少系统IO次数，从而提高读写的效率。

9.2 字节缓冲流

构造方法

- `public BufferedInputStream(InputStream in)`：创建一个 新的缓冲输入流。
- `public BufferedOutputStream(OutputStream out)`： 创建一个新的缓冲输出流。

构造举例，代码如下：

```
// 创建字节缓冲输入流
BufferedInputStream bis = new BufferedInputStream(new
FileInputStream("bis.txt"));
// 创建字节缓冲输出流
BufferedOutputStream bos = new BufferedOutputStream(new
FileOutputStream("bos.txt"));
```

效率测试

查询API，缓冲流读写方法与基本的流是一致的，我们通过复制大文件（375MB），测试它的效率。

1. 基本流，代码如下：

```
public class BufferedDemo {
    public static void main(String[] args) throws FileNotFoundException {
        // 记录开始时间
        long start = System.currentTimeMillis();
    // 创建流对象
        try (
          FileInputStream fis = new FileInputStream("jdk9.exe");
          FileOutputStream fos = new FileOutputStream("copy.exe")
        ){
          // 读写数据
            int b;
            while ((b = fis.read()) != -1) {
                fos.write(b);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    // 记录结束时间
        long end = System.currentTimeMillis();
        System.out.println("普通流复制时间:"+(end - start)+" 毫秒");
    }
}
```

```
十几分钟过去了...
```

2. 缓冲流，代码如下：

```
public class BufferedDemo {
    public static void main(String[] args) throws FileNotFoundException {
        // 记录开始时间
        long start = System.currentTimeMillis();
    // 创建流对象
        try (
          BufferedInputStream bis = new BufferedInputStream(new
FileInputStream("jdk9.exe"));
       BufferedOutputStream bos = new BufferedOutputStream(new
FileOutputStream("copy.exe"));
        ){
        // 读写数据
            int b;
            while ((b = bis.read()) != -1) {
                bos.write(b);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    // 记录结束时间
        long end = System.currentTimeMillis();
        System.out.println("缓冲流复制时间:"+(end - start)+" 毫秒");
    }
}

缓冲流复制时间:8016 毫秒
```

如何更快呢?

使用数组的方式，代码如下：

```
public class BufferedDemo {
    public static void main(String[] args) throws FileNotFoundException {
        // 记录开始时间
        long start = System.currentTimeMillis();
    // 创建流对象
        try (
      BufferedInputStream bis = new BufferedInputStream(new
FileInputStream("jdk9.exe"));
     BufferedOutputStream bos = new BufferedOutputStream(new
FileOutputStream("copy.exe"));
        ){
            // 读写数据
            int len;
            byte[] bytes = new byte[8*1024];
```

```
            while ((len = bis.read(bytes)) != -1) {
                bos.write(bytes, 0 , len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    // 记录结束时间
        long end = System.currentTimeMillis();
        System.out.println("缓冲流使用数组复制时间:"+(end - start)+" 毫秒");
    }
}
缓冲流使用数组复制时间:666 毫秒
```

9.3 字符缓冲流

构造方法

- `public BufferedReader(Reader in)`：创建一个 新的缓冲输入流。
- `public BufferedWriter(Writer out)`： 创建一个新的缓冲输出流。

构造举例，代码如下：

```
// 创建字符缓冲输入流
BufferedReader br = new BufferedReader(new FileReader("br.txt"));
// 创建字符缓冲输出流
BufferedWriter bw = new BufferedWriter(new FileWriter("bw.txt"));
```

特有方法

字符缓冲流的基本方法与普通字符流调用方式一致，不再阐述，我们来看它们具备的特有方法。

- BufferedReader：`public String readLine()`：读一行文字。
- BufferedWriter：`public void newLine()`：写一行行分隔符,由系统属性定义符号。

`readLine`方法演示，代码如下：

```
public class BufferedReaderDemo {
    public static void main(String[] args) throws IOException {
         // 创建流对象
        BufferedReader br = new BufferedReader(new FileReader("in.txt"));
    // 定义字符串,保存读取的一行文字
        String line  = null;
        // 循环读取,读取到最后返回null
        while ((line = br.readLine())!=null) {
            System.out.print(line);
            System.out.println("------");
        }
    // 释放资源
        br.close();
```

```
    }
}
```

`newLine`方法演示，代码如下：

```
public class BufferedWriterDemo throws IOException {
    public static void main(String[] args) throws IOException  {
        // 创建流对象
    BufferedWriter bw = new BufferedWriter(new FileWriter("out.txt"));
        // 写出数据
        bw.write("黑马");
        // 写出换行
        bw.newLine();
        bw.write("程序");
        bw.newLine();
        bw.write("员");
        bw.newLine();
    // 释放资源
        bw.close();
    }
}
输出效果：
黑马
程序
员
```

9.4 练习:文本排序

请将文本信息恢复顺序。

3.侍中、侍郎郭攸之、费祎、董允等，此皆良实，志虑忠纯，是以先帝简拔以遗陛下。愚以为宫中之事，事无大小，悉以咨之，然后施行，必得裨补阙漏，有所广益。
8.愿陛下托臣以讨贼兴复之效，不效，则治臣之罪，以告先帝之灵。若无兴德之言，则责攸之、祎、允等之慢，以彰其咎；陛下亦宜自谋，以咨诹善道，察纳雅言，深追先帝遗诏，臣不胜受恩感激。
4.将军向宠，性行淑均，晓畅军事，试用之于昔日，先帝称之曰能，是以众议举宠为督。愚以为营中之事，悉以咨之，必能使行阵和睦，优劣得所。
2.宫中府中，俱为一体，陟罚臧否，不宜异同。若有作奸犯科及为忠善者，宜付有司论其刑赏，以昭陛下平明之理，不宜偏私，使内外异法也。
1.先帝创业未半而中道崩殂，今天下三分，益州疲弊，此诚危急存亡之秋也。然侍卫之臣不懈于内，忠志之士忘身于外者，盖追先帝之殊遇，欲报之于陛下也。诚宜开张圣听，以光先帝遗德，恢弘志士之气，不宜妄自菲薄，引喻失义，以塞忠谏之路也。
9.今当远离，临表涕零，不知所言。
6.臣本布衣，躬耕于南阳，苟全性命于乱世，不求闻达于诸侯。先帝不以臣卑鄙，猥自枉屈，三顾臣于草庐之中，咨臣以当世之事，由是感激，遂许先帝以驱驰。后值倾覆，受任于败军之际，奉命于危难之间，尔来二十有一年矣。
7.先帝知臣谨慎，故临崩寄臣以大事也。受命以来，夙夜忧叹，恐付托不效，以伤先帝之明，故五月渡泸，深入不毛。今南方已定，兵甲已足，当奖率三军，北定中原，庶竭驽钝，攘除奸凶，兴复汉室，还于旧都。此臣所以报先帝而忠陛下之职分也。至于斟酌损益，进尽忠言，则攸之、祎、允之任也。
5.亲贤臣，远小人，此先汉所以兴隆也；亲小人，远贤臣，此后汉所以倾颓也。先帝在时，每与臣论此事，未尝不叹息痛恨于桓、灵也。侍中、尚书、长史、参军，此悉贞良死节之臣，愿陛下亲之信之，则汉室之隆，可计日而待也。

案例分析

1. 逐行读取文本信息。
2. 解析文本信息到集合中。
3. 遍历集合，按顺序，写出文本信息。

案例实现

```
public class BufferedTest {
    public static void main(String[] args) throws IOException {
        // 创建map集合,保存文本数据,键为序号,值为文字
        HashMap<String, String> lineMap = new HashMap<>();

        // 创建流对象
        BufferedReader br = new BufferedReader(new FileReader("in.txt"));
        BufferedWriter bw = new BufferedWriter(new FileWriter("out.txt"));

        // 读取数据
        String line  = null;
        while ((line = br.readLine())!=null) {
            // 解析文本
            String[] split = line.split("\\.");
            // 保存到集合
            lineMap.put(split[0],split[1]);
        }
        // 释放资源
        br.close();
```

```
        // 遍历map集合
        for (int i = 1; i <= lineMap.size(); i++) {
            String key = String.valueOf(i);
            // 获取map中文本
            String value = lineMap.get(key);
            // 写出拼接文本
            bw.write(key+"."+value);
            // 写出换行
            bw.newLine();
        }
    // 释放资源
        bw.close();
    }
}
```

第十章 转换流

10.1 字符编码和字符集

字符编码

计算机中储存的信息都是用二进制数表示的，而我们在屏幕上看到的数字、英文、标点符号、汉字等字符是二进制数转换之后的结果。按照某种规则，将字符存储到计算机中，称为**编码** 。反之，将存储在计算机中的二进制数按照某种规则解析显示出来，称为**解码** 。比如说，按照A规则存储，同样按照A规则解析，那么就能显示正确的文本符号。反之，按照A规则存储，再按照B规则解析，就会导致乱码现象。

编码:字符(能看懂的)--字节(看不懂的)

解码:字节(看不懂的)-->字符(能看懂的)

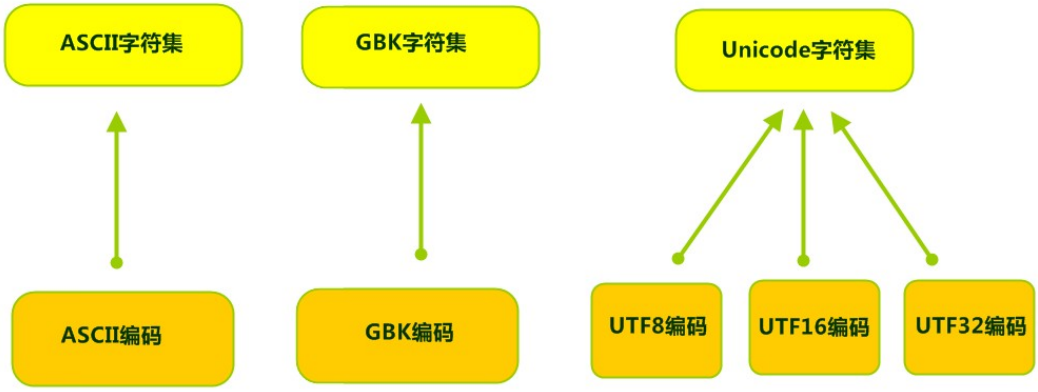
- **字符编码**`Character Encoding`：就是一套自然语言的字符与二进制数之间的对应规则。

  编码表:生活中文字和计算机中二进制的对应规则

字符集

- **字符集** `Charset`： 也叫编码表。是一个系统支持的所有字符的集合，包括各国家文字、标点符号、图形符号、数字等。

计算机要准确的存储和识别各种字符集符号，需要进行字符编码，一套字符集必然至少有一套字符编码。常见字符集有ASCII字符集、GBK字符集、Unicode字符集等。



可见，当指定了**编码**，它所对应的**字符集**自然就指定了，所以**编码**才是我们最终要关心的。

- **ASCII字符集**：
  - ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）是基于拉丁字母的一套电脑编码系统，用于显示现代英语，主要包括控制字符（回车键、退格、换行键等）和可显示字符（英文大小写字符、阿拉伯数字和西文符号）。
  - 基本的ASCII字符集，使用7位（bits）表示一个字符，共128字符。ASCII的扩展字符集使用8位（bits）表示一个字符，共256字符，方便支持欧洲常用字符。
- **ISO-8859-1字符集**：
  - 拉丁码表，别名Latin-1，用于显示欧洲使用的语言，包括荷兰、丹麦、德语、意大利语、西班牙语等。
  - ISO-8859-1使用单字节编码，兼容ASCII编码。
- **GBxxx字符集**：
  - GB就是国标的意思，是为了显示中文而设计的一套字符集。
  - **GB2312**：简体中文码表。一个小于127的字符的意义与原来相同。但两个大于127的字符连在一起时，就表示一个汉字，这样大约可以组合了包含7000多个简体汉字，此外数学符号、罗马希腊的字母、日文的假名们都编进去了，连在ASCII里本来就有的数字、标点、字母都统统重新编了两个字节长的编码，这就是常说的"全角"字符，而原来在127号以下的那些就叫"半角"字符了。
  - **GBK**：最常用的中文码表。是在GB2312标准基础上的扩展规范，使用了双字节编码方案，共收录了21003个汉字，完全兼容GB2312标准，同时支持繁体汉字以及日韩汉字等。
  - **GB18030**：最新的中文码表。收录汉字70244个，采用多字节编码，每个字可以由1个、2个或4个字节组成。支持中国国内少数民族的文字，同时支持繁体汉字以及日韩汉字等。
- **Unicode字符集**：
  - Unicode编码系统为表达任意语言的任意字符而设计，是业界的一种标准，也称为统一码、标准万国码。
  - 它最多使用4个字节的数字来表达每个字母、符号，或者文字。有三种编码方案，UTF-8、UTF-16和UTF-32。最为常用的UTF-8编码。
  - UTF-8编码，可以用来表示Unicode标准中任何字符，它是电子邮件、网页及其他存储或传送文字的应用中，优先采用的编码。互联网工程工作小组（IETF）要求所有互联网协议都必须支持UTF-8编码。所以，我们开发Web应用，也要使用UTF-8编码。它使用一至四个字节为每个字符编码，编码规则：
    1. 128个US-ASCII字符，只需一个字节编码。

2. 拉丁文等字符，需要二个字节编码。
3. 大部分常用字（含中文），使用三个字节编码。
4. 其他极少使用的Unicode辅助字符，使用四字节编码。

10.2 编码引出的问题

在IDEA中，使用`FileReader`读取项目中的文本文件。由于IDEA的设置，都是默认的`UTF-8`编码，所以没有任何问题。但是，当读取Windows系统中创建的文本文件时，由于Windows系统的默认是GBK编码，就会出现乱码。

```
public class ReaderDemo {
    public static void main(String[] args) throws IOException {
        FileReader fileReader = new FileReader("E:\\File_GBK.txt");
        int read;
        while ((read = fileReader.read()) != -1) {
            System.out.print((char)read);
        }
        fileReader.close();
    }
}
输出结果：
锘挎
```

那么如何读取GBK编码的文件呢?

10.3 InputStreamReader类

转换流`java.io.InputStreamReader`，是Reader的子类，是从字节流到字符流的桥梁。它读取字节，并使用指定的字符集将其解码为字符。它的字符集可以由名称指定，也可以接受平台的默认字符集。

构造方法

- `InputStreamReader(InputStream in)`：创建一个使用默认字符集的字符流。
- `InputStreamReader(InputStream in, String charsetName)`：创建一个指定字符集的字符流。

构造举例，代码如下：

```
InputStreamReader isr = new InputStreamReader(new FileInputStream("in.txt"));
InputStreamReader isr2 = new InputStreamReader(new FileInputStream("in.txt") ,
"GBK");
```

指定编码读取

```
public class ReaderDemo2 {
    public static void main(String[] args) throws IOException {
        // 定义文件路径,文件为gbk编码
```

```
        String FileName = "E:\\file_gbk.txt";
        // 创建流对象,默认UTF8编码
        InputStreamReader isr = new InputStreamReader(new
FileInputStream(FileName));
        // 创建流对象,指定GBK编码
        InputStreamReader isr2 = new InputStreamReader(new
FileInputStream(FileName) , "GBK");
    // 定义变量,保存字符
        int read;
        // 使用默认编码字符流读取,乱码
        while ((read = isr.read()) != -1) {
            System.out.print((char)read); // ??h?
        }
        isr.close();

        // 使用指定编码字符流读取,正常解析
        while ((read = isr2.read()) != -1) {
            System.out.print((char)read);// 大家好
        }
        isr2.close();
    }
}
```

10.4 OutputStreamWriter类

转换流`java.io.OutputStreamWriter`，是Writer的子类，是从字符流到字节流的桥梁。使用指定的字符集将字符编码为字节。它的字符集可以由名称指定，也可以接受平台的默认字符集。

构造方法

- `OutputStreamWriter(OutputStream in)`: 创建一个使用默认字符集的字符流。
- `OutputStreamWriter(OutputStream in, String charsetName)`: 创建一个指定字符集的字符流。

构造举例，代码如下：

```
OutputStreamWriter isr = new OutputStreamWriter(new
FileOutputStream("out.txt"));
OutputStreamWriter isr2 = new OutputStreamWriter(new
FileOutputStream("out.txt") , "GBK");
```

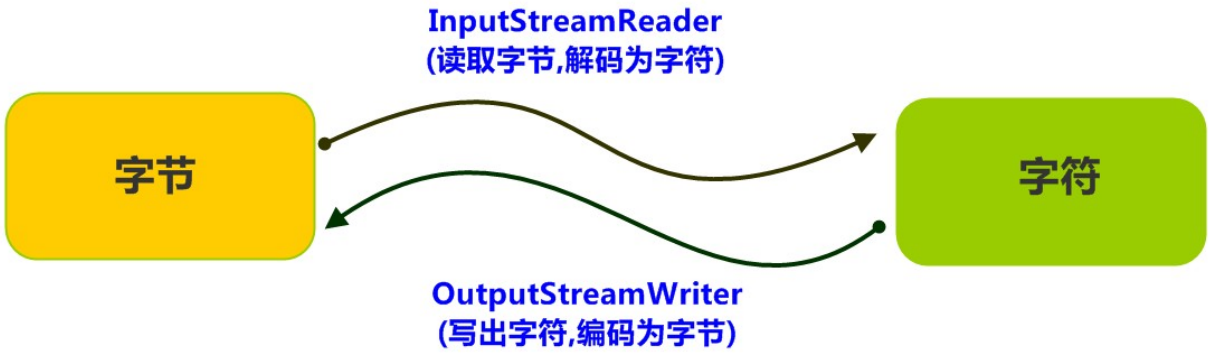
指定编码写出

```
public class OutputDemo {
    public static void main(String[] args) throws IOException {
        // 定义文件路径
        String FileName = "E:\\out.txt";
        // 创建流对象,默认UTF8编码
```

```
        OutputStreamWriter osw = new OutputStreamWriter(new
FileOutputStream(FileName));
        // 写出数据
        osw.write("你好"); // 保存为6个字节
        osw.close();

    // 定义文件路径
    String FileName2 = "E:\\out2.txt";
      // 创建流对象,指定GBK编码
        OutputStreamWriter osw2 = new OutputStreamWriter(new
FileOutputStream(FileName2),"GBK");
        // 写出数据
        osw2.write("你好");// 保存为4个字节
        osw2.close();
    }
}
```

转换流理解图解

转换流是字节与字符间的桥梁!



10.5 练习：转换文件编码

将GBK编码的文本文件，转换为UTF-8编码的文本文件。

案例分析

1. 指定GBK编码的转换流，读取文本文件。
2. 使用UTF-8编码的转换流，写出文本文件。

案例实现

```
public class TransDemo {
   public static void main(String[] args) {
      // 1.定义文件路径
      String srcFile = "file_gbk.txt";
        String destFile = "file_utf8.txt";
    // 2.创建流对象
      // 2.1 转换输入流,指定GBK编码
```

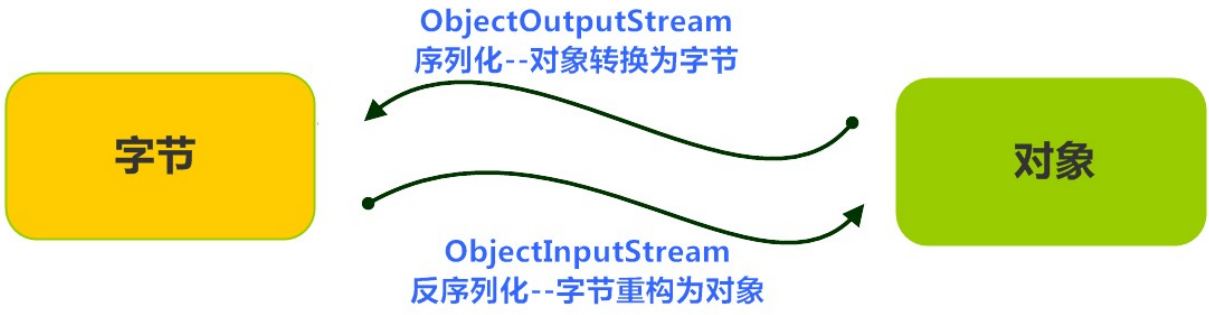
```
        InputStreamReader isr = new InputStreamReader(new
FileInputStream(srcFile) , "GBK");
      // 2.2 转换输出流,默认utf8编码
        OutputStreamWriter osw = new OutputStreamWriter(new
FileOutputStream(destFile));
    // 3.读写数据
      // 3.1 定义数组
        char[] cbuf = new char[1024];
      // 3.2 定义长度
        int len;
      // 3.3 循环读取
        while ((len = isr.read(cbuf))!=-1) {
            // 循环写出
            osw.write(cbuf,0,len);
        }
      // 4.释放资源
        osw.close();
        isr.close();
    }
}
```

第十一章 序列化

11.1 概述

Java 提供了一种对象**序列化**的机制。用一个字节序列可以表示一个对象，该字节序列包含该`对象的数据`、`对象的类型`和`对象中存储的属性`等信息。字节序列写出到文件之后，相当于文件中**持久保存**了一个对象的信息。

反之，该字节序列还可以从文件中读取回来，重构对象，对它进行**反序列化**。`对象的数据`、`对象的类型`和`对象中存储的数据`信息，都可以用来在内存中创建对象。看图理解序列化：



11.2 ObjectOutputStream类

`java.io.ObjectOutputStream`类，将Java对象的原始数据类型写出到文件,实现对象的持久存储。

构造方法

- `public ObjectOutputStream(OutputStream out)`： 创建一个指定OutputStream的ObjectOutputStream。

构造举例，代码如下：

```
FileOutputStream fileOut = new FileOutputStream("employee.txt");
ObjectOutputStream out = new ObjectOutputStream(fileOut);
```

序列化操作

1. 一个对象要想序列化，必须满足两个条件:

- 该类必须实现`java.io.Serializable`接口，`Serializable`是一个标记接口，不实现此接口的类将不会使任何状态序列化或反序列化，会抛出`NotSerializableException`。
- 该类的所有属性必须是可序列化的。如果有一个属性不需要可序列化的，则该属性必须注明是瞬态的，使用`transient`关键字修饰。

```
public class Employee implements java.io.Serializable {
    public String name;
    public String address;
    public transient int age; // transient瞬态修饰成员,不会被序列化
    public void addressCheck() {
        System.out.println("Address  check : " + name + " -- " + address);
    }
}
```

2.写出对象方法

- `public final void writeObject (Object obj)`:将指定的对象写出。

```
public class SerializeDemo{
    public static void main(String [] args)   {
      Employee e = new Employee();
      e.name = "zhangsan";
      e.address = "beiqinglu";
      e.age = 20;
      try {
          // 创建序列化流对象
          ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("employee.txt"));
          // 写出对象
          out.writeObject(e);
          // 释放资源
          out.close();
          fileOut.close();
          System.out.println("Serialized data is saved"); // 姓名，地址被序列化，
年龄没有被序列化。
        } catch(IOException i)   {
            i.printStackTrace();
```

```
        }
    }
}
```

输出结果：

```
Serialized data is saved
```

11.3 ObjectInputStream类

ObjectInputStream反序列化流，将之前使用ObjectOutputStream序列化的原始数据恢复为对象。

构造方法

- `public ObjectInputStream(InputStream in)`： 创建一个指定InputStream的ObjectInputStream。

反序列化操作1

如果能找到一个对象的class文件，我们可以进行反序列化操作，调用`ObjectInputStream`读取对象的方法：

- `public final Object readObject ()`：读取一个对象。

```
public class DeserializeDemo {
   public static void main(String [] args)   {
        Employee e = null;
        try {
            // 创建反序列化流
            FileInputStream fileIn = new FileInputStream("employee.txt");
            ObjectInputStream in = new ObjectInputStream(fileIn);
            // 读取一个对象
            e = (Employee) in.readObject();
            // 释放资源
            in.close();
            fileIn.close();
        }catch(IOException i) {
            // 捕获其他异常
            i.printStackTrace();
            return;
        }catch(ClassNotFoundException c)  {
          // 捕获类找不到异常
            System.out.println("Employee class not found");
            c.printStackTrace();
            return;
        }
        // 无异常,直接打印输出
        System.out.println("Name: " + e.name);  // zhangsan
        System.out.println("Address: " + e.address); // beiqinglu
        System.out.println("age: " + e.age); // 0
    }
}
```

对于JVM可以反序列化对象，它必须是能够找到class文件的类。如果找不到该类的class文件，则抛出一个 `ClassNotFoundException` 异常。

反序列化操作2

另外，当JVM反序列化对象时，能找到class文件，但是class文件在序列化对象之后发生了修改，那么反序列化操作也会失败，抛出一个`InvalidClassException`异常。发生这个异常的原因如下：

- 该类的序列版本号与从流中读取的类描述符的版本号不匹配
- 该类包含未知数据类型
- 该类没有可访问的无参数构造方法

`Serializable` 接口给需要序列化的类，提供了一个序列版本号。`serialVersionUID` 该版本号的目的在于验证序列化的对象和对应类是否版本匹配。

```
public class Employee implements java.io.Serializable {
     // 加入序列版本号
     private static final long serialVersionUID = 1L;
     public String name;
     public String address;
     // 添加新的属性 ,重新编译, 可以反序列化,该属性赋为默认值.
     public int eid;

     public void addressCheck() {
         System.out.println("Address  check : " + name + " -- " + address);
     }
}
```

11.4 练习：序列化集合

1. 将存有多个自定义对象的集合序列化操作，保存到`list.txt`文件中。
2. 反序列化`list.txt` ，并遍历集合，打印对象信息。

案例分析

1. 把若干学生对象 ，保存到集合中。
2. 把集合序列化。
3. 反序列化读取时，只需要读取一次，转换为集合类型。
4. 遍历集合，可以打印所有的学生信息

案例实现

```
public class SerTest {
  public static void main(String[] args) throws Exception {
    // 创建 学生对象
    Student student = new Student("老王", "laow");
    Student student2 = new Student("老张", "laoz");
```

```
    Student student3 = new Student("老李", "laol");

    ArrayList<Student> arrayList = new ArrayList<>();
    arrayList.add(student);
    arrayList.add(student2);
    arrayList.add(student3);
    // 序列化操作
    // serializ(arrayList);

    // 反序列化
    ObjectInputStream ois  = new ObjectInputStream(new
FileInputStream("list.txt"));
    // 读取对象,强转为ArrayList类型
    ArrayList<Student> list  = (ArrayList<Student>)ois.readObject();

        for (int i = 0; i < list.size(); i++ ){
            Student s = list.get(i);
          System.out.println(s.getName()+"--"+ s.getPwd());
        }
  }

  private static void serializ(ArrayList<Student> arrayList) throws Exception
{
    // 创建 序列化流
    ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("list.txt"));
    // 写出对象
    oos.writeObject(arrayList);
    // 释放资源
    oos.close();
  }
}
```

第十二章 打印流

12.1 概述

平时我们在控制台打印输出，是调用`print`方法和`println`方法完成的，这两个方法都来自于`java.io.PrintStream`类，该类能够方便地打印各种数据类型的值，是一种便捷的输出方式。

12.2 PrintStream类

构造方法

- `public PrintStream(String fileName)`：使用指定的文件名创建一个新的打印流。

构造举例，代码如下：

```
PrintStream ps = new PrintStream("ps.txt");
```

改变打印流向

`System.out`就是`PrintStream`类型的，只不过它的流向是系统规定的，打印在控制台上。不过，既然是流对象，我们就可以玩一个"小把戏"，改变它的流向。

```
public class PrintDemo {
    public static void main(String[] args) throws IOException {
    // 调用系统的打印流,控制台直接输出97
        System.out.println(97);

    // 创建打印流,指定文件的名称
        PrintStream ps = new PrintStream("ps.txt");

        // 设置系统的打印流流向,输出到ps.txt
        System.setOut(ps);
        // 调用系统的打印流,ps.txt中输出97
        System.out.println(97);
    }
}
```