I/O 框架

一、数据流

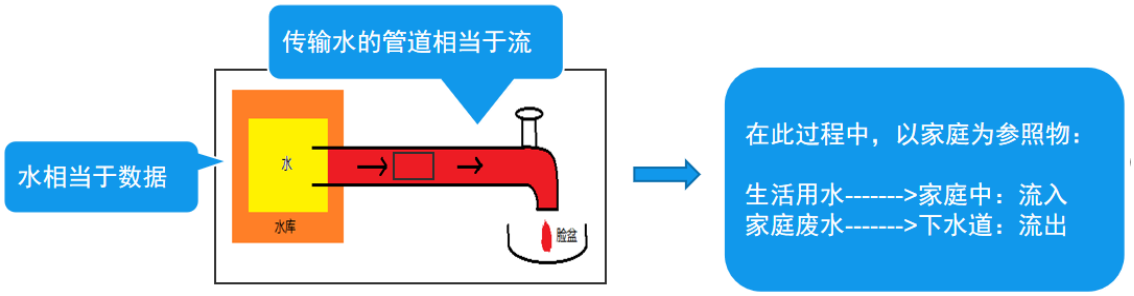
1.1 概念

> - 内存与硬盘存储之间传输数据的通道。
> - 水借助管道传输；数据借助流传输。

Java语言中I/O类库中常使用流这个概念,他代表任何有能力产出数据的数据源对象或者用能力接受数据源的对象!

通俗点说:I/O就是进行数据传递的一种手段!

生活中流



1.2 流的分类

> 通信的方式多种多样[文件,控制塔,网络数据,字符和字节等]所有Java的设计者,设计了大量的类来解决问题!
>
> 我们需要了解多种IO流的使用方式!

1.2.1 按方向【重点】

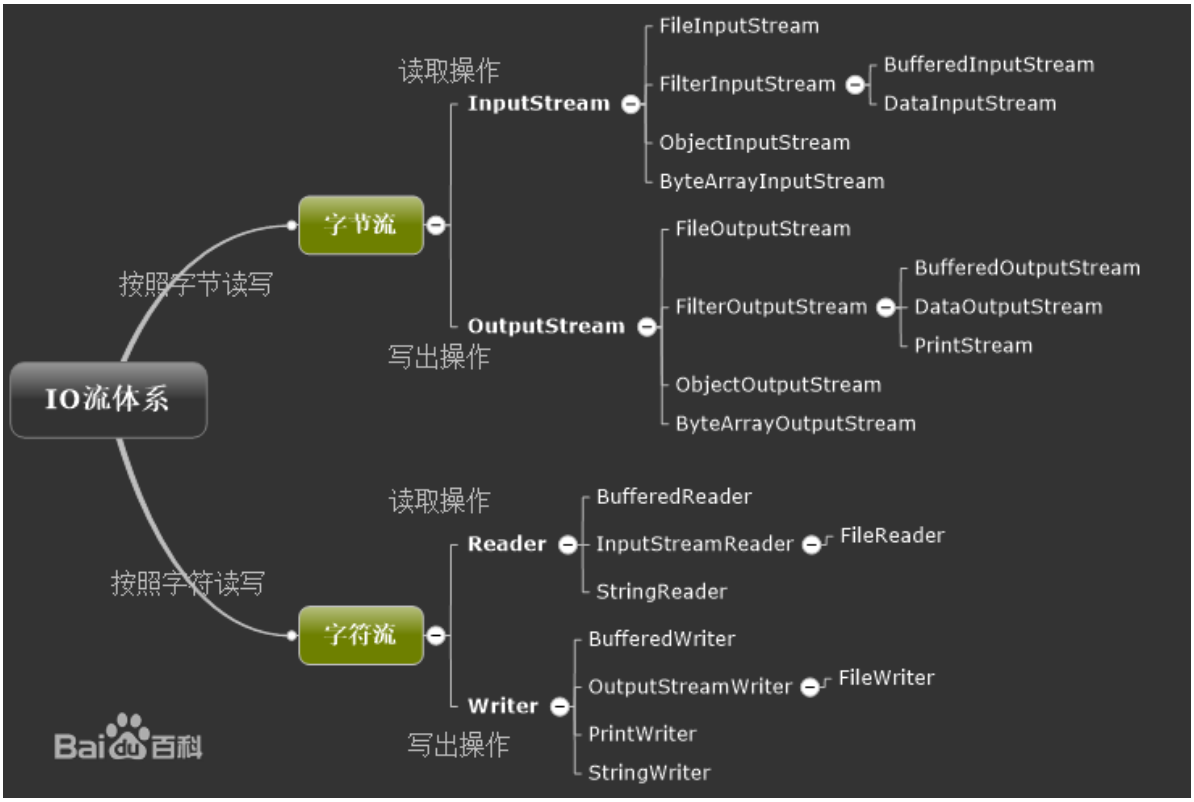
> - 输入流：将<存储设备>中的内容读入到<内存>中。
> - 输出流：将<内存>中的内容写入到<存储设备>中。

1.2.2 按单位

> - 字节流：以字节为单位，可以读写所有数据 。
> - 字符流：以字符为单位，只能读写文本数据 。

1.2.3 按功能

> - 节点流：具有实际传输数据的读写功能。
> - 过滤流：在节点流的基础之上增强功能。



二、字节流【重点】

2.1 字节抽象类

> InputStream：字节输入流
>
> - public int read(){}。读单个字节
> - public int read(byte[] b){}。读取字节到字节数组中
> - public int read(byte[] b,int off,int len){}。 读取字节到字节数组中! off数组的写偏移量, len数组中写入的长度
> - 注意: off+ len > b数组的长度会出现数组越界问题
> - 如果没有数据会返回 -1 如果有数据 返回读取的长度

> OutputStream：字节输出流
>
> - public void write(byte[] b){}。 将字节数组数据写出到指定的位置!
> - public void write(byte[] b,int off,int len){}。 将字节数组数据写出的位置! off代表数组写出的偏移量,len代表写出字节数量!

2.2 字节文件流[节点流实现类]

> FileOutputStream：
>
> - 文件输出流是用于将数据输出到外部文件
> - 用于写入诸如图像数据的原始字节流
> - Java程序--> 字节数据 --> 外部文件
> - 实例化方式:
>   - `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。

- `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。
- boolean append的参数代表是否追加,默认是覆盖
- `FileOutputStream(String name)` 创建文件输出流以指定的名称[全路径]写入文件。
- `FileOutputStream(String name,boolean append)` 创建文件输出流以指定的名称写入文件。

FileInputStream：

- 从文件系统中的文件获取输入字节。
- 用于读取诸如图像数据的原始字节流.
- 系统文件--> 字节数据 --> Java程序
- 实例化方式
  - `FileInputStream(String)`通过打开与实际文件的连接来创建一个 `FileInputStream`，该文件由文件系统的文件路径创建。
  - `FileInputStream(File file)` 通过打开与实际文件的连接创建一个 `FileInputStream`！

2.2.1 FileInputStream读取文件

```
/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/3 15:57
 * description:练习字节文件输入流!
 * todo:  将d:\\test.txt 文件中! 读入到程序中!
 */
public class ByteFileInStream {

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

        /**
         * 1.创建一个字节文件输入流!
         * @param: String  读取文件的磁盘位置!
         */
        FileInputStream fis = new FileInputStream("D:\\test.txt");


        //2.读取数据
        //2.1 单字节读取
//        int data = -1;
//        while ((data = fis.read()) != -1) {
//            System.out.println("read = " + (char) data);
//        }

        //2.2 多字节读取
        byte[] buf=new byte[1024];
        int count= -1;
        while((count=fis.read(buf))!=-1) {
            System.out.println(new String(buf,0,count));
        }
```

```
        //3.关闭流
        fis.close();
        System.out.println("执行完毕");


    }

}
```

2.2.2 FileOutputStream写入文件。

> 演示内容写出

```
/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2021/10/3 15:19
 * description:练习字节文件输出流!
 * todo:  在d:\\test.txt 文件中! 写入hello io!
 */
public class ByteFileOutStream {

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

        /**
         * 1.创建一个字节文件输出流!
         *   将数据从程序内存写到指定的磁盘文件!
         * @param: String  写入磁盘的位置!
         * @param: Boolean 是否追加
         */
        FileOutputStream fos  = new FileOutputStream("D:\\test.txt",true);

        //2. 写入内容
        fos.write("hello".getBytes("utf-8"));
        fos.write(" io!".getBytes("utf-8"));

        fos.close();
        System.out.println("写出执行完毕! ");
    }

}
```

> 演示换行写入

```
package com.atguigu.bytestream.out;

import java.io.FileOutputStream;

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2021/10/3 15:19
 * description:练习字节文件输出流!  换行练习
 * todo:  在d:\\test.txt 文件中! 写入hello io!
```

```
 */
public class ByteFileOutStreamChangeLine {

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

        /**
         * 1.创建一个字节文件输出流!
         *   将数据从程序内存写到指定的磁盘文件!
         * @param: String  写入磁盘的位置!
         * @param: Boolean 是否追加
         */
        FileOutputStream fos  = new FileOutputStream("D:\\test.txt",true);

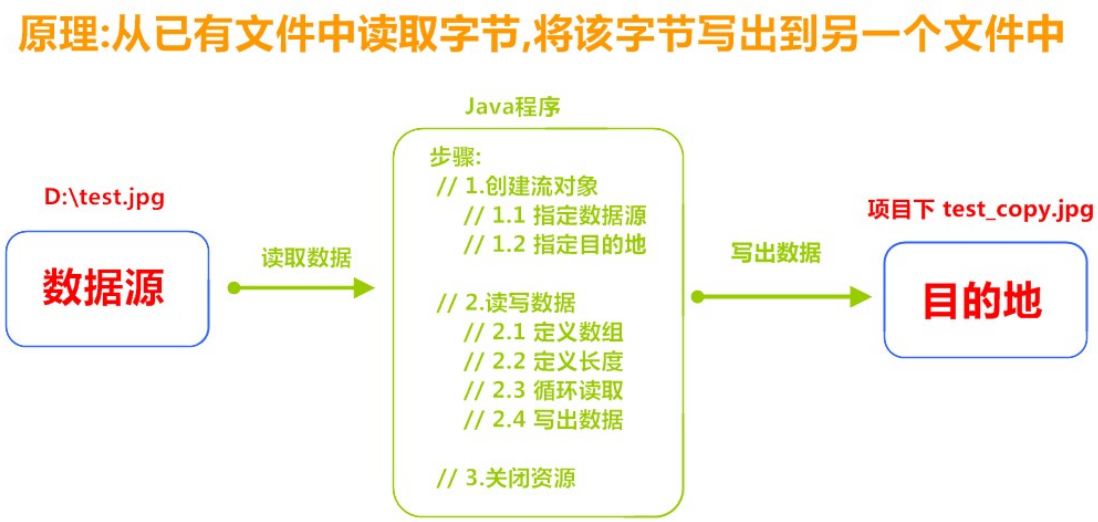
        //2. 写入内容
        fos.write("hello".getBytes("utf-8"));
        fos.write("\r\n".getBytes("utf-8"));
        fos.write(" io!".getBytes("utf-8"));

        fos.close();
        System.out.println("写出执行完毕! ");
    }

}
```

- 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。

2.2.3 文件操作练习




```
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);
        }

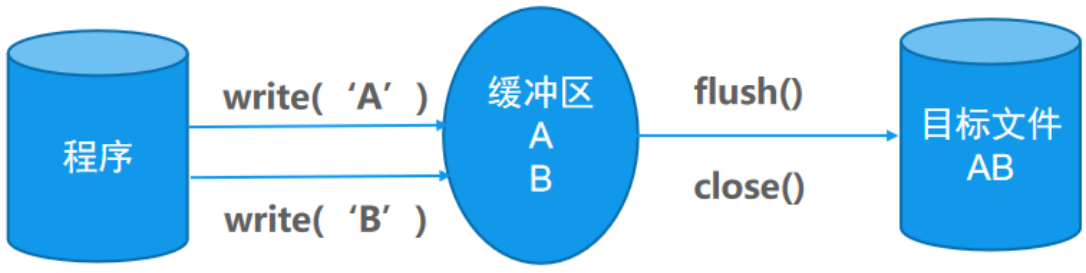
        // 3.关闭资源
        fos.close();
        fis.close();
    }
}
```

> 小贴士：
>
> 流的关闭原则：先开后关，后开先关。

2.3 字节缓冲流[过滤流]

> - 提高IO效率，减少访问磁盘的次数。
> - 数据存储在缓冲区中，flush是将缓存区的内容写入文件中，也可以直接close。

字节缓冲流



2.3.1 字节缓冲流分类

> - **字节缓冲流**：
>   - `BufferedInputStream` [字节缓冲输入流]

Constructor and Description
`BufferedInputStream(InputStream in)` 创建一个 `BufferedInputStream` 并保存其参数，输入流 `in` ，供以后使用。
`BufferedInputStream(InputStream in, int size)` 创建 `BufferedInputStream` 具有指定缓冲区大小，并保存其参数，输入流 `in` ，供以后使用。

- `BufferedOutputStream` [字节缓冲输出流]

Constructor and Description
`BufferedOutputStream(OutputStream out)` 创建一个新的缓冲输出流，以将数据写入指定的底层输出流。
`BufferedOutputStream(OutputStream out, int size)` 创建一个新的缓冲输出流，以便以指定的缓冲区大小将数据写入指定的底层输出流。

2.3.2 字节输入缓冲流原理

FileInputStream源码:

```
package java.io;

public class FileInputStream extends InputStream{

    /**
     *从输入流中读取一个字节
     *该方法为private私有方法，用户不能直接调用。
     *该方法为native本地方法，这是因为Java语言不能直接与操作系统或计算机硬件交互，
     *只能通过调用C/C++编写的本地方法来实现对磁盘数据的访问。
     */
    private native int read0() throws IOException;
    //调用native方法read0()每次读取一个字节
    public int read() throws IOException {
        Object traceContext = IoTrace.fileReadBegin(path);
        int b = 0;
        try {
            b = read0();
        } finally {
            IoTrace.fileReadEnd(traceContext, b == -1 ? 0 : 1);
        }
        return b;
    }
    /**
     * 从输入流中读取多个字节到byte数组中
     * 该方法也是私有本地方法，不对用户开放，只供内部调用。
     */
    private native int readBytes(byte b[], int off, int len) throws IOException;


    //调用native方法readBytes(b, 0, b.length)每次读取多个字节
    public int read(byte b[]) throws IOException {
        Object traceContext = IoTrace.fileReadBegin(path);
        int bytesRead = 0;
        try {
```

```
            bytesRead = readBytes(b, 0, b.length);
        } finally {
            IoTrace.fileReadEnd(traceContext, bytesRead == -1 ? 0 : bytesRead);

        }
        return bytesRead;
    }
    //从此输入流中将最多 len 个字节的数据读入一个 byte 数组中。
    public int read(byte b[], int off, int len) throws IOException {
        Object traceContext = IoTrace.fileReadBegin(path);
        int bytesRead = 0;
        try {
            bytesRead = readBytes(b, off, len);
        } finally {
            IoTrace.fileReadEnd(traceContext, bytesRead == -1 ? 0 : bytesRead);

        }
        return bytesRead;
    }

}
```

通过源码可以看到，如果用read()方法读取一个文件，每读取一个字节就要访问一次硬盘，这种读取的方式效率是很低的。即便使用read(byte b[])方法一次读取多个字节，当读取的文件较大时，也会频繁的对磁盘操作。
为了提高字节输入流的工作效率，Java提供了BufferedInputStream类。

首先解释一下BufferedInputStream的基本原理：

API文档的解释：在创建 BufferedInputStream时，会创建一个内部缓冲区数组。在读取流中的字节时，可根据需要从包含的输入流再次填充该内部缓冲区，一次填充多个字节。

也就是说，Buffered类初始化时会创建一个较大的byte数组，一次性从底层输入流中读取多个字节来填充byte数组，当程序读取一个或多个字节时，可直接从byte数组中获取，当内存中的byte读取完后，会再次用底层输入流填充缓冲区数组。

这种从直接内存中读取数据的方式要比每次都访问磁盘的效率高很多。下面通过分析源码，进一步理解其原理：

BufferedInputStream源码:

```
package java.io;
public class BufferedInputStream extends FilterInputStream {

    //缓冲区数组默认大小8192Byte,也就是8K
    private static int defaultBufferSize = 8192;
    /**
     * 内部缓冲数组，会根据需要进行填充。
     * 大小默认为8192字节，也可以用构造函数自定义大小
     */
    protected volatile byte buf[];
    /**
     * 缓冲区中还没有读取的字节数
     * 当count=0时，说明缓冲区内容已读完，会再次填充
     */
    protected int count;
    // 缓冲区指针，记录缓冲区当前读取位置
    protected int pos;
```

```
    //真正读取字节的还是InputStream
    private InputStream getInIfOpen() throws IOException {
        InputStream input = in;
        if (input == null)
            throw new IOException("Stream closed");
        return input;
    }
    //创建空缓冲区
    private byte[] getBufIfOpen() throws IOException {
        byte[] buffer = buf;
        if (buffer == null)
            throw new IOException("Stream closed");
        return buffer;
    }
    //创建默认大小的BufferedInputStream
    public BufferedInputStream(InputStream in) {
        this(in, defaultBufferSize);
    }
    //此构造方法可以自定义缓冲区大小
    public BufferedInputStream(InputStream in, int size) {
        super(in);
        if (size <= 0) {
            throw new IllegalArgumentException("Buffer size <= 0");
        }
        buf = new byte[size];
    }
    /**
     * 填充缓冲区数组
     */
    private void fill() throws IOException {
        byte[] buffer = getBufIfOpen();
        if (markpos < 0)
            pos = 0;
        //....部分源码省略
        count = pos;
        int n = getInIfOpen().read(buffer, pos, buffer.length - pos);
        if (n > 0)
            count = n + pos;
    }
    /**
     * 读取一个字节
     * 与FileInputStream中的read()方法不同的是，这里是从缓冲区数组中读取了一个字节
     * 也就是直接从内存中获取的，效率远高于前者
     */
    public synchronized int read() throws IOException {
        if (pos >= count) {
            fill();
            if (pos >= count)
                return -1;
        }
        return getBufIfOpen()[pos++] & 0xff;
    }
    //从缓冲区中一次读取多个字节
    private int read1(byte[] b, int off, int len) throws IOException {
        int avail = count - pos;
        if (avail <= 0) {
            if (len >= getBufIfOpen().length && markpos < 0) {
                return getInIfOpen().read(b, off, len);
```

```
            }
            fill();
            avail = count - pos;
            if (avail <= 0) return -1;
        }
        int cnt = (avail < len) ? avail : len;
        System.arraycopy(getBufIfOpen(), pos, b, off, cnt);
        pos += cnt;
        return cnt;
    }
}
```

2.3.3 字节缓冲输入流使用

```
package com.atguigu.buffer.in;
/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/3 17:09
 * description:字节输入缓冲流
 */
public class ByteFileInBufferStream {

    public static void main(String[] args) throws Exception{
        //1创建BufferedInputStream
        FileInputStream fis=new FileInputStream("D:\\test.txt");
        /**
         * 创建缓存流！注意：默认的缓冲区大小是8个字节！可以通过构造函数第二个参数设置！
         */
        BufferedInputStream bis=new BufferedInputStream(fis);
        //2读取
        //2.1单个字节读取
//      int data=0;
//      while((data=bis.read())!=-1) {
//          System.out.print((char)data);
//      }
        //2.2一次读取多个字节
        byte[] buf=new byte[1024];
        int count=0;
        while((count=bis.read(buf))!=-1) {
            System.out.println(new String(buf,0,count,"UTF-8"));
        }
        //3关闭
        bis.close();
        fis.close();
    }

}
```

2.3.4 字节缓冲输出流原理

FileoutputStream源码:

```
/**
     * Writes the specified byte to this file output stream.
     *
     * @param   b   the byte to be written.
     * @param   append   {@code true} if the write operation first
     *     advances the position to the end of file
     *   写出单个字节,调用c/c++完成系统磁盘交互
     */
    private native void write(int b, boolean append) throws IOException;

    /**
     * Writes the specified byte to this file output stream. Implements
     * the <code>write</code> method of <code>OutputStream</code>.
     *
     * @param      b   the byte to be written.
     * @exception  IOException  if an I/O error occurs.
     */
    public void write(int b) throws IOException {
        write(b, append);
    }

    /**
     * Writes a sub array as a sequence of bytes.
     * @param b the data to be written
     * @param off the start offset in the data
     * @param len the number of bytes that are written
     * @param append {@code true} to first advance the position to the
     *     end of file
     * @exception IOException If an I/O error has occurred.
     * 写出字节数组,调用c/c++完成系统磁盘交互
     */
    private native void writeBytes(byte b[], int off, int len, boolean append)
        throws IOException;

    /**
     * Writes <code>b.length</code> bytes from the specified byte array
     * to this file output stream.
     *
     * @param      b   the data.
     * @exception  IOException  if an I/O error occurs.
     */
    public void write(byte b[]) throws IOException {
        writeBytes(b, 0, b.length, append);
    }

    /**
     * Writes <code>len</code> bytes from the specified byte array
     * starting at offset <code>off</code> to this file output stream.
     *
     * @param      b     the data.
     * @param      off   the start offset in the data.
     * @param      len   the number of bytes to write.
     * @exception  IOException  if an I/O error occurs.
```

```
     */
    public void write(byte b[], int off, int len) throws IOException {
        writeBytes(b, off, len, append);
    }
```

BufferedOutputStream源码:

```
public
class BufferedOutputStream extends FilterOutputStream {
    /**
     *  输出字节缓存区
     */
    protected byte buf[];

    /**
        缓存区有效字节长度!
     */
    protected int count;

    /**
      * 初始化字节缓存流对象,默认缓存区 8字节
     */
    public BufferedOutputStream(OutputStream out) {
        this(out, 8192);
    }

    /**
       初始化字节缓冲流对象,指定缓冲区大小!
     */
    public BufferedOutputStream(OutputStream out, int size) {
        super(out);
        if (size <= 0) {
            throw new IllegalArgumentException("Buffer size <= 0");
        }
        buf = new byte[size];
    }


    /** 将缓存区数据写到磁盘 */
    private void flushBuffer() throws IOException {
        if (count > 0) {
            out.write(buf, 0, count);
            count = 0;
        }
    }

    /**
     *  将当自己写到缓存区!
     *  如果缓冲区有效字节数量大于或者等于缓存区大小!证明缓存区没有位置!
     *   开始写入磁盘! 否则写到缓存区!
     */
    public synchronized void write(int b) throws IOException {
        if (count >= buf.length) {
            flushBuffer();
        }
        buf[count++] = (byte)b;
```

```
    }


    /**
        写入多字节,如果写入长度大于缓存区,直接写出!也将缓存区写出!
        没有超过写到缓存区!
        增加缓存区有效字节长度为本次添加长度!
     */
    public synchronized void write(byte b[], int off, int len) throws
IOException {
        if (len >= buf.length) {
            /* If the request length exceeds the size of the output buffer,
               flush the output buffer and then write the data directly.
               In this way buffered streams will cascade harmlessly. */
            flushBuffer();
            out.write(b, off, len);
            return;
        }
        if (len > buf.length - count) {
            flushBuffer();
        }
        System.arraycopy(b, off, buf, count, len);
        count += len;
    }

    /**
          缓存区写到本地磁盘!
     */
    public synchronized void flush() throws IOException {
        flushBuffer();
        out.flush();
    }
}
```

2.3.5 字节缓冲输出流使用

```
package com.atguigu.buffer.out;

import java.io.BufferedOutputStream;
import java.io.FileOutputStream;

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/3 17:22
 * description: 字节缓存输出流
 */
public class ByteFileOutBufferStream {

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

        FileOutputStream fos=new FileOutputStream("D:\\buffer.txt");
        BufferedOutputStream bos=new BufferedOutputStream(fos);
        //2写入文件
```

```
        for(int i=0;i<10;i++) {
            //写入8K缓冲区
            bos.write("helloworld\r\n".getBytes());
        }

        bos.flush();//刷新到硬盘

        //3关闭(内部调用flush方法)
        bos.close();
        fos.close();
    }

}
```

2.3.6 性能提升对比

> 对比下有缓冲流和没用的性能对比

```
package com.atguigu.buffer.comparison;

import org.junit.Test;

import java.io.BufferedInputStream;
import java.io.FileInputStream;

/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/3 17:45
 * description:普通读取和缓冲读取的性能对比
 */
public class BufferStream {

    /**
     * 测试普通读取文件，一个字节一个字节
     */
    @Test
    public void testCommons() throws Exception{

        long start = System.currentTimeMillis();

        FileInputStream fis = new FileInputStream("D:\\test.txt");

        int data = -1;

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


        fis.close();

        long end = System.currentTimeMillis();
```

```
        System.out.println("消耗时间："+(end-start));

        //91毫秒
    }


    /**
     * 测试普通读取文件，一个字节一个字节
     */
    @Test
    public void testBUffer() throws Exception{

        long start = System.currentTimeMillis();

        FileInputStream fis = new FileInputStream("D:\\test.txt");

        BufferedInputStream bfs = new BufferedInputStream(fis);

        int data = -1;

        while ((data = bfs.read())!= -1) {
            System.out.print("(char)data = " + (char)data);
        }

        bfs.close();
        fis.close();

        long end = System.currentTimeMillis();

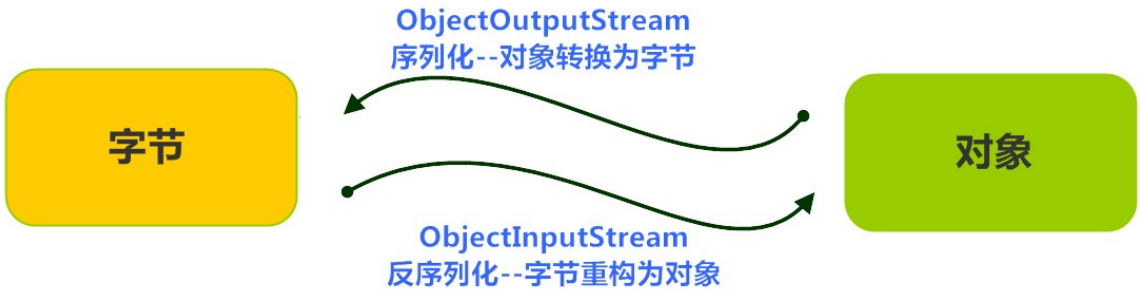
        System.out.println("消耗时间："+(end-start));

        //30毫秒
    }

}
```

2.4 对象流和序列化

> Java 提供了一种对象**序列化**的机制。用字节序列可以表示一个对象，该字节序列包含该`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：
>
> 序列化是一种非常重要运行内存数据的缓存机制!



思考：序列化、反序列化有什么作用和场景呢？

2.4.1 对象序列化

> 案例演示：使用对象流实现序列化和反序列化。

```
public class Student implements Serializable{

    /**
     * serialVersionUID:序列化版本号ID,
     */
    private static final long serialVersionUID = 100L;
    private String name;
    private transient int age;

    public static String country="中国";

    public Student() {
        // TODO Auto-generated constructor stub
    }
    public Student(String name, int age) {
        super();
        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 "Student [name=" + name + ", age=" + age + "]";
    }
}
```

> 序列化，对象输出流！

```
public class TestSerializable {
    public static void main(String[] args) throws Exception{
        //1创建对象流
        FileOutputStream fos=new FileOutputStream("d:\\stu.bin");
        ObjectOutputStream oos=new ObjectOutputStream(fos);
        //2序列化(写入操作)
        Student zhangsan=new Student("张三", 20);
        Student lisi=new Student("李四", 22);
        ArrayList<Student> list=new ArrayList<>();
        list.add(zhangsan);
        list.add(lisi);
        oos.writeObject(list);

        //3关闭
        oos.close();
        System.out.println("序列化完毕");
    }
}
```

> 反序列化，对象输入流!

```
public class TestDeSerializable  {
    public static void main(String[] args) throws Exception {
        //1创建对象流
        FileInputStream fis=new FileInputStream("d:\\stu.bin");
        ObjectInputStream ois=new ObjectInputStream(fis);
        //2读取文件(反序列化)
//      Student s=(Student)ois.readObject();
//      Student s2=(Student)ois.readObject();
        ArrayList<Student> list=(ArrayList<Student>)ois.readObject();
        //3关闭
        ois.close();
        System.out.println("执行完毕");
//      System.out.println(s.toString());
//      System.out.println(s2.toString());
        System.out.println(list.toString());
    }
}
```

> 对象序列化的细节：
>
> - 必须实现Serializable接口。
> - 显示声明序列化id!
> - 必须保证其所有属性均可序列化。
> - transient修饰为临时属性，不参与序列化。
> - 读取到文件尾部的标志：java.io.EOFException。

三、字符编码

3.1 字节

> 所谓字节(Byte)，是计算机数据存储的一种计量单位。一个二进制位称为比特(bit)，8个比特组成一个字节，也就是说一个字节可以用于区分256个整数(0~255)。由此我们可以知道，字节本是面向计算机数据存储及传输的基本单位，后续的字符也就是以字节为单位存储的，不同编码的字符占用的字节数不同。
>
> 那么在Java中，除了存储的意义外，Java还将字节Byte作为一种基本数据类型，该数据类型在内存中占用一个字节，用于(-128~127)范围内的整数

```
byte a = -128;
byte b = 127;
```

总的来说，字节在Java中有两种含义：

- 存储的单位
- Java的数据类型，用于表示-128~127范围的整数

3.2 字符

> 计算机底层存储的是字节，字符的设计则是用于展示符号。屏幕上显示的各种文字，数字，符号等就是解码的字符。所以我们说字符是用来显示的符号，它将存储的字节转换成人们看得懂的符号，因此字符的核心就是定义字节与展示符号之间的关系，这种映射关系通常也叫做编码。

3.3 为什么有多种编码方式

> 为什么要编码呢？前面我们知道数据都是以字节为单位存储在计算机中，字节可以区分256个整数，最容易想到的就是将这256个整数定义为256种状态并分别对应256个字符。但是人类符号太多了，256种是不够的。所以人们想到将多个字节合并起来表示人类语言符号，编码的问题就转化成了字节的组合问题。
>
> 由于语种太多，也造就了多种的编码方式！

3.4 常见编码格式

> 如今有很多编码格式，常见的如ASCII、ISO-8859-1、GB2312、GBK、UTF-8、UTF-16等等。

编码	说明
ASCII	编码是最基础的编码格式，标准的ASCII码一共有128个，占用字节的低7位，将英语系语种的符号都能覆盖住，但是总的来说能表示的字符还是非常有限。
ISO-8859-1	收录除ASCII外，还包括西欧、希腊语、泰语、阿拉伯语、希伯来语对应的文字符号。
GB2312	UTF-32编码使用4个字节，也就是32位二进制存储Unicode字符，效率高但是空间浪费。是双字节编码，意味着它使用两个字节来表示符号，包含有6763个汉字。包含简体中文！
GBK	UTF-8编码是一种变长的编码方式，它使用1～6个字节来存储，对于英语系的字符使用一个字节，向下兼容ASCII，对于汉字则使用两个字节，依次类推，这样就能够节省一定的空间。是GB2312的一个扩展，也是双字节编码，能够表示21003个汉字，且向下兼容GB2312。
BIG5	UTF-16编码是介于两者之间的一种编码方式。对于部分字符采用2个字节，另一部分字符采用4个字节。因此UTF-16无法兼容ASCII。台湾省常用编码，收录的是繁体中文。
UTF-8	编码的规范越来越多，不同语言的国家都定义了自己的语言符号编码标准，一时间编码标准百花齐放，在互联网的时代里交流十分不便，不同编码体系之间的信息交流都需要采用不同的解码方案，不然就会出现乱码的现象。 于是国际标准化组织ISO制定了一个能够容纳世界上所有文字和符号的字符编码方案Unicode。Unicode是一个字符集，它规定了人类所有字符对应的二进制数，至于这个二进制数怎么存储则是由开发者来进行实现。其中比较流行的实现是UTF-8和UTF-16，还有一种UTF-32。

> 扩展Unicode：
>
> UTF-32编码使用4个字节，也就是32位二进制存储Unicode字符，效率高但是空间浪费。
>
> UTF-8编码是一种变长的编码方式，它使用1～6个字节来存储，对于英语系的字符使用一个字节，向下兼容ASCII，对于汉字则使用两个字节，依次类推，这样就能够节省一定的空间。
>
> UTF-16编码是介于两者之间的一种编码方式。对于部分字符采用2个字节，另一部分字符采用4个字节。因此UTF-16无法兼容ASCII。
>
> 在平时的使用中，UTF-8的使用还是比较多，就是由于它既能向下兼容ASCII，还能够在一定程度上节省空间。

注：当编码方式和解码方式不一致时，会出现乱码。

四、字符流【重点】

> 字符流：就是在字节流的基础上，加上编码，形成的数据流
>
> 字符流出现的意义：因为字节流在操作字符时，可能会有中文导致的乱码，所以由字节流引申出了字符流。
>
> 如果是文本或者其他字符类型文件,推荐使用字符流读取!

4.1 字符抽象类

> Reader：字符输入流 所有字符输入流的超类
>
> - public int read(){}。
> - 读一个字符 该方法将阻塞，直到字符可用
> - 返回结果 0-65535,如果没有数据返回 -1;
> - public int read(char[] c){}。
>   - 将字符读入数组。 该方法将阻塞，直到某些输入可用!
>   - 返回结果为读取的字符数,如果没有更多返回-1!
> - public int read(char[] b,int off,int len){}。
>   - 读取字符到数组,该方法阻塞
>   - offset为 b数组的起始偏移量
>   - len代表本次读取的长度

> Writer：字符输出流 字符输出流的父类
>
> - public void write(int n){}。
>   - 写出一个字符
> - public void write(String str){}。
>   - 写出一个字符串
> - public void write(char[] c,int offset,int len){}。
>   - 写出一个字符数组
>   - offset数组写出的偏移量
>   - len写出的长度
> - public void write(char[] c){}。
>   - 写出一个字符数组

4.2 字符节点流 [字符流实现类]

> FileReader： 文件字符输入流
>
> - FileReader(File file)`创建一个新的`FileReader`，给定要读取的文件的名称。
> - FileReader(String fileName)`创建一个新的`FileReader`，给定要读取的文件的名称。

> FileWriter：文件字符输出流
>
> - `FileWriter(File file,boolean append)` 给一个File对象构造一个FileWriter对象。
> - `FileWriter(File file)` 给一个File对象构造一个FileWriter对象。
> - `FileWriter(String fileName)` 给一个File对象构造一个FileWriter对象。
> - `FileWriter(String fileName,boolean append)` 给一个File对象构造一个FileWriter对象。

4.2.1 字符文件输入流

```
package com.atguigu.charstream;

import java.io.FileReader;

/**
 * projectName: demos
 *
 * @author: 赵伟风
```

```
 * time: 2022/3/3 18:33
 * description:字符输入流
 */
public class FileCharInputStream {

    public static void main(String[] args) throws Exception {
        //1创建FileReader 文件字符输入流
        FileReader fr=new FileReader("d:\\test.txt");
        //获取默认的默编码格式!
        String encoding = fr.getEncoding();
        System.out.println("encoding = " + encoding);

        //2读取
        //2.1单个字符读取
//      int data=0;
//      while((data=fr.read())!=-1) {//读取一个字符
//          System.out.print((char)data);
//      }
        //2.2多字符读取
        char[] buf=new char[1024];
        int count=0;
        while((count=fr.read(buf))!=-1) {
            System.out.println(new String(buf, 0, count));
        }

        //3关闭
        fr.close();
    }
}
```

4.2.2 字符文件输出流

```
/**
 * projectName: demos
 *
 * @author: 赵伟风
 * time: 2022/3/3 18:33
 * description:字符输出流
 */
public class FileCharOutputStream {

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

        //1创建FileWriter对象
        FileWriter fw=new FileWriter("d:\\write.txt");
        //2写入
        for(int i=0;i<10;i++) {
            fw.write("java是世界上最好的语言\r\n");
        }
        //3关闭
        fw.close();
        System.out.println("执行完毕");

    }
}
```

4.3 字符缓冲流介绍和作用

1. 字符缓存流分类
    1. BufferedWriter 字符输出缓冲流

> - 将文本写入字符输出流，缓冲字符，以提供单个字符，数组和字符串的高效写入。
>
> 可以指定缓冲区大小，或者可以接受默认大小。 默认值足够大，可用于大多数用途。
>
> 提供了一个newLine（）方法，它使用平台自己的系统属性`line.separator`定义的行分隔符概念。 并非所有平台都使用换行符（'\ n'）来终止行。 因此，调用此方法来终止每个输出行，因此优选直接写入换行符。
>
> 一般来说，Writer将其输出立即发送到底层字符或字节流。 除非需要提示输出，否则建议将BufferedWriter包装在其write（）操作可能很昂贵的Writer上，例如FileWriters和OutputStreamWriters。 例如，
>
> ```
>  PrintWriter out
>   = new PrintWriter(new BufferedWriter(new
> FileWriter("foo.out")));
> ```
>
> 将缓冲PrintWriter的输出到文件。没有缓冲，每次调用print（）方法都会使字符转换为字节，然后立即写入文件，这可能非常低效。

    2. BufferedReader 字符串输入缓冲流

> - ```
> public class BufferedReader
>   extends Reader
> ```
>
> 从字符输入流读取文本，缓冲字符，以提供字符，数组和行的高效读取。
>
> 可以指定缓冲区大小，或者可以使用默认大小()。 默认值足够大，可用于大多数用途。
>
> 通常，由读取器做出的每个读取请求将引起对底层字符或字节流的相应读取请求。 因此，建议将`BufferedReader`包装在其`read`（）操作可能昂贵的读取器上，例如`FileReaders`和`InputStreamReaders`。 例如，
>
> ```
> BufferedReader in
>  = new BufferedReader(new FileReader("foo.in"));
> ```
>
> 将缓冲指定文件的输入。没有缓冲，每次调用`read`（）或`readLine`（）可能会导致从文件中读取字节，转换成字符，然后返回，这可能非常低效。

2. 字符缓冲优势
    1. 有读写缓冲区[长度为8192],避免频繁转码和磁盘操作提升效率
    2. 字符输出缓冲流提供newLine(),可以快速指定新行字符
    3. 字符输入缓冲流提供readLine();可以快速按行读取字符

4.3.1 字符缓冲输入流

```
public class TestBufferedReader {
    public static void main(String[] args) throws Exception{
        //1创建缓冲流
        FileReader fr=new FileReader("d:\\write.txt");
        BufferedReader br=new BufferedReader(fr);
        //2读取
        //2.1第一种方式
//      char[] buf=new char[1024];
//      int count=0;
//      while((count=br.read(buf))!=-1) {
//          System.out.print(new String(buf,0,count));
//      }
        //2.2第二种方式，一行一行的读取
        String line=null;
        while((line=br.readLine())!=null) {
            System.out.println(line);
        }

        //3关闭
        br.close();
    }
}
```

4.3.2 字符缓冲输出流

```
public class TestBufferedWriter {
    public static void main(String[] args) throws Exception{
        //1创建BufferedWriter对象
        FileWriter fw=new FileWriter("d:\\buffer.txt");
        BufferedWriter bw=new BufferedWriter(fw);
        //2写入
        for(int i=0;i<10;i++) {
            bw.write("好好学习，天天向上");
            bw.newLine();//写入一个换行符 windows \r\n  linux  \n
        }

    bw.flush();
        //3关闭
        bw.close();
        System.out.println("执行完毕");

    }
}
```

4.4 打印流

> PrintWriter：打印流，只有输出流！没有输入流！

- ```
  public PrintWriter(String fileName) throws FileNotFoundException {
      this(new BufferedWriter(new OutputStreamWriter(new
  FileOutputStream(fileName))),
           false);
  ```
  ```

> - 封装了print() / println()方法，支持写入后换行。
> - 支持数据原样打印[obj.toString();]。
> - 内部自动包裹缓冲流.
> - 对比FilterWriter
>
>   FileWriter获取Writer：获得字符流，编码为系统默认
>
>   PrintWriter获取Writer：获得字符流，编码为系统默认，同时增加写八种基本类型、字符串、对象以及缓冲区的功能

```
public class TestPrintWriter {
    public static void main(String[] args) throws Exception {
        //1创建打印流
        PrintWriter pw=new PrintWriter("d:\\print.txt");
        //2打印
        pw.println(97);
        pw.println(true);
        pw.println(3.14);
        pw.println('a');
        //3关闭
        pw.close();
        System.out.println("执行完毕");
    }
}
```

4.5 转换流

> 转换流：InputStreamReader/OutputStreamWriter
>
> - 可将字节流转换为字符流。
> - 可设置字符的编码方式。[注意,编码格式必须和文件相同否则乱码]
>
> 在idea中，使用`FileReader`读取项目中的文本文件。由于idea的设置UTF-8编码但是，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。
>
> 这种场景下，就需要转换流!
>
> 转换流不会改变文件原有编码格式,只是改变开发工具中读取的默认编码编码格式而已!

> 案例演示:

```
public class TestInputStreamReader {
    public static void main(String[] args) throws Exception {
        //1创建InputStreamReader对象
        FileInputStream fis=new FileInputStream("d:\\write.txt");
        InputStreamReader isr=new InputStreamReader(fis, "gbk");
        //2读取文件
        int data=0;
        while((data=isr.read())!=-1) {
            System.out.print((char)data);
        }
        //3关闭
        isr.close();
    }
}
```

```
public class TestOutputStreamWriter {
    public static void main(String[] args) throws Exception{
        //1创建OutputStreamWriter
        FileOutputStream fos=new FileOutputStream("d:\\info.txt");
        OutputStreamWriter osw=new OutputStreamWriter(fos, "utf-8");
        //2写入
        for(int i=0;i<10;i++) {
            osw.write("我爱北京，我爱故乡\r\n");
            osw.flush();
        }
        //3关闭
        osw.close();
        System.out.println("执行成功");
    }
}
```

五、File、FileFilter

5.1 File类

> 概念：代表物理盘符中的一个文件或者文件夹。

> 常见方法：

方法名	描述
createNewFile()	创建一个新文件。
mkdir()	创建一个新目录。
delete()	删除文件或空目录。
exists()	判断File对象所对象所代表的对象是否存在。
getAbsolutePath()	获取文件的绝对路径。
getName()	取得名字。
getParent()	获取文件/目录所在的目录。
isDirectory()	是否是目录。
isFile()	是否是文件。
length()	获得文件的长度。
listFiles()	列出目录中的所有内容。
renameTo()	修改文件名为。

> 案例演示：

```
public class TestFile {
    public static void main(String[] args) throws Exception {
        //separator();
        //fileOpe();
        directoryOpe();
```

```
    }
    //（1）分隔符
    public static void separator() {
        System.out.println("路径分隔符"+File.pathSeparator);
        System.out.println("名称分隔符"+File.separator);
    }
    //（2）文件操作
    public static void fileOpe() throws Exception {
        //1创建文件 createNewFile()
        File file=new File("d:\\file.txt");
        //System.out.println(file.toString());
        if(!file.exists()) {
            boolean b=file.createNewFile();
            System.out.println("创建结果:"+b);
        }
        //2删除文件
        //2.1直接删除
        //System.out.println("删除结果:"+file.delete());
        //2.2使用jvm退出时删除
//      file.deleteOnExit();
//      Thread.sleep(5000);

        //3获取文件信息
        System.out.println("获取文件的绝对路径:"+file.getAbsolutePath());
        System.out.println("获取路径:"+file.getPath());
        System.out.println("获取文件名称:"+file.getName());
        System.out.println("获取父目录:"+file.getParent());
        System.out.println("获取文件长度:"+file.length());
        System.out.println("文件创建时间:"+new
Date(file.lastModified()).toLocaleString());


        //4判断
        System.out.println("是否可写:"+file.canWrite());
        System.out.println("是否时文件:"+file.isFile());
        System.out.println("是否隐藏:"+file.isHidden());

    }

    //（3）文件夹操作
    public static void directoryOpe() throws Exception{
        //1 创建文件夹
        File dir=new File("d:\\aaa\\bbb\\ccc");
        System.out.println(dir.toString());
        if(!dir.exists()) {
            //dir.mkdir();//只能创建单级目录
            System.out.println("创建结果:"+dir.mkdirs());//创建多级目录
        }

        //2 删除文件夹
        //2.1直接删除(注意删除空目录)
        //System.out.println("删除结果:"+dir.delete());
        //2.2使用jvm删除
//      dir.deleteOnExit();
//      Thread.sleep(5000);
        //3获取文件夹信息
        System.out.println("获取绝对路径: "+dir.getAbsolutePath());
        System.out.println("获取路径:"+dir.getPath());
```

```
        System.out.println("获取文件夹名称："+dir.getName());
        System.out.println("获取父目录："+dir.getParent());
        System.out.println("获取创建时间:"+new
Date(dir.lastModified()).toLocaleString());


        //4判断
        System.out.println("是否时文件夹:"+dir.isDirectory());
        System.out.println("是否时隐藏："+dir.isHidden());

        //5遍历文件夹
        File dir2=new File("d:\\图片");
        String[] files=dir2.list();
        System.out.println("---------------------------------");
        for (String string : files) {
            System.out.println(string);
        }

    }

}
```

5.2 FileFilter接口

> FileFilter：文件过滤器接口
>
> - boolean accept(File pathname)。
> - 当调用File类中的listFiles()方法时，支持传入FileFilter接口接口实现类，对获取文件进行过滤，只有满足条件的文件的才可出现在listFiles()的返回值中。

> 案例演示：过滤所有的.jpg图片。

```
public class TestFileFilter{
    public static void main(String[] args){
        File dir=new File("d:\\图片");
        File[] files2=dir.listFiles(new FileFilter() {

            @Override
            public boolean accept(File pathname) {
                if(pathname.getName().endsWith(".jpg")) {
                    return true;
                }
                return false;
            }
        });
        for (File file : files2) {
            System.out.println(file.getName());
        }
    }
}
```

六、Properties实现流操作

> Properties：属性集合。

> 特点：
>
> - 存储属性名和属性值。
> - 属性名和属性值都是字符串类型。
> - 没有泛型。
> - 和流有关。

> 案例演示：Properties实现流操作。

```
public class TestProperties {
    public static void main(String[] args) throws Exception {
        //1创建集合
        Properties properties=new Properties();
        //2添加数据
        properties.setProperty("username", "zhangsan");
        properties.setProperty("age", "20");
        System.out.println(properties.toString());
        //3遍历
        //3.1-----keySet----略
        //3.2-----entrySet----略
        //3.3-----stringPropertyNames()---
        Set<String> pronames=properties.stringPropertyNames();
        for (String pro : pronames) {
            System.out.println(pro+"====="+properties.getProperty(pro));
        }
        //4和流有关的方法
        //----------list方法---------
        PrintWriter pw=new PrintWriter("d:\\print.txt");
        properties.list(pw);
        pw.close();

        //----------2store方法 保存-----------
        FileOutputStream fos=new FileOutputStream("d:\\store.properties");
        properties.store(fos, "注释");
        fos.close();

        //----------3load方法 加载-------------
        Properties properties2=new Properties();
        FileInputStream fis=new FileInputStream("d:\\store.properties");
        properties2.load(fis);
        fis.close();
        System.out.println(properties2.toString());

    }
}
```