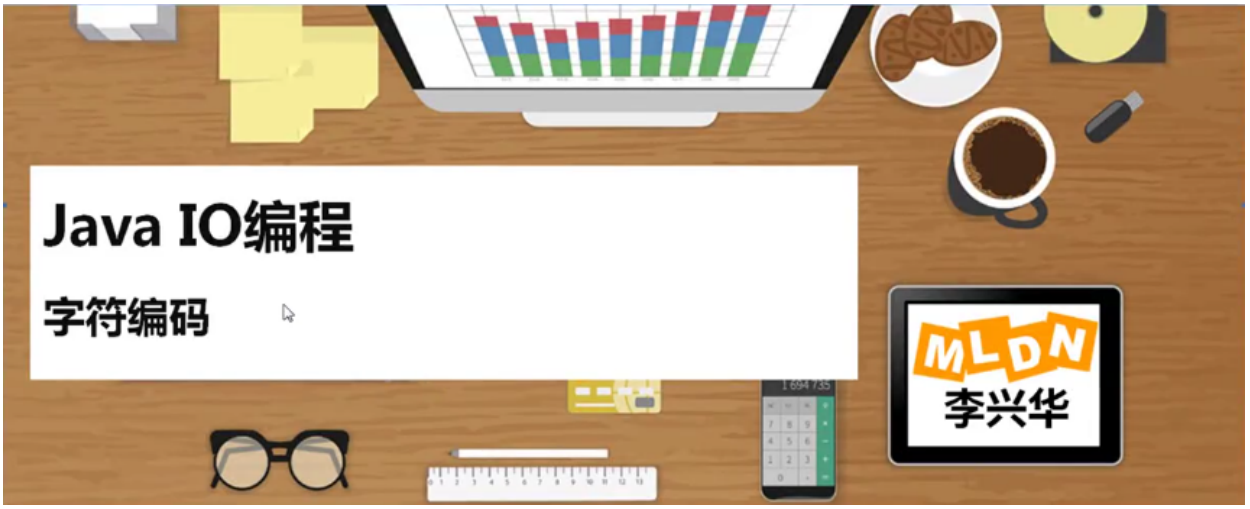


博客： https://www.cnblogs.com/HOsystem/p/14116443.html

2、具体内容

在计算机的世界里面只认0、1的数据，如果要想描述一些问兔子的编码就需要对这些二进制的数据进行组合，所以才有了现在可以看见的中文，但是在进行编码的时候如果要想正确显示出内容则一定需要有解码，所以编码和解码肯定要采用统一的标准，那么如果不统一的时候就会出现乱码。

那么在实际的开发之中对于常用的编码有如下几种：

·GBK/GB2312：国标编码，可以描述中文信息，其中GB2312只描述简体中文，而GBK包含有简体中文与繁体中文；

·ISO8859-1：国际通用编码，可以用其描述所有的字母信息，如果是象形文字则需要进行编码处理；

·UNICODE编码：采用十六进制的方式存储，可以描述所有的文字信息；

·UTF编码：象形文字部分使用十六进制编码，而普通的字母采用的是ISO8859-1编码，它的优势在于适合于快速的传输，节约带宽，也就成为了我们在开发之中首选的编码，主要使用“UTF-8”编码；

如果要想知道当前系统中支持的编码规则，则可以采用如下代码列出全部的本机属性：

范例：列出本机属性

```
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
        System.getProperties().list(System.out);
    }
}
```

部分信息：	【文件路径分割符】file.separator=\
	【文件默认编码】file.encoding=UTF-8

也就是说如果现在什么都不设置的话，则采用的编码就是UTF-8

范例：编写程序

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

```
import java.io.FileOutputStream;
import java.io.OutputStream;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		OutputStream output = new FileOutputStream("D:" + File.separator + "mldn.txt");
		output.write("中华人民共和国万岁".getBytes("UTF-8"));
		output.close();
	}
}
```

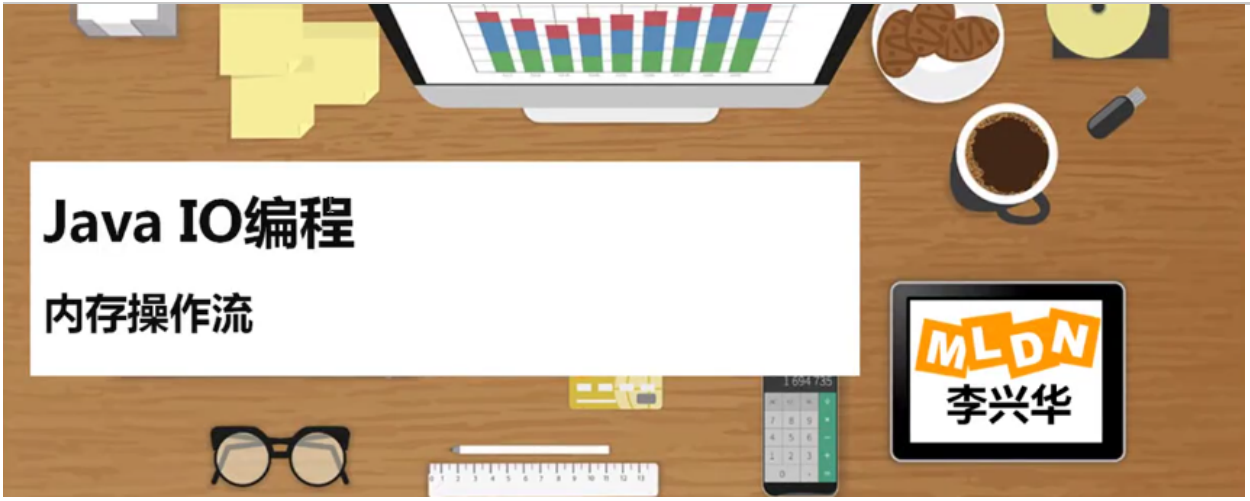
```
output.write("中华人民共和国万岁".getBytes("UTF-8"));
```

此时为默认的处理操作，不设置编码的时候就采用默认的编码方式进行。

范例：强制性设置编码

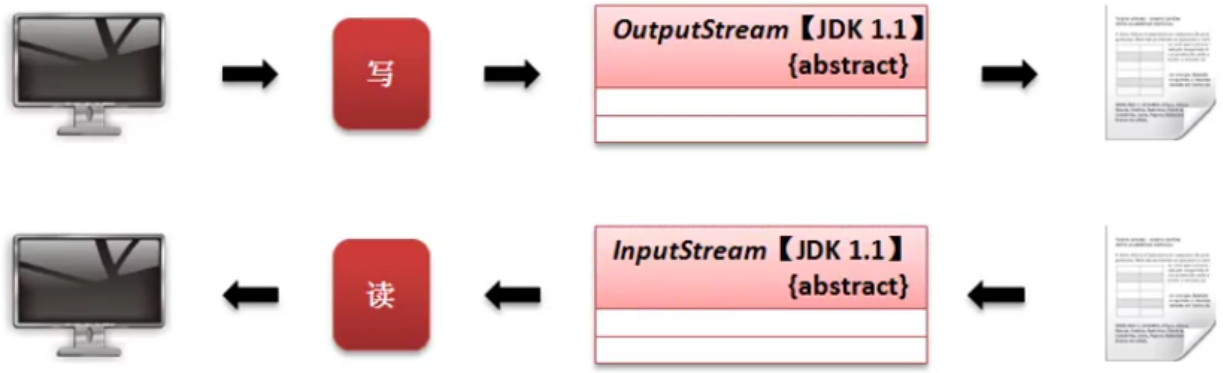
```
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		OutputStream output = new FileOutputStream("D:" + File.separator + "mldn.txt");
		output.write("中华人民共和国万岁".getBytes("ISO8859-1"));
		output.close();
	}
}
```

项目中出现的乱码问题就是编码和解码标准不统一，而最好的解决乱码的方式，所有的编码都是用UTF-8。



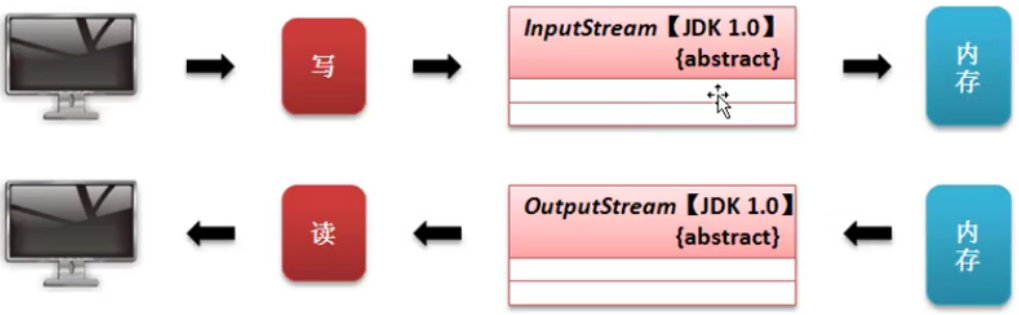
2、具体内容

在之前使用的全部都是文件操作流，文件操作流的特点，程序利用InputStream读取文件内容，而后程序利用OutputStream向文件输出内容，所有的操作都是以文件为终端的。



假设说现在需要实现IO操作，可是又不希望产生文件（临时文件）则就可以以内存为终端进行处理，这个时候的流程：

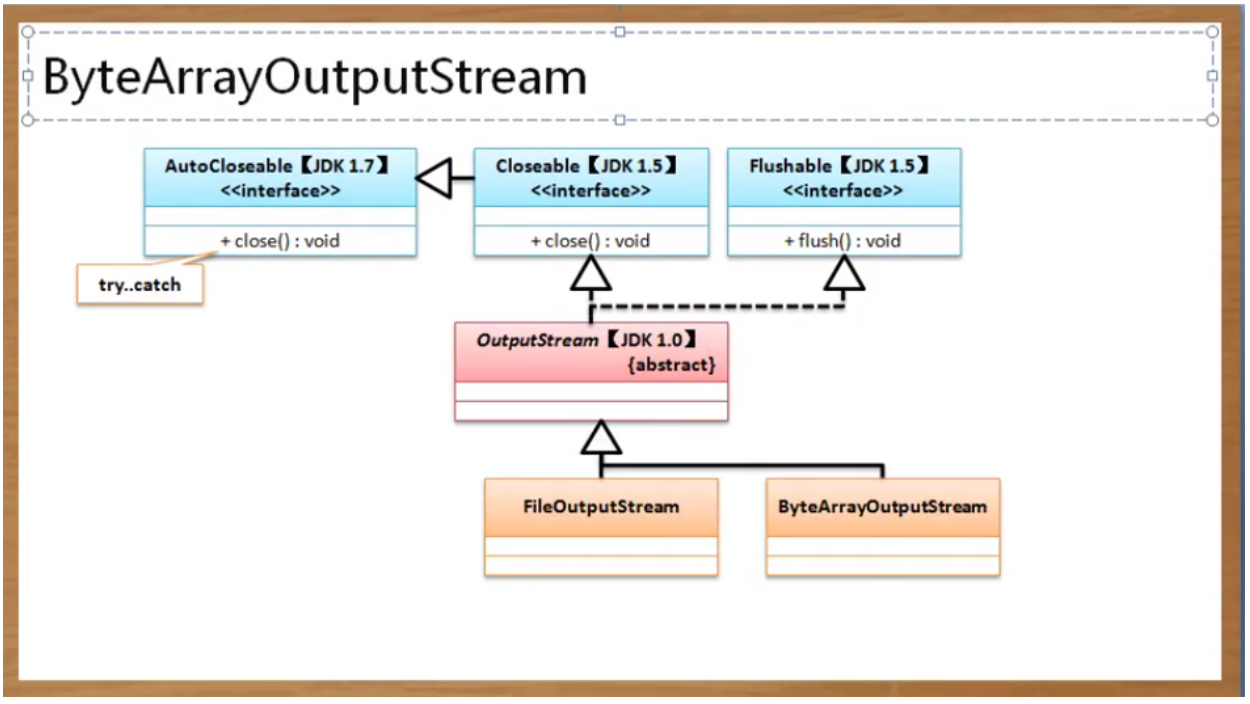
内存流



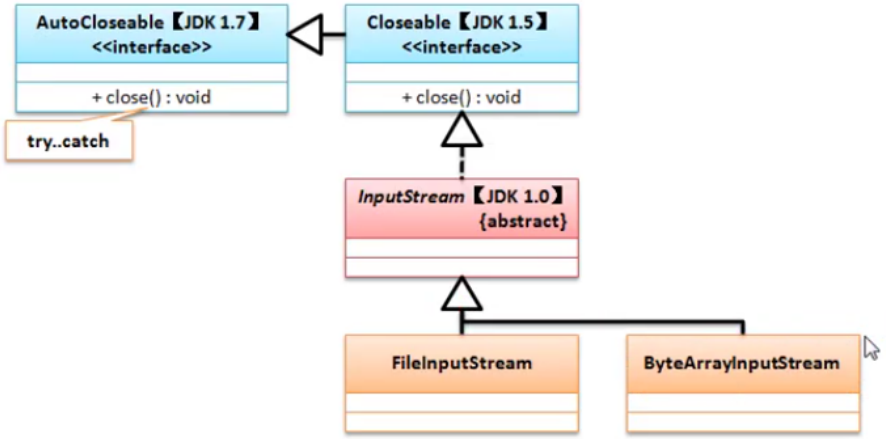
在Java里面提供有两类的内存操作流：

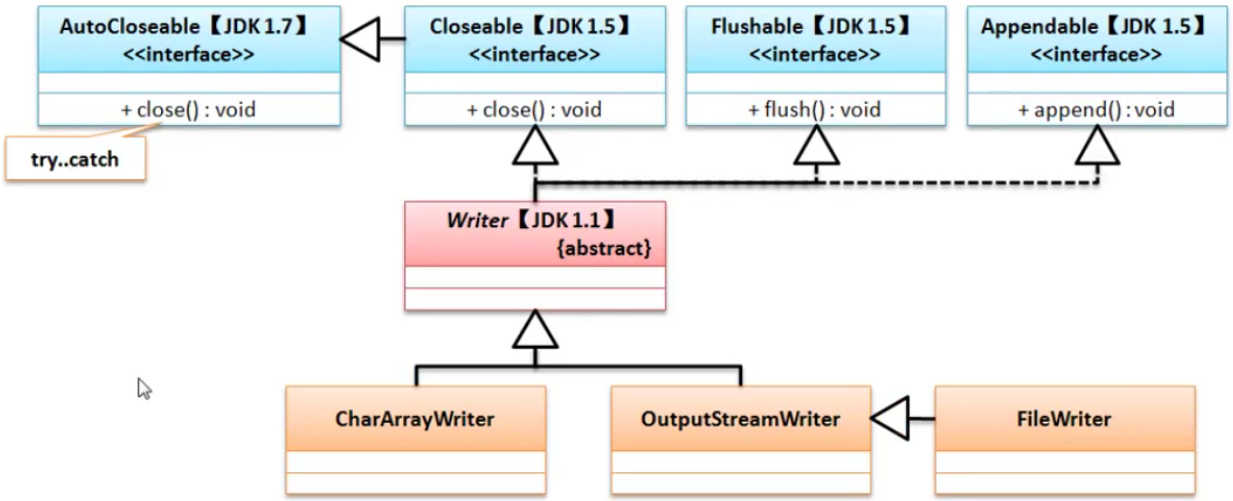
·字节内存操作流：ByteArrayOutputStream、ByteArrayInputStream；

·字符内存操作流：CharArrayWriter、CharArrayReader；

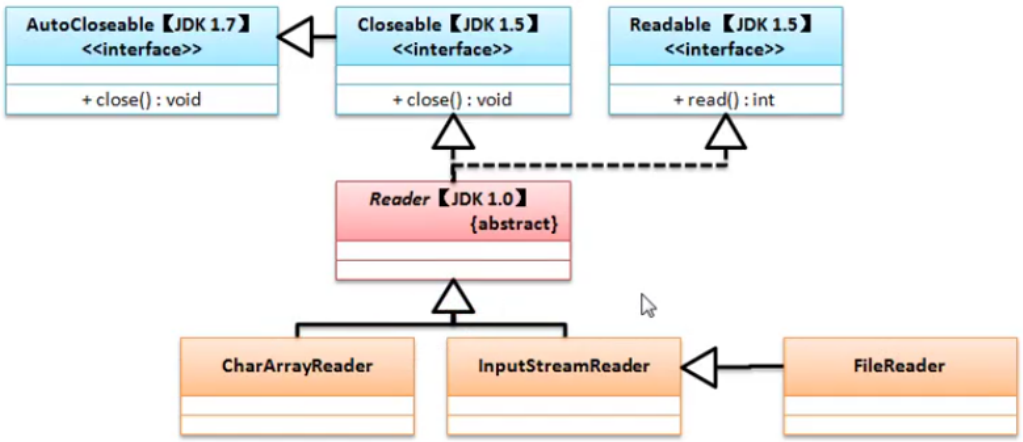


ByteArrayInputStream





CharArrayReader



下面以ByteArrayOutputStream和ByteArrayInputStream类为主进行内存的使用分析，首先来分析各自的构造方法：

·**ByteArrayInputStream构造**：public ByteArrayInputStream(byte[] buf);

·**ByteArrayOutputStream构造**：public ByteArrayOutputStream();

在ByteArrayOutputStream类里面有一个重要的方法，这个方法可以获取全部保存在内存流中的数据信息，该方法为：

·获取数据：public byte[] toByteArray();

·使用字符串的形式来获取：public String toString();

范例：利用内存流实现一个小写字母转大写字母的操作

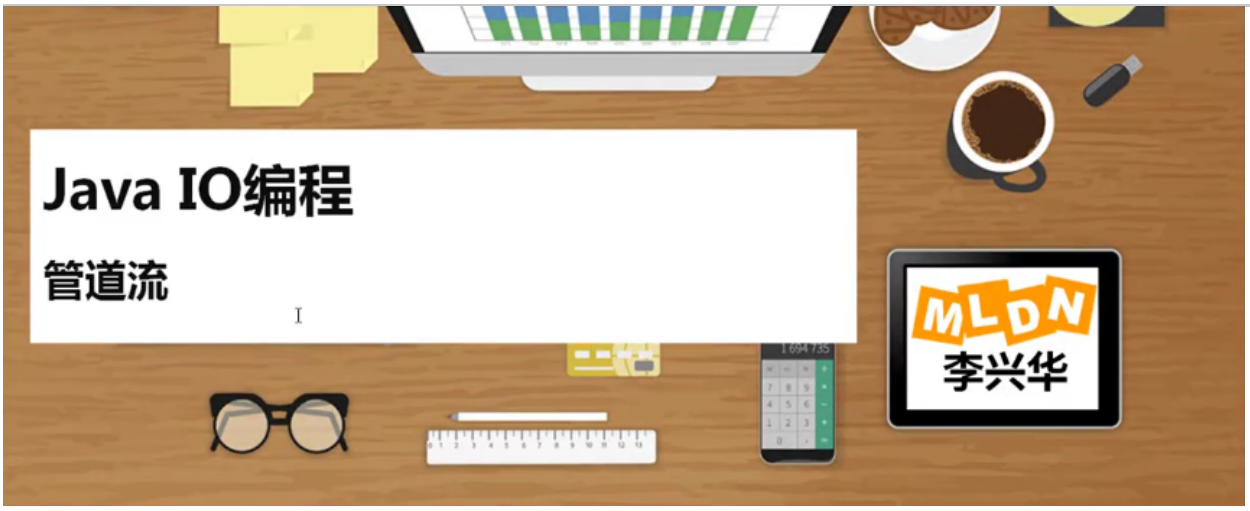
```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		String str = "www.mldn.cn" ;  // 小写字母
		InputStream input = new ByteArrayInputStream(str.getBytes()) ; // 将数据保存在内存流
		OutputStream output = new ByteArrayOutputStream() ;	// 读取内存中的数据
		int data = 0 ;
		while ((data = input.read()) != -1) {		// 每次读取一个字节
			output.write(Character.toUpperCase(data)); // 保存数据
		}
		System.out.println(output);
		input.close();
		output.close();
	}
}
```

如果现在不希望只是以字符串的形式返回，因为可能存放的是其它二进制的数据，那么此时就可以利用ByteArrayOutputStream子类的扩展功能获取全部字节数据。

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		String str = "www.mldn.cn" ;  // 小写字母
		InputStream input = new ByteArrayInputStream(str.getBytes()) ; // 将数据保存在内存流
		// 必须使用子类来调用子类自己的扩展方法
		ByteArrayOutputStream output = new ByteArrayOutputStream() ;	// 读取内存中的数据
		int data = 0 ;
		while ((data = input.read()) != -1) {		// 每次读取一个字节
			output.write(Character.toUpperCase(data)); // 保存数据
		}
		byte result [] = output.toByteArray() ; // 获取全部数据
```

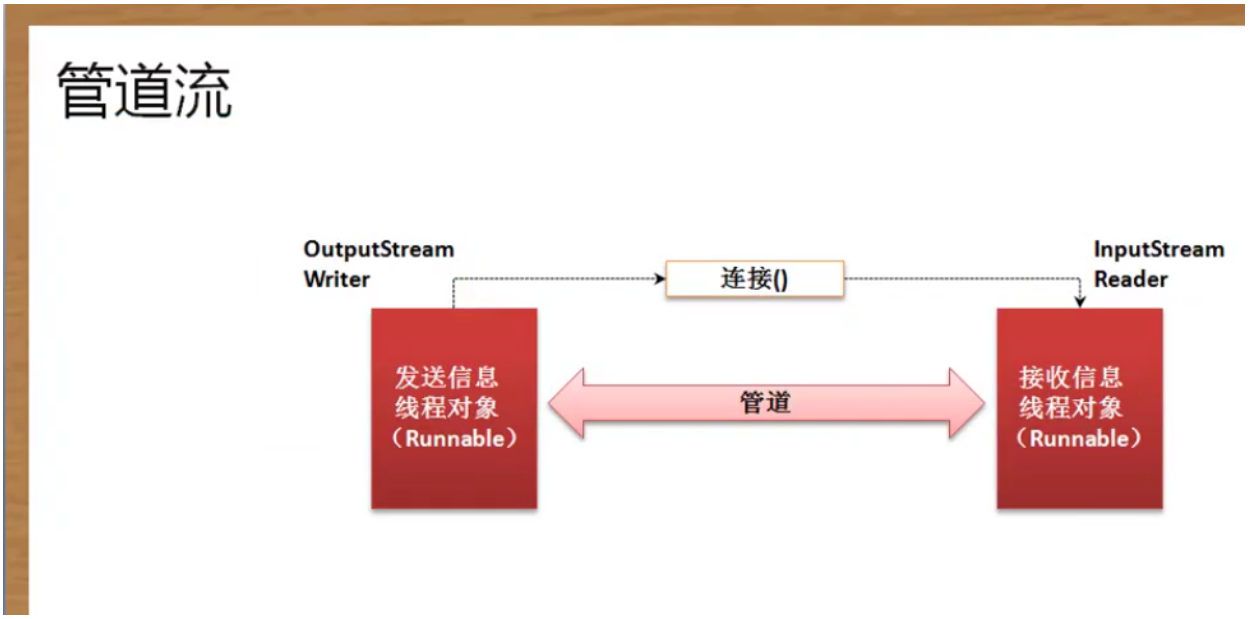
```
            System.out.println(new String(result));  // 自己处理字节数据
            input.close();
            output.close();
        }
}
```

在最初的时候可以利用ByteArrayOutputStream实现大规模文本文件的读取。



2、具体内容

管道流主要的功能是实现两个线程之间的IO处理操作。



对于管道流也是分为两类：

·**字节管道流**：PipedInputStream、PipedOutputStream；

|- 连接处理：public void connect(PipedInputStream snk) throws IOException；

·**字符管道流**：PipedReader、PipedWriter；

|- 连接处理：public void connect(PipedWriter src) throws IOException；

范例：实现管道操作

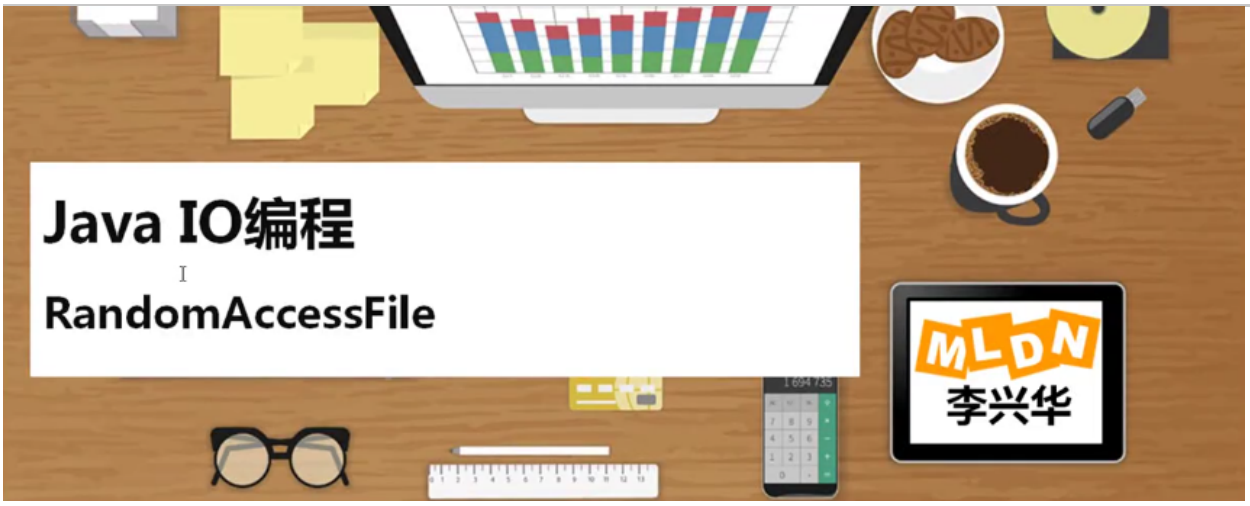
```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		SendThread send = new SendThread();
		ReceiveThread receive = new ReceiveThread();
		send.getOutput().connect(receive.getInput()); // 进行管道连接
		new Thread(send, "消息发送线程").start();
		new Thread(receive, "消息接收线程").start();
	}
}
class SendThread implements Runnable {
	private PipedOutputStream output; // 管道的输出流

	public SendThread() {
		this.output = new PipedOutputStream(); // 实例化管道输出流
	}
	@Override
	public void run() {
		for (int x = 0; x < 10; x++) {
			try { // 利用管道实现数据的发送处理
				this.output.write(
						("【第" + (x + 1) + "次信息发送 - " +
Thread.currentThread().getName() + "】www.mldn.cn\n").getBytes());
			} catch (IOException e) {
				e.printStackTrace();
			}
		}
		try {
			this.output.close();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
	public PipedOutputStream getOutput() {
		return output;
	}
}
class ReceiveThread implements Runnable {
	private PipedInputStream input;

	public ReceiveThread() {
		this.input = new PipedInputStream();
	}
	@Override
	public void run() {
		byte data[] = new byte[1024];
		int len = 0 ;
		ByteArrayOutputStream bos = new ByteArrayOutputStream() ;// 所有的数据保存到
内存输出流
```

```
            try {
                while((len = this.input.read(data)) != -1) {
                    bos.write(data,0,len); // 所有的数据保存到内存流
                }
                System.out.println("  {" + Thread.currentThread().getName() + "接收消息}  \n"
+ new String(bos.toByteArray()));
                bos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                this.input.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        public PipedInputStream getInput() {
            return input;
        }
}
```

管道就类似于医院打点滴效果，一个只是负责发送，一个负责接收，中间靠一个管道连接。

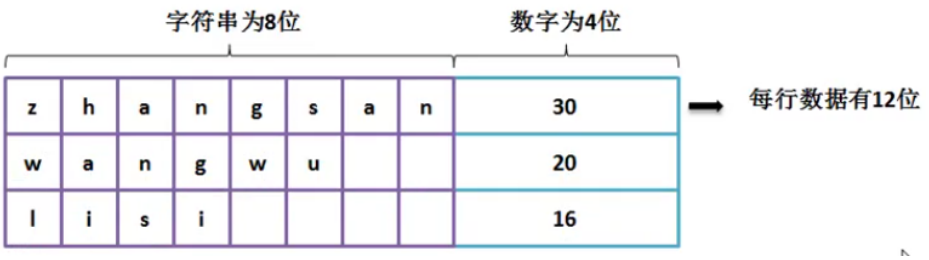


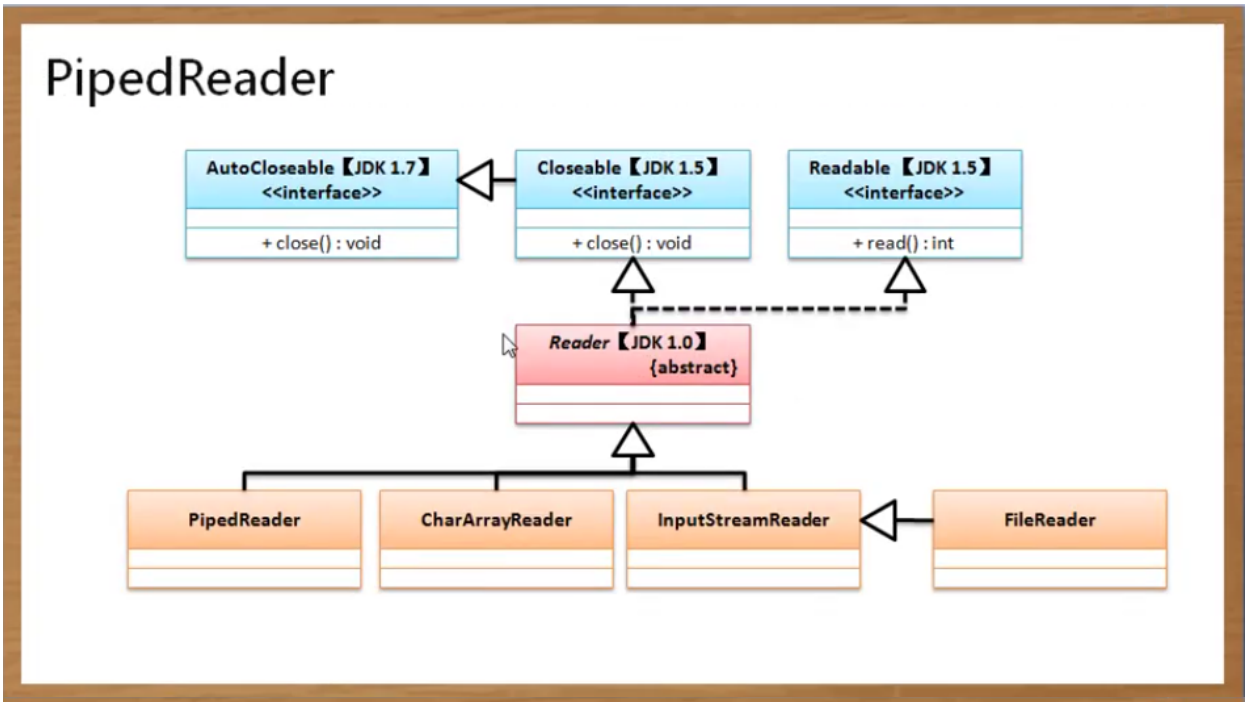
2、具体内容

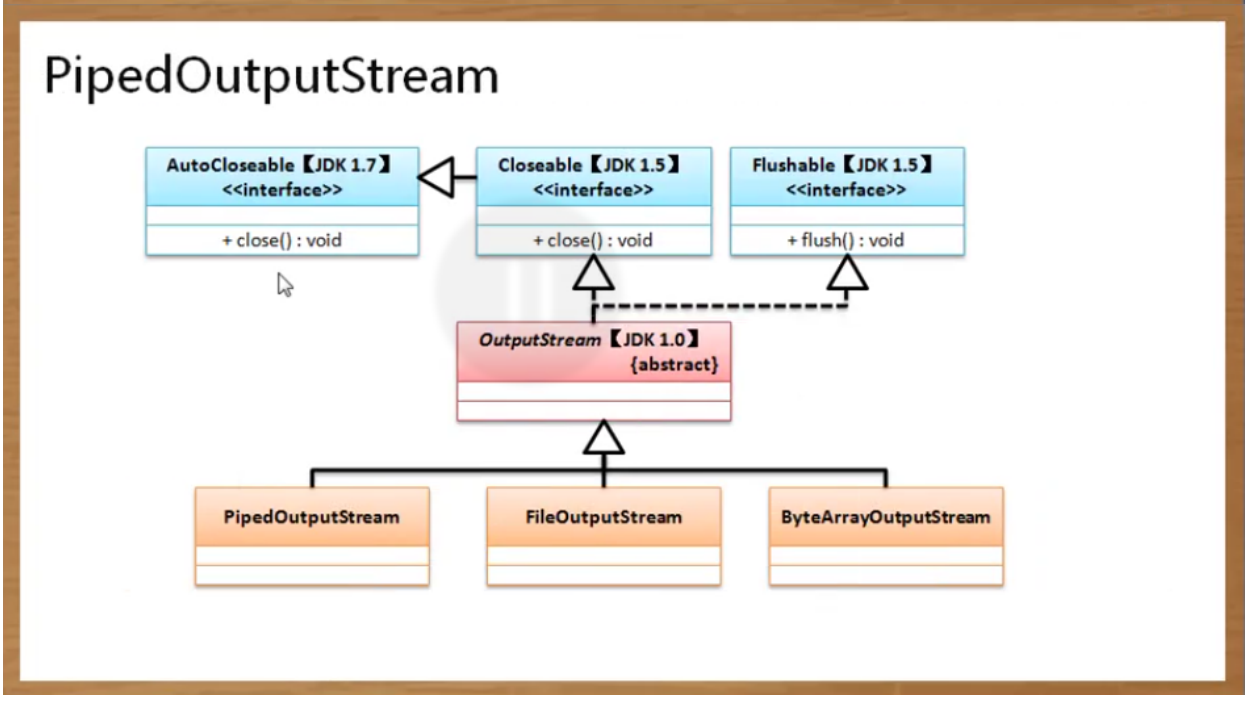
对于文件内容的处理操作主要是通过InputStream（Reader）、OutputStream（Writer）来实现，但是利用这些类实现的内容读取只能够将数据部分部分读取进来，如果说现在有这样一种要求。

现在给了一个非常庞大的文件，这个文件的大小有20G，如果此时按照传统的IO操作进行读取和分析根本就不可能完成，所以这种情况下载java.io包里面就有一个RandomAccessFile类，这个类可以实现文件的跳跃式的读取，可以只读取中间的部分内容（前提：需要有一个完善的保存形式），数据的保存位数要都确定好。

RandomAccessFile







RandomAccessFile类里面定义有如下的操作方法：

·构造方法：public RandomAccessFile(File file,String mode) throws FileNotFoundException；

|- 文件处理方式：r、rw；

范例：实现文件的保存

```
import java.io.File;
import java.io.RandomAccessFile;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		File file = new File("d:" + File.separator + "mldn.txt"); // 定义操作文件
		RandomAccessFile raf = new RandomAccessFile(file, "rw"); // 读写模式
		String names[] = new String[] { "zhangsan", "wangwu  ", "lisi    " };
		int ages[] = new int[] { 30, 20, 16 };
		for (int x = 0; x < names.length; x++) {
			raf.write(names[x].getBytes()); // 写入字符串
			raf.writeInt(ages[x]);
		}
		raf.close();
	}
}
```

RandomAccessFile最大的特点是在于数据的读取处理上，因为所有的数据是按照固定的长度进行的保存，所以读取的时候就可以进行跳字节读取：

·向下跳：public int skipBytes(int n) throws IOException；

·向回跳：public void seek(long pos) throws IOException；

范例：读取数据

```
import java.io.File;
import java.io.RandomAccessFile;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		File file = new File("d:" + File.separator + "mldn.txt"); // 定义操作文件
		RandomAccessFile raf = new RandomAccessFile(file, "rw"); // 读写模式
		{	// 读取“李四”的数据，跳过24位
			raf.skipBytes(24) ;
			byte[] data = new byte[8];
			int len = raf.read(data) ;
			System.out.println("姓名：" + new String(data,0,len).trim() + "、年龄：" +
raf.readInt());
		}
		{	// 读取“王五”的数据，回跳12位
			raf.seek(12);
			byte[] data = new byte[8];
			int len = raf.read(data) ;
			System.out.println("姓名：" + new String(data,0,len).trim() + "、年龄：" +
raf.readInt());
		}
		{	// 读取“张三”的数据，回跳头
			raf.seek(0);	// 回到顶点
			byte[] data = new byte[8];
```

```
                int len = raf.read(data) ;
                System.out.println("姓名：" + new String(data,0,len).trim() + "、年龄：" +
raf.readInt());
            }
            raf.close();
        }
}
```

整体的使用之中由用户自行定义要读取的位置，而后按照指定的结构进行数据的读取。