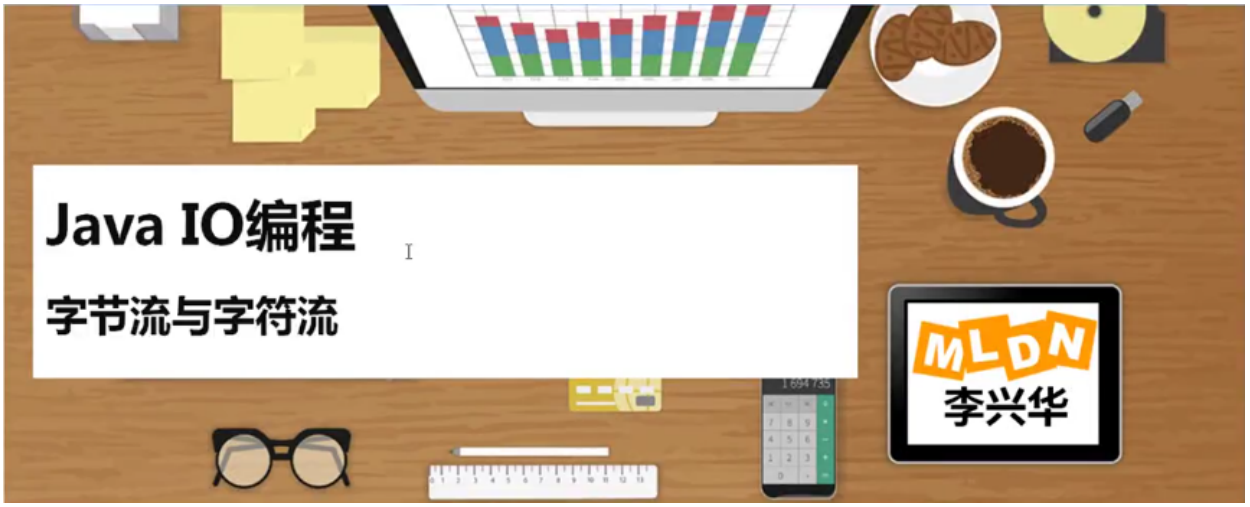
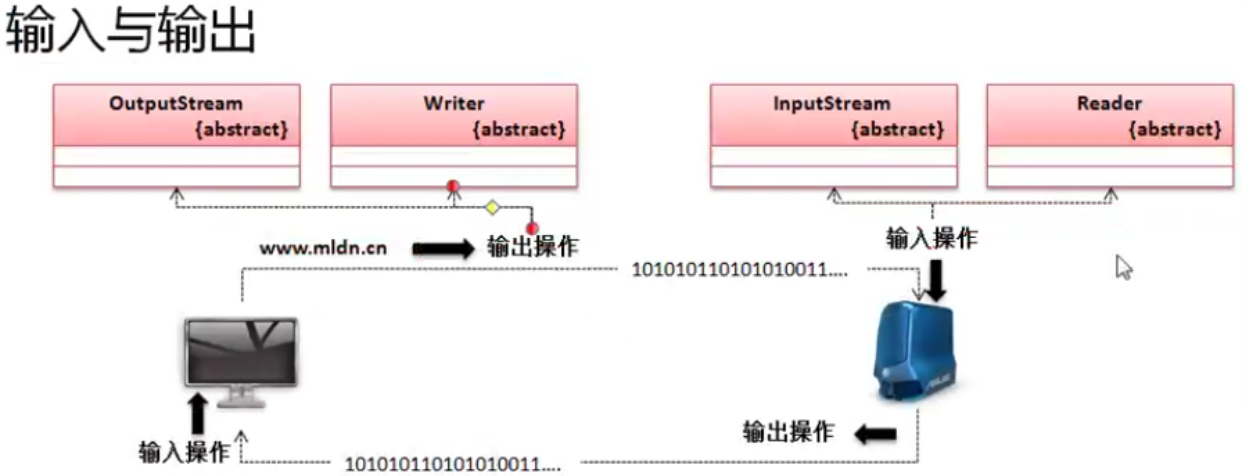


博客： https://www.cnblogs.com/HOsystem/p/14116443.html

2、具体内容

在java.io包里面File类是唯一 一个与文件本身有关的程序处理类，但是File只能够操作文件本身而不能够操作文件的内容，或者说在实际的开发之中IO操作的核心意义在于：输入与输出操作。而对于程序而言，输入与输出可能来自于不同的环境，例如：通过电脑连接服务器上进行浏览的时候，实际上此时客户端发出了一个信息，而后服务器接收到此信息之后进行回应处理。



对于服务器或者是客户端而言实际上传递的就是一种数据流的处理形式，而所谓的数据流指的就是字节数据。而对于这种流的处理形式在java.io包里面提供有两类支持：

·字节处理流：OutputStream（输出字节流）、InputStream（输入字节流）；

·字符处理流：Writer（输出字符流）、Reader（输入字符流）；

所有的流操作都应该采用如下统一的步骤进行，下面以文件处理的流程为例：

·如果现在要进行的是文件的读写操作，则一定要通过File类找到一个文件路径；

·通过字节流或字符流的子类为父类对象实例化；

·利用字节流或字符流中的方法实现数据的输入与输出操作；

·流的操作属于资源操作，资源操作必须进行关闭处理；

■字节输出流：OutputStream

字节的数据是以byte类型为主实现的操作，在进行字节内容输出的时候可以使用OutputStream类完成，这个类的基本定义如下：

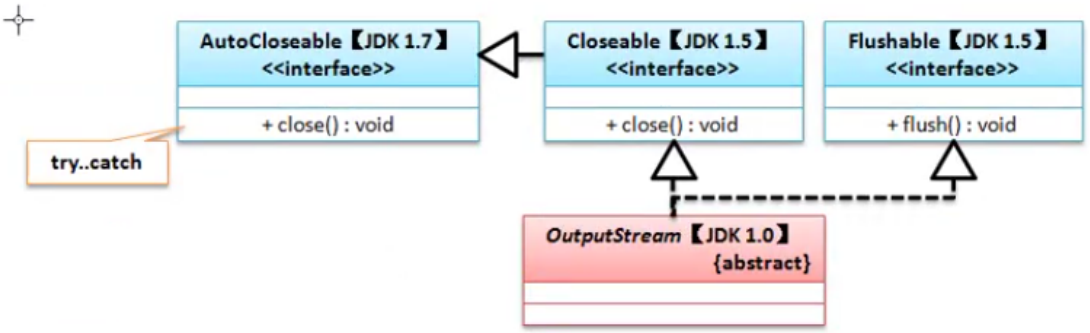
```
public abstract class OutputStream extends Object implements Closeable, Flushable
```

首先可以发现这个类实现了两个接口，于是基本的对应关系如下：

Closeable：	**Flushable：**

public interface Closeable extends AutoCloseable {
public void close() throws Exception;
}
	public interface Flushable{
	public void flush() throws IOException;
	}

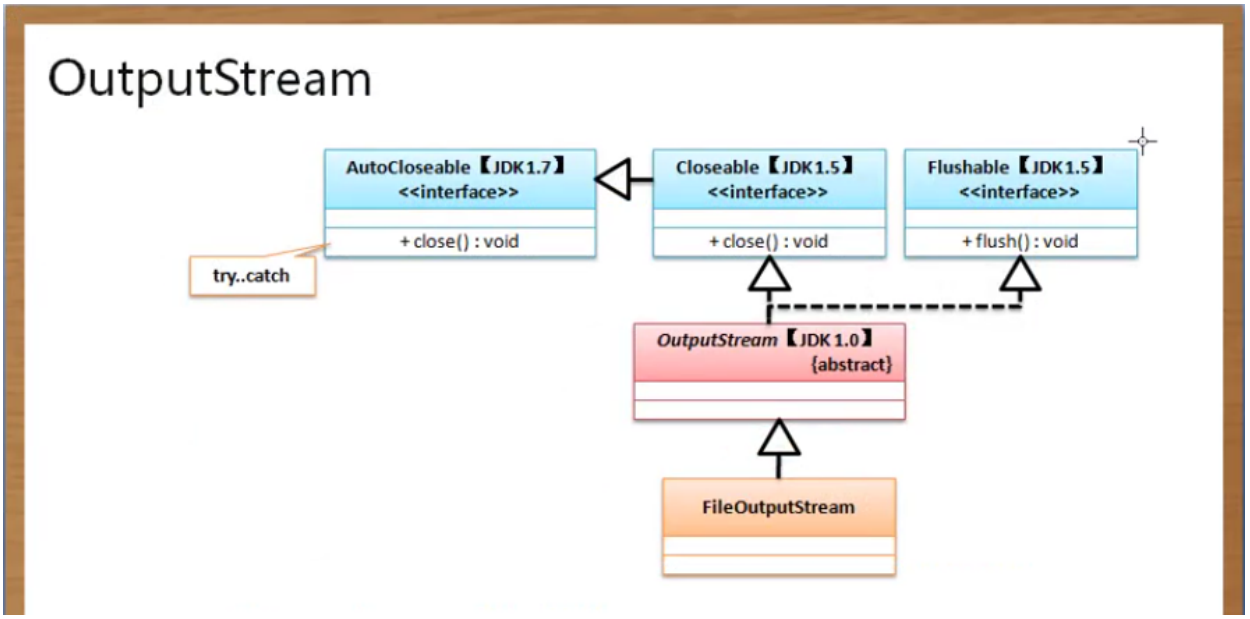
OutputStream



OutputStream类定义的是一个公共的输出操作标准，而在这个操作标准里面一共定义有三个内容输出的方法。

No	**方法名称**	**类型**	**描述**
01	public abstract void write(int b) throws IOException	普通	输出单个字节数据
02	public void write(byte[] b) throws IOException	普通	输出一组字节数据
03	**public void write(byte[] b,int off,int len) throws IOException**	普通	输出部分字节数据

但是急需要注意的一个核心问题在于：OutputStream类毕竟是一个抽象类，而这个抽象类如果要想获得实例化对象，按照传统的认识应该通过子类实例的向上转完成，如果说现在要进行的是文件处理操作，则可以使用FileOutputStream子类：



因为最终都需要发生向上转型的处理关系，所以对于此时的FileOutputStream子类核心的关注点就可以放在构造方法：

·【覆盖】构造方法：public FileOutputStream(File file) throws FileNotFoundException；

·【追加】构造方法：public FileOutputStream(File file,boolean append) throws FileNotFoundException；

范例：使用OutputStream类实现内容的输出

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		File file = new File("D:" + File.separator + "hello" +
			File.separator + "mldn.txt");		// 1、指定要操作的文件的路径
		if (!file.getParentFile().exists()) {		// 文件不存在
			file.getParentFile().mkdirs() ;		// 创建父目录
		}
		OutputStream output = new FileOutputStream(file) ; // 2、通过子类实例化
		String str = "www.mldn.cn" ; // 要输出的文件内容
		output.write(str.getBytes()); // 3、将字符串变为字节数组并输出
		output.close(); // 4、关闭资源
	}
}
```

本程序是采用了最为标准的形式实现了输出的操作处理，并且在整体的处理之中，只是创建了文件的父目录，但是并没有创建文件，而在执行后会发现文件可以自动帮助用户创建。另外需要提醒的是，由于OutputStream子类也属于AutoCloseable接口子类，所以对于close()方法也可以简化使用。

范例：自动关闭处理

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

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		File file = new File("D:" + File.separator + "hello" + File.separator + "mldn.txt"); //
1、指定要操作的文件的路径
		if (!file.getParentFile().exists()) { // 文件不存在
			file.getParentFile().mkdirs(); // 创建父目录
		}
		try (OutputStream output = new FileOutputStream(file, true)) {//true表示追加数据
			String str = "www.mldn.cn\r\n"; // 要输出的文件内容
			output.write(str.getBytes()); // 3、将字符串变为字节数组并输出
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

是否使用自动的关闭取决于项目的整体结构，另外还需要提醒的是，整个的程序里面最终是输出了一组的字节数据，但是千万不要忘记了，OutputStream类之中定义的输出方法一共有三个。

■字节输入流：InputStream

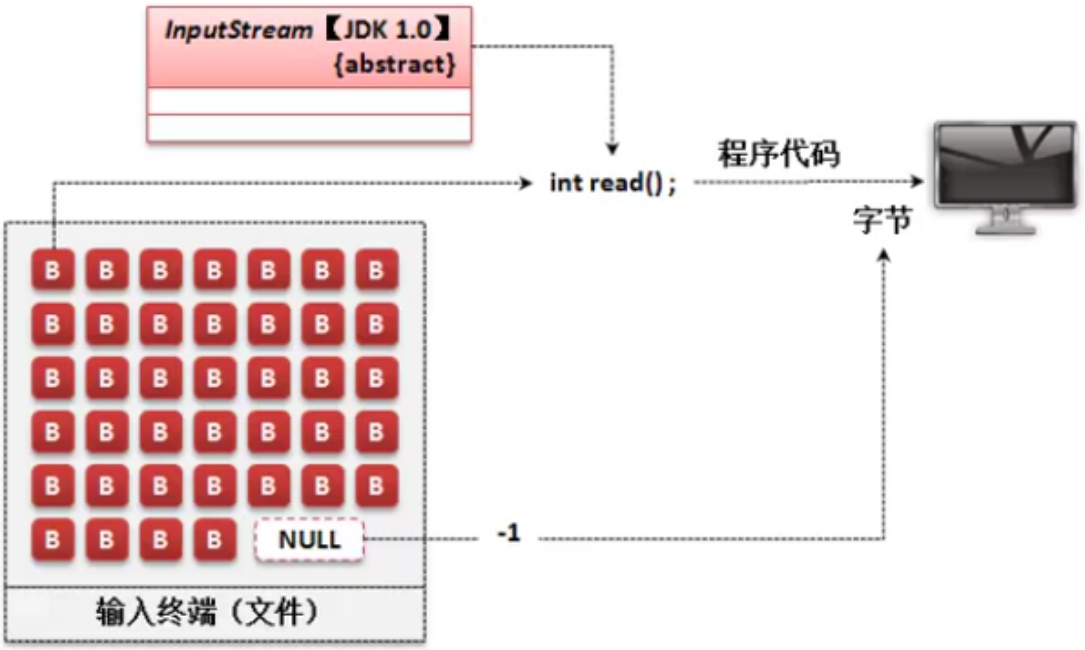
与OutputStream类对应的一个流就是字节输入流，InputStream类主要实现的就是字节数据读取，该类定义如下：

```
public abstract class InputStream extends Object implements Closeable
```

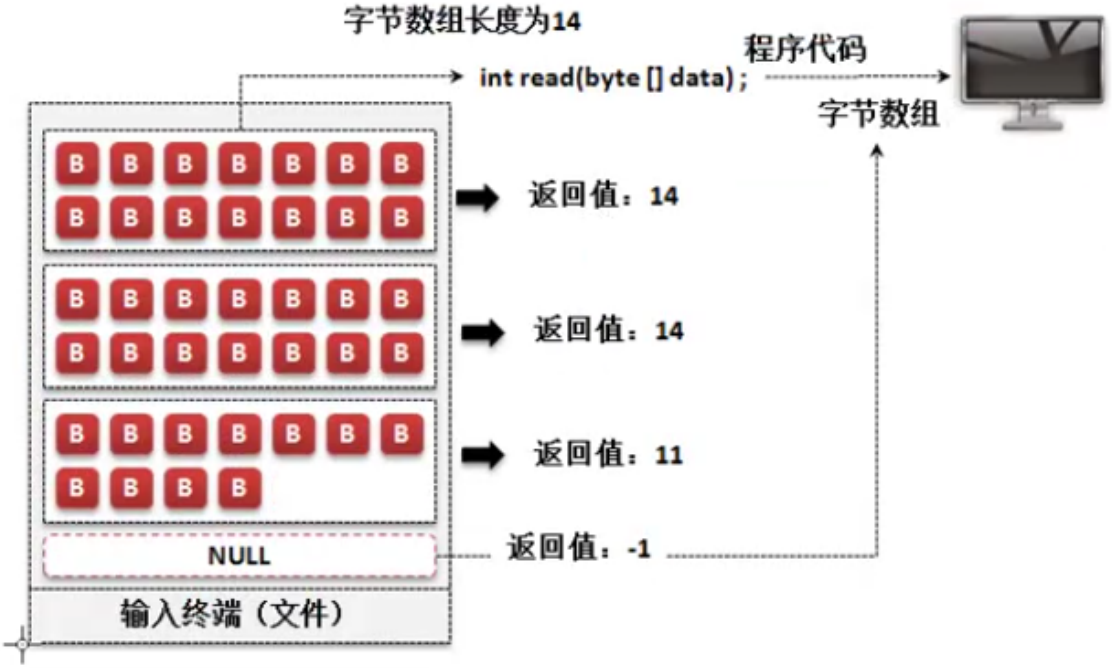
在InputStream类里面定义有如下的几个核心方法：

No	方法名称	类型	描述
01	public abstract int read() throws IOException	普通	读取单个字节数据，如果现在已经读取到底，则返回-1
02	**public int read(byte[] b) throws IOException**	普通	读取一组字节数据，返回的是读取的个数，如果没有数据已经读取到底则返回-1
03	public int read(byte[] b,int off,int len) throws IOException	普通	读取一组字节数据（只占数组的部分）

输入（int read()）

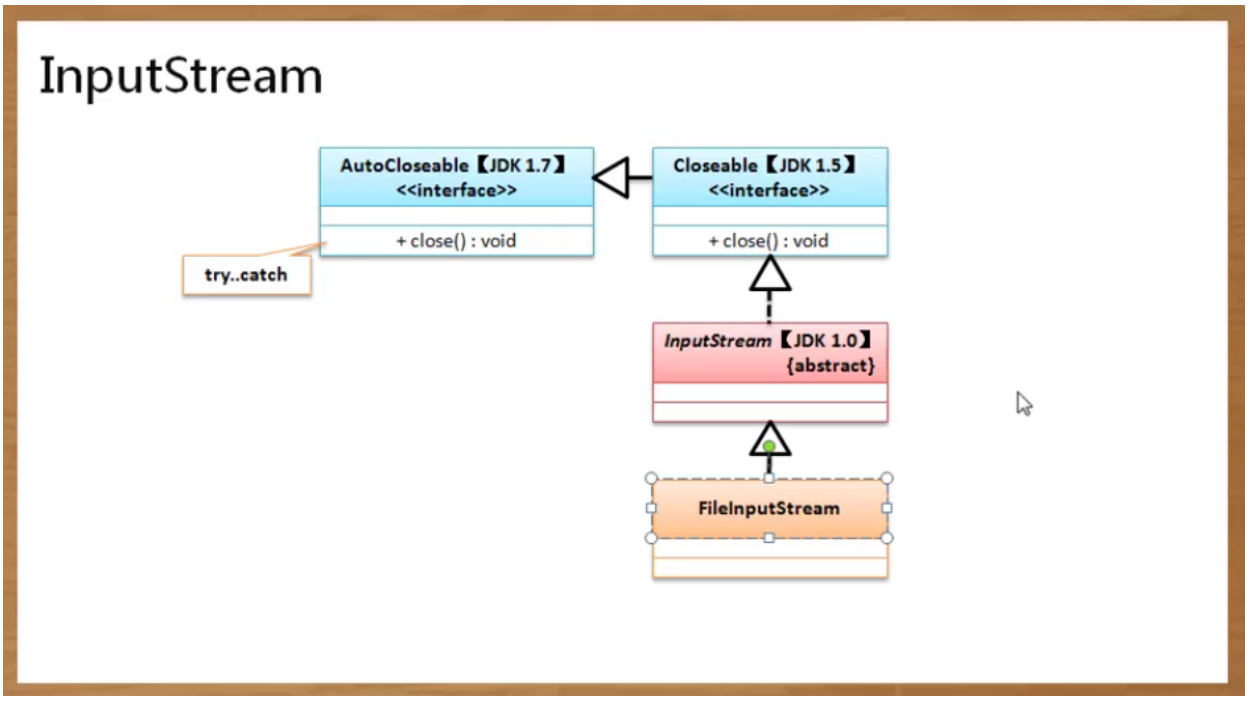


输入（int read(byte data[])）



InputStream类属于一个抽象类，这时应该依靠它的子类来实例化对象，如果要从文件读取一定使用FileInputStream子类，对于子类而言只关心父类对象实例化。

构造方法：public FileInputStream(File file) throws FileNotFoundException;



范例：读取数据

```
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		File file = new File("D:" + File.separator + "hello" + File.separator + "mldn.txt");
		InputStream input = new FileInputStream(file) ;
		byte data [] = new byte [1024] ; // 开辟一个缓冲区读取数据
		int len = input.read(data) ; // 读取数据，数据全部保存在字节数组之中，返回读取个数
		System.out.println("【" + new String(data, 0, len) + "】");
		input.close();
	}
}
```

对于字节输入流里面最为麻烦的问题就在于：使用read()方法读取的时候只能够以字节数组为主进行接收。

特别需要注意的是从JDK1.9开始在InputStream类里面增加了一个新的方法：public byte[] readAllBytes() throws IOException；

范例：新方法

```
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		File file = new File("D:" + File.separator + "hello" + File.separator + "mldn.txt");
		InputStream input = new FileInputStream(file) ;
		byte data [] = input.readAllBytes() ; // 读取全部数据
		System.out.println("【" + new String(data) + "】");
		input.close();
	}
```

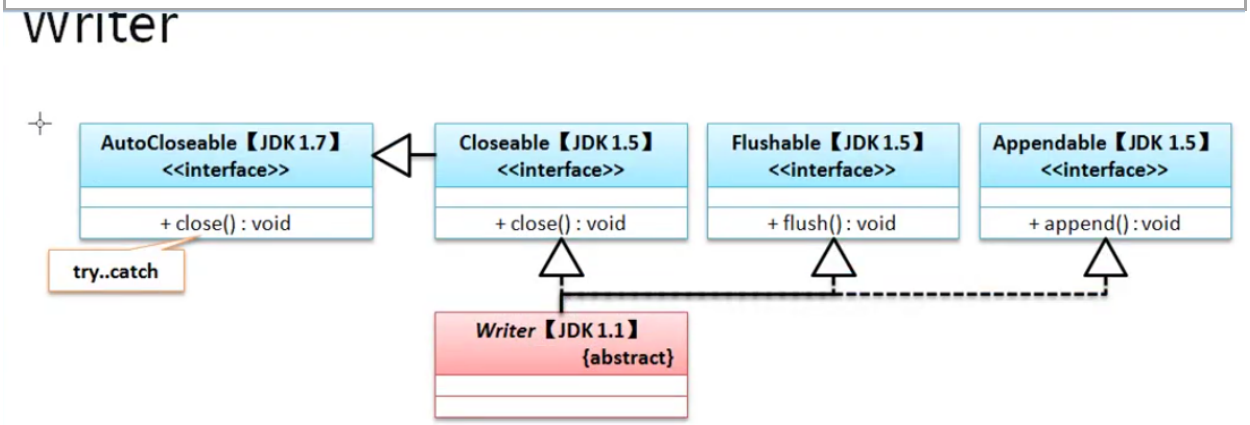
```
}
```

如果现在要读取的内容很大很大的时候，那么这种读取直接搞死你的程序。如果要使用尽量不要超过10KB。

■字符输出流：Writer

使用OutputStream字节输出流进行数据输出的时候使用的都是字节类型的数据，而很多的情况下字符串的输出是比较方便的，所以对于java.io包而言，在JDK1.1的时候又推出了字符输出流：Writer，这个类的定义如下：

```
public abstract class Writer  extends Object implements Appendable, Closeable, Flushable
```



在Writer类里面提供有许多的输出操作方法，重点来看两个：

·输出字符数组：public void write(char[] cbuf) throws IOException；

·输出字符串：public void write(String str) throws IOException；

范例：使用Writer输出

```
import java.io.File;
import java.io.FileWriter;
import java.io.Writer;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		File file = new File("D:" + File.separator + "hello" + File.separator + "mldn.txt");
		if (!file.getParentFile().exists()) {
			file.getParentFile().mkdirs(); // 父目录必须存在
		}
		Writer out = new FileWriter(file) ;
		String str = "www.mldn.cn" ;
		out.write(str);
		out.close();
	}
}
```

```
import java.io.File;
import java.io.FileWriter;
import java.io.Writer;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
```

```
        File file = new File("D:" + File.separator + "hello" + File.separator + "mldn.txt");
        if (!file.getParentFile().exists()) {
            file.getParentFile().mkdirs(); // 父目录必须存在
        }
        Writer out = new FileWriter(file) ;
        //Writer out = new FileWriter(file,true) ; //追加需要使用true
        String str = "www.mldn.cn" ;
        out.write(str);
        out.append("hello");       //追加输出内容
        out.close();
    }
}
```

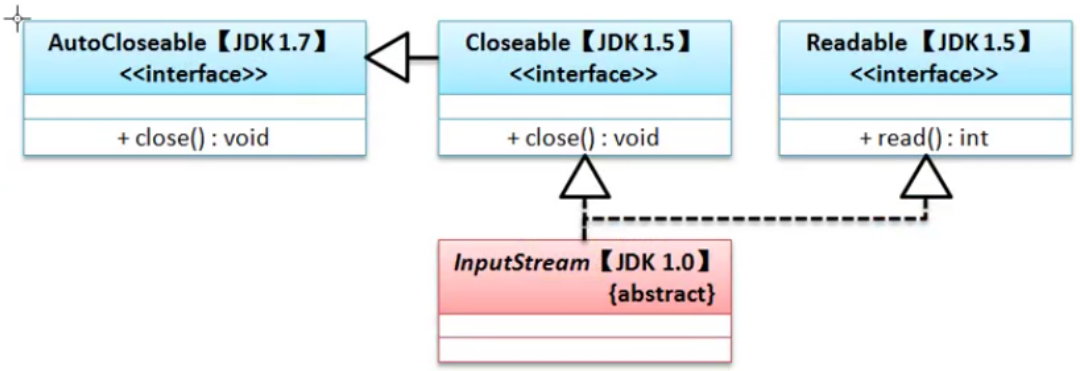
使用Writer输出的最大优势在于可以直接利用字符串完成。Writer是字符流，字符处理的优势在于中文数据上。

■字符输入流：Reader

Reader是实现字符输入流的一种类型，其本身属于一个抽象类，这个类的定义如下：

```
public abstract class Reader extends Object implements Readable, Closeable
```

Reader



Reader类里面并没有像Writer类一样提供有整个字符串的输入处理操作，只能够利用字符数据来实现接收：

·接收数据：public int read(char[] cbuf) throws IOException；

范例：实现数据读取

```
import java.io.File;
import java.io.FileReader;
import java.io.Reader;
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
        File file = new File("D:" + File.separator + "hello" + File.separator + "mldn.txt");
        if (file.exists()) {       // 文件存在则进行读取
            Reader in = new FileReader(file) ;
            char data[] = new char[1024];
            int len = in.read(data) ;
```

```
            System.out.println("读取内容：" + new String(data,0,len));
            in.close();
        }
    }
}
```

字符流读取的时候只能够按照数组的形式来实现处理操作。

■字节流与字符流的区别

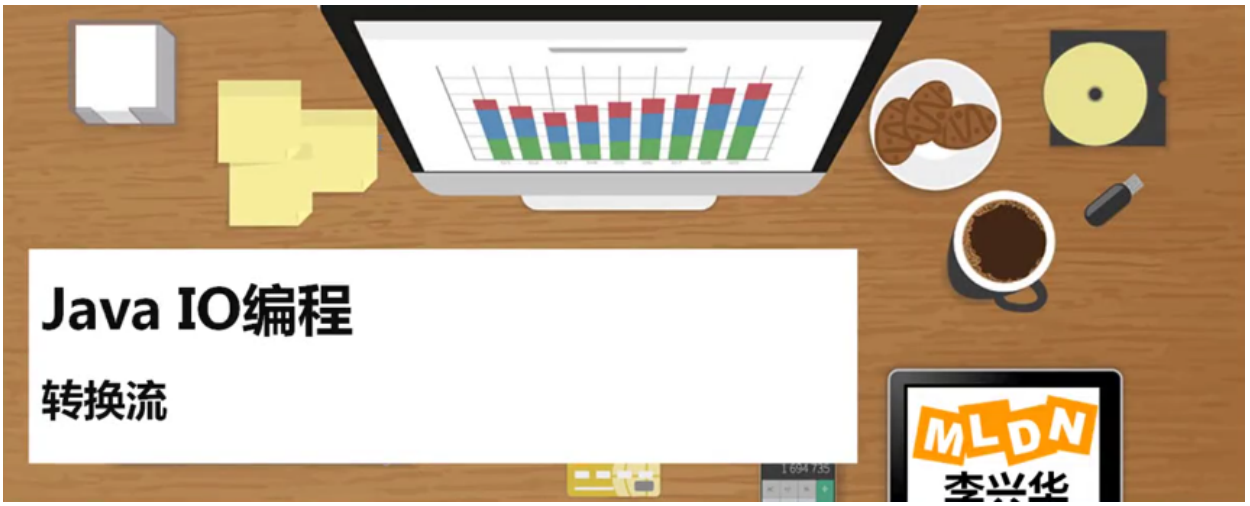
现在通过一系列的分析已经可以清楚字节流与字符流的基本操作了，但是对于这两类流依然是存在有区别的，重点来分析一下输出的处理操作。在使用OutputStream和Writer输出的最后发现都使用了close()方法进行了关闭处理。

在使用OutputStream类输出的时候如果没有使用close()方法关闭输出流发现内容依然可以实现正常的输出，但是如果在使用Writer的时候没有使用close()方法关闭输出流，那么这个时候内容将无法进行输出，因为Writer使用到了缓冲区，当使用了close()方法的时候实际上会出现有强制刷新缓冲区的情况，所以这个时候会将内容进行输出，如果没有关闭，那么将无法进行输出操作，所以此时如果在不关闭的情况下要想将全部的内容输出可以使用flush()方法强制清空。

范例：使用Writer并强制性清空

```
import java.io.File;
import java.io.FileWriter;
import java.io.Writer;
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
        File file = new File("D:" + File.separator + "hello" + File.separator + "mldn.txt");
        if (!file.getParentFile().exists()) {
            file.getParentFile().mkdirs(); // 父目录必须存在
        }
        Writer out = new FileWriter(file) ;
        String str = "www.mldn.cn" ;
        out.write(str);
        out.flush(); // 强制性刷新
    }
}
```

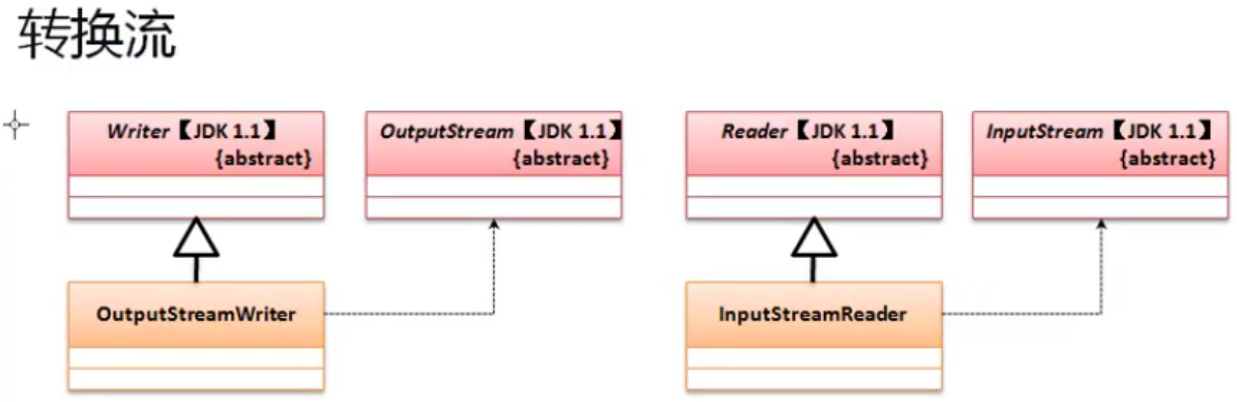
字节流在进行处理的时候并不会使用到缓冲区，而字符流会使用到缓冲区。另外使用缓冲区的字符流更加适合于进行中文数据的处理，所以在日后的程序开发之中，如果要涉及到包含有中文信息的输出一般都会使用字符流处理，但是从另外一方面来讲，字节流和字符流的基本处理形式是相似的，由于IO很多情况下都是进行数据的传输使用（二进制）所以本次的讲解将以字节流为主。



2、具体内容

所谓的转换流指的是可以实现字节流与字符流操作的功能转换，例如：进行输出的时候OutputStream需要将内容变为字节数组后才可以进行输出，而Writer可以直接输出字符串，这一点是方便的，所以很多人就认为需要提供有一种转换的机制，来实现不同流类型的转化操作，为此在java.io包里面提供有两个类：InputStreamReader、OutputStreamWriter。

类	InputStreamReader:	OutputStreamWriter:
定义	public class OutputStreamWriter extends Writer	public class InputStreamReader extends Reader
构造方法	public OutputStreamWriter(OutputStream out)	public InputStreamReader (InputStream in)

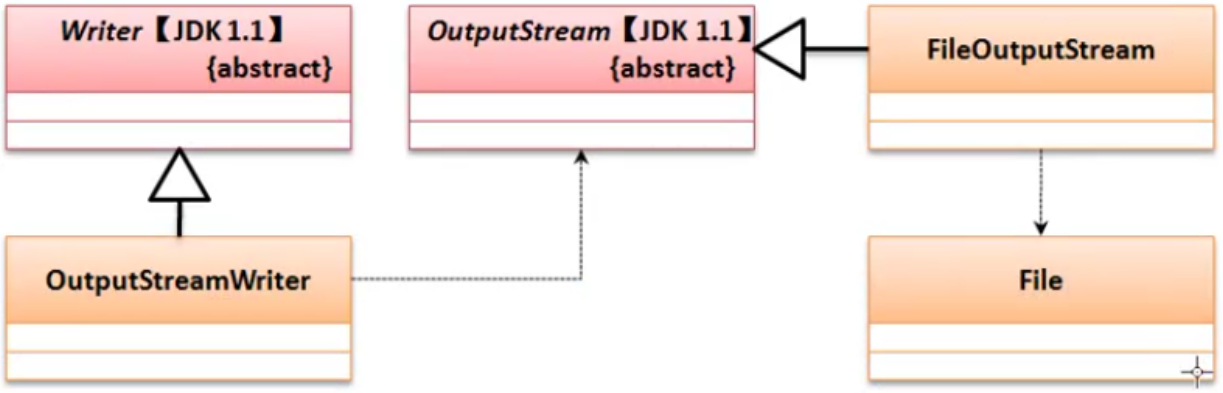


通过类的继承结构与构造方法可以发现，所谓的转换处理就是将接收到的字节流对象通过向上转型变为字符流对象。

范例：观察转换

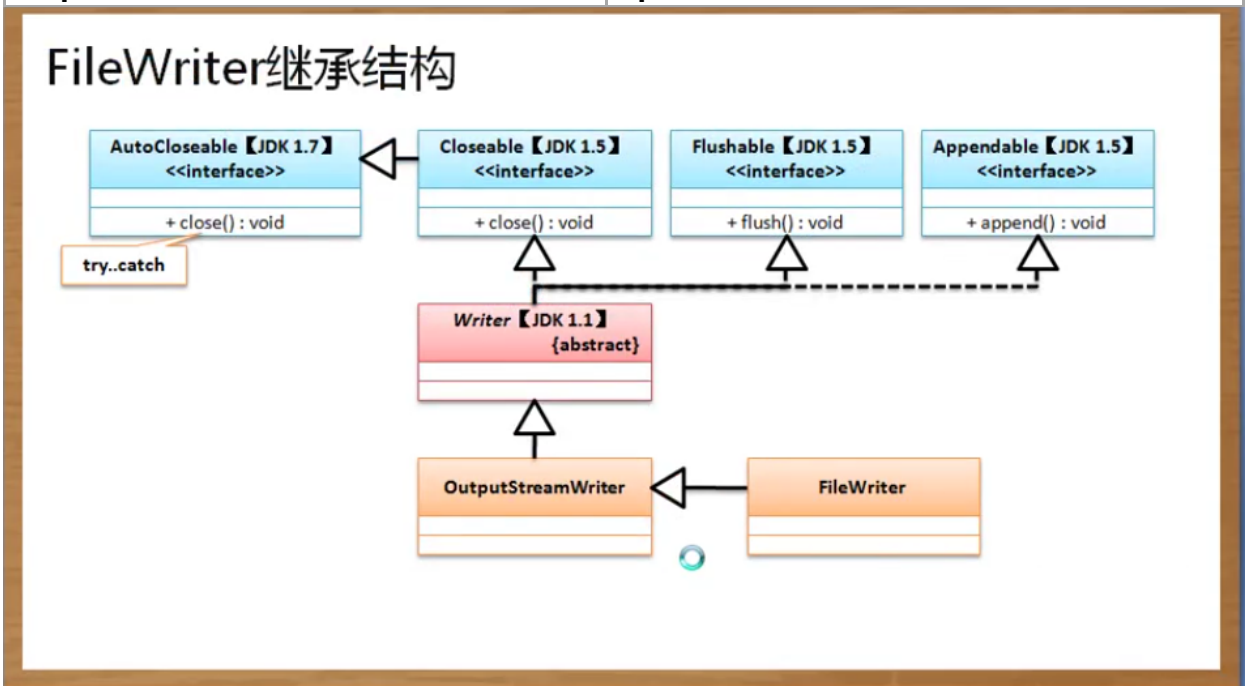
```
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
```

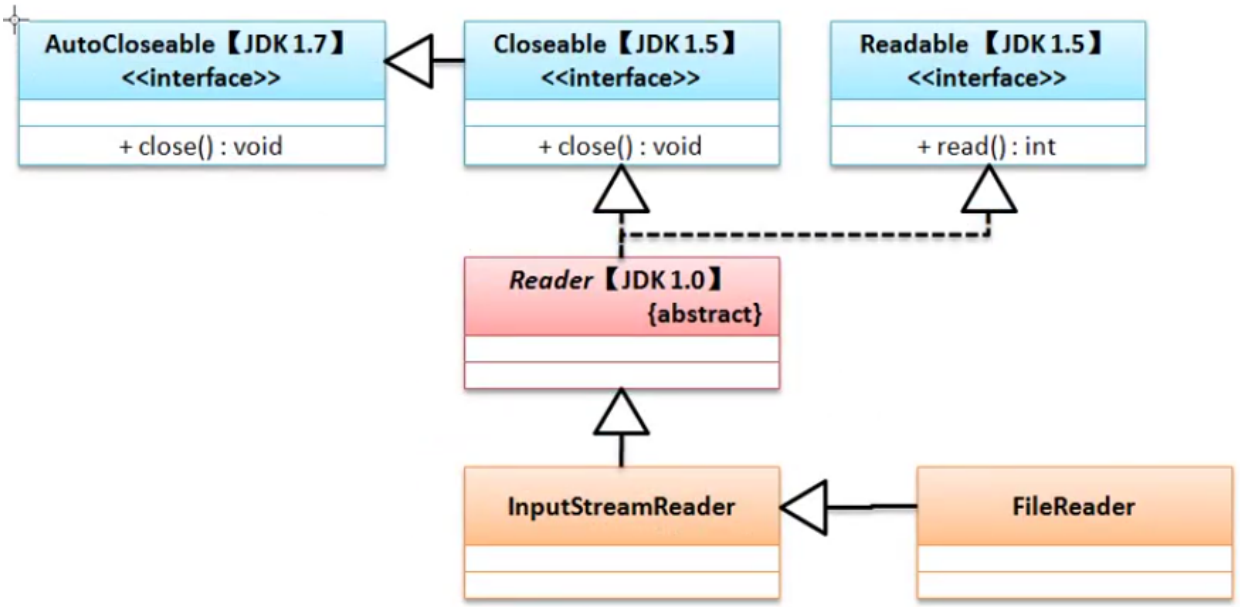
```
        File file = new File("D:" + File.separator + "hello" + File.separator + "mldn.txt");
        if (!file.getParentFile().exists()) {
            file.getParentFile().mkdirs(); // 父目录必须存在
        }
        OutputStream output = new FileOutputStream(file) ;
        Writer out = new OutputStreamWriter(output) ; // 字节流变为字符流
        out.write("www.mldn.cn"); // 直接输出字符串，字符流适合于处理中文
        out.close();
    }
}
```



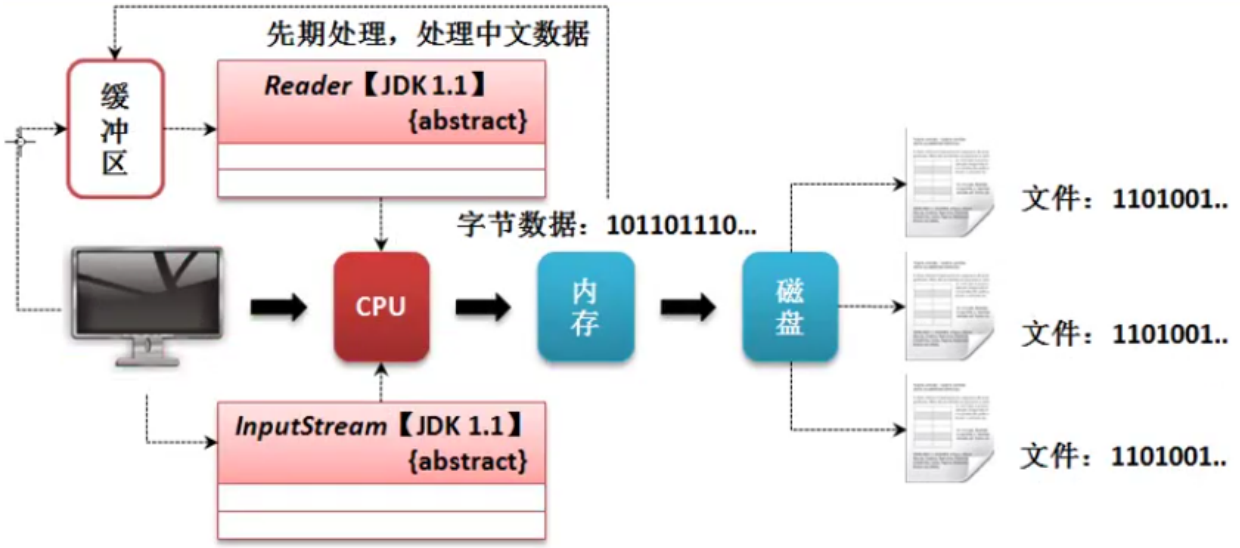
讲解转换流的主要目的基本上不是为了让开发者去记住它，而是知道有这样一种功能，但同时更多的是需要进行结构的分析处理。通过之前的字节流与字符流的一系列的分析之后你会发现OutputStream类有FileOutputStream直接子类、InputStream类有FileInputStream直接子类，但是来观察一下FileWriter、FileReader类的继承关系。

FileWriter	**FileReader**
public class FileWriter extends **OutputStreamWriter**	public class FileReader extends **InputStreamReader**





实际上所谓的缓存都是指的是程序中间的一道处理缓冲区。



2、具体内容

在操作系统里面有一个copy命令，这个命令的主要功能是可以实现文件的拷贝处理，现在要求模拟这个命令，通过初始化参数输入拷贝的源文件路径与拷贝的目标路径实现文件的拷贝处理。

需求分析：

·需要实现文件的拷贝操作，那么这种拷贝就有可能拷贝各种类型的文件，所以肯定使用字节流；

·在进行拷贝的时候有可能需要考虑到大文件的拷贝问题；

实现方案：

·**方案一**：使用InputStream将全部要拷贝的内容直接读取到程序里面，而后一次性的输出到目标文件；

|- 如果现在拷贝的文件很大，基本上程序就死了；

·**方案二**：采用部分拷贝，读取一部分输出一部分数据，如果现在要采用第二种做法，核心的操作方法：

|- InputStream：public int read(byte[] b) throws IOException；

|- OutputStream：public void write(byte[] b,int off, int len) throws IOException；

范例：实现文件拷贝处理

```
//需要在编译的时候输入文件的名称要包含后缀名，两个文件名
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
class FileUtil {  // 定义一个文件操作的工具类
	private File srcFile ; // 源文件路径
	private File desFile ; // 目标文件路径
	public FileUtil(String src,String des) {
		this(new File(src),new File(des)) ;
	}
	public FileUtil(File srcFile,File desFile) {
		this.srcFile = srcFile ;
		this.desFile = desFile ;
	}
	public boolean copy() throws Exception {	// 文件拷贝处理
		if (!this.srcFile.exists()) {  // 源文件必须存在!
			System.out.println("拷贝的源文件不存在！");
			return false ; // 拷贝失败
		}
		if (!this.desFile.getParentFile().exists()) {
			this.desFile.getParentFile().mkdirs() ; // 创建父目录
		}
		byte data [] = new byte[1024] ; // 开辟一个拷贝的缓冲区
```

```
            InputStream input = null ;
            OutputStream output = null ;
            try {
                  input = new FileInputStream(this.srcFile) ;
                  output = new FileOutputStream(this.desFile) ;
                  int len = 0 ;
                  do {
                        len = input.read(data) ; // 拷贝的内容都在data数组
                        if (len != -1) {
                              output.write(data, 0, len);
                        }
                  } while (len != -1) ;
                  return true ;
            } catch (Exception e) {
                  throw e ;
            } finally {
                  if (input != null) {
                        input.close();
                  }
                  if (output != null) {
                        output.close() ;
                  }
            }
      }
}
public class JavaAPIDemo {
      public static void main(String[] args) throws Exception {
            if (args.length != 2) {       // 程序执行出错
                  System.out.println("命令执行错误，执行结构：java JavaAPIDemo 拷贝源文件路
径 拷贝目标文件路径");
                  System.exit(1);
            }
            long start = System.currentTimeMillis() ;
            FileUtil fu = new FileUtil(args[0],args[1]) ;
            System.out.println(fu.copy() ? "文件拷贝成功！" : "文件拷贝失败！");
            long end = System.currentTimeMillis() ;
            System.out.println("拷贝完成的时间：" + (end - start));
      }
}
```

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
class FileUtil {  // 定义一个文件操作的工具类
      private File srcFile ; // 源文件路径
      private File desFile ; // 目标文件路径
      public FileUtil(String src,String des) {
            this(new File(src),new File(des)) ;
      }
      public FileUtil(File srcFile,File desFile) {
```

```
            this.srcFile = srcFile ;
            this.desFile = desFile ;
        }
        public boolean copy() throws Exception {      // 文件拷贝处理
            if (!this.srcFile.exists()) {   // 源文件必须存在!
                System.out.println("拷贝的源文件不存在！");
                return false ; // 拷贝失败
            }
            if (!this.desFile.getParentFile().exists()) {
                this.desFile.getParentFile().mkdirs() ; // 创建父目录
            }
            byte data [] = new byte[1024] ; // 开辟一个拷贝的缓冲区
            InputStream input = null ;
            OutputStream output = null ;
            try {
                input = new FileInputStream(this.srcFile) ;
                output = new FileOutputStream(this.desFile) ;
                int len = 0 ;
                // 1、读取数据到数组之中，随后返回读取的个数、len = input.read(data
                // 2、判断个数是否是-1，如果不是则进行写入、(len = input.read(data)) != -1
                while ((len = input.read(data)) != -1) {
                    output.write(data, 0, len);
                }
                return true ;
            } catch (Exception e) {
                throw e ;
            } finally {
                if (input != null) {
                    input.close();
                }
                if (output != null) {
                    output.close() ;
                }
            }
        }
}
public class JavaAPIDemo {
        public static void main(String[] args) throws Exception {
            if (args.length != 2) {       // 程序执行出错
                System.out.println("命令执行错误，执行结构：java JavaAPIDemo 拷贝源文件路
径 拷贝目标文件路径");
                System.exit(1);
            }
            long start = System.currentTimeMillis() ;
            FileUtil fu = new FileUtil(args[0],args[1]) ;
            System.out.println(fu.copy() ? "文件拷贝成功！" : "文件拷贝失败！");
            long end = System.currentTimeMillis() ;
            System.out.println("拷贝完成的时间：" + (end - start));
        }
}
```

但是需要注意的是，以上的做法是属于文件拷贝的最原始的实现，而从JDK1.9开始InputStream和Reader类中都追加有数据转存的处理操作方法：

·**InputStream**：public long transferTo(OutputStream out) throws IOException；

·**Reader**：public long transferTo(Writer out) throws IOException；

范例：使用转存的方式处理

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
class FileUtil {  // 定义一个文件操作的工具类
	private File srcFile ; // 源文件路径
	private File desFile ; // 目标文件路径
	public FileUtil(String src,String des) {
		this(new File(src),new File(des)) ;
	}
	public FileUtil(File srcFile,File desFile) {
		this.srcFile = srcFile ;
		this.desFile = desFile ;
	}
	public boolean copy() throws Exception {	// 文件拷贝处理
		if (!this.srcFile.exists()) {  // 源文件必须存在!
			System.out.println("拷贝的源文件不存在！");
			return false ; // 拷贝失败
		}
		if (!this.desFile.getParentFile().exists()) {
			this.desFile.getParentFile().mkdirs() ; // 创建父目录
		}
		InputStream input = null ;
		OutputStream output = null ;
		try {
			input = new FileInputStream(this.srcFile) ;
			output = new FileOutputStream(this.desFile) ;
			input.transferTo(output) ;
			return true ;
		} catch (Exception e) {
			throw e ;
		} finally {
			if (input != null) {
				input.close();
			}
			if (output != null) {
				output.close() ;
			}
		}
	}
}
public class JavaAPIDemo {
```

```
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {       // 程序执行出错
            System.out.println("命令执行错误，执行结构：java JavaAPIDemo 拷贝源文件路径 拷贝目标文件路径");
            System.exit(1);
        }
        long start = System.currentTimeMillis() ;
        FileUtil fu = new FileUtil(args[0],args[1]) ;
        System.out.println(fu.copy() ? "文件拷贝成功！" : "文件拷贝失败！");
        long end = System.currentTimeMillis() ;
        System.out.println("拷贝完成的时间：" + (end - start));
    }
}
```

此时千万要注意程序的运行版本问题。那么如果说现在对此程序要求进一步扩展，可以实现一个文件目录的拷贝呢？一旦进行了文件目录的拷贝还需要拷贝所有的子目录中的文件。

范例：文件夹拷贝

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
class FileUtil {  // 定义一个文件操作的工具类
    private File srcFile ; // 源文件路径
    private File desFile ; // 目标文件路径
    public FileUtil(String src,String des) {
        this(new File(src),new File(des)) ;
    }
    public FileUtil(File srcFile,File desFile) {
        this.srcFile = srcFile ;
        this.desFile = desFile ;
    }
    public boolean copyDir() throws Exception {
        try {
            this.copyImpl(this.srcFile) ;
            return true ;
        } catch (Exception e) {
            return false ;
        }
    }
    private void copyImpl(File file) throws Exception {        // 递归操作
        if (file.isDirectory()) {       // 是目录
            File results [] = file.listFiles() ; // 列出全部目录组成
            if (results != null) {
                for (int x = 0 ; x < results.length ; x ++) {
                    copyImpl(results[x]) ;
                }
            }
        } else {     // 是文件
```

```
                String newFilePath = file.getPath().replace(this.srcFile.getPath() +
File.separator, "") ;
                File newFile = new File(this.desFile,newFilePath) ; // 拷贝的目标路径
                this.copyFileImpl(file, newFile) ;
            }
        }
        private boolean copyFileImpl(File srcFile,File desFile) throws Exception {
            if (!desFile.getParentFile().exists()) {
                desFile.getParentFile().mkdirs() ; // 创建父目录
            }
            InputStream input = null ;
            OutputStream output = null ;
            try {
                input = new FileInputStream(srcFile) ;
                output = new FileOutputStream(desFile) ;
                input.transferTo(output) ;
                return true ;
            } catch (Exception e) {
                throw e ;
            } finally {
                if (input != null) {
                    input.close();
                }
                if (output != null) {
                    output.close() ;
                }
            }
        }

        public boolean copy() throws Exception {     // 文件拷贝处理
            if (!this.srcFile.exists()) {  // 源文件必须存在!
                System.out.println("拷贝的源文件不存在！");
                return false ; // 拷贝失败
            }
            return this.copyFileImpl(this.srcFile, this.desFile) ;
        }
}
public class JavaAPIDemo {
        public static void main(String[] args) throws Exception {
            if (args.length != 2) {       // 程序执行出错
                System.out.println("命令执行错误，执行结构：java JavaAPIDemo 拷贝源文件路
径 拷贝目标文件路径");
                System.exit(1);
            }
            long start = System.currentTimeMillis() ;
            FileUtil fu = new FileUtil(args[0],args[1]) ;
            if (new File(args[0]).isFile()) {  // 文件拷贝
                System.out.println(fu.copy() ? "文件拷贝成功！" : "文件拷贝失败！");
            } else {     // 目录拷贝
                System.out.println(fu.copyDir() ? "文件拷贝成功！" : "文件拷贝失败！");
            }
            long end = System.currentTimeMillis() ;
```

```
            System.out.println("拷贝完成的时间：" + (end - start));
        }
}
```

本程序是IO操作的核心代码，本程序可以理解整个的IO处理机制就非常容易理解了。