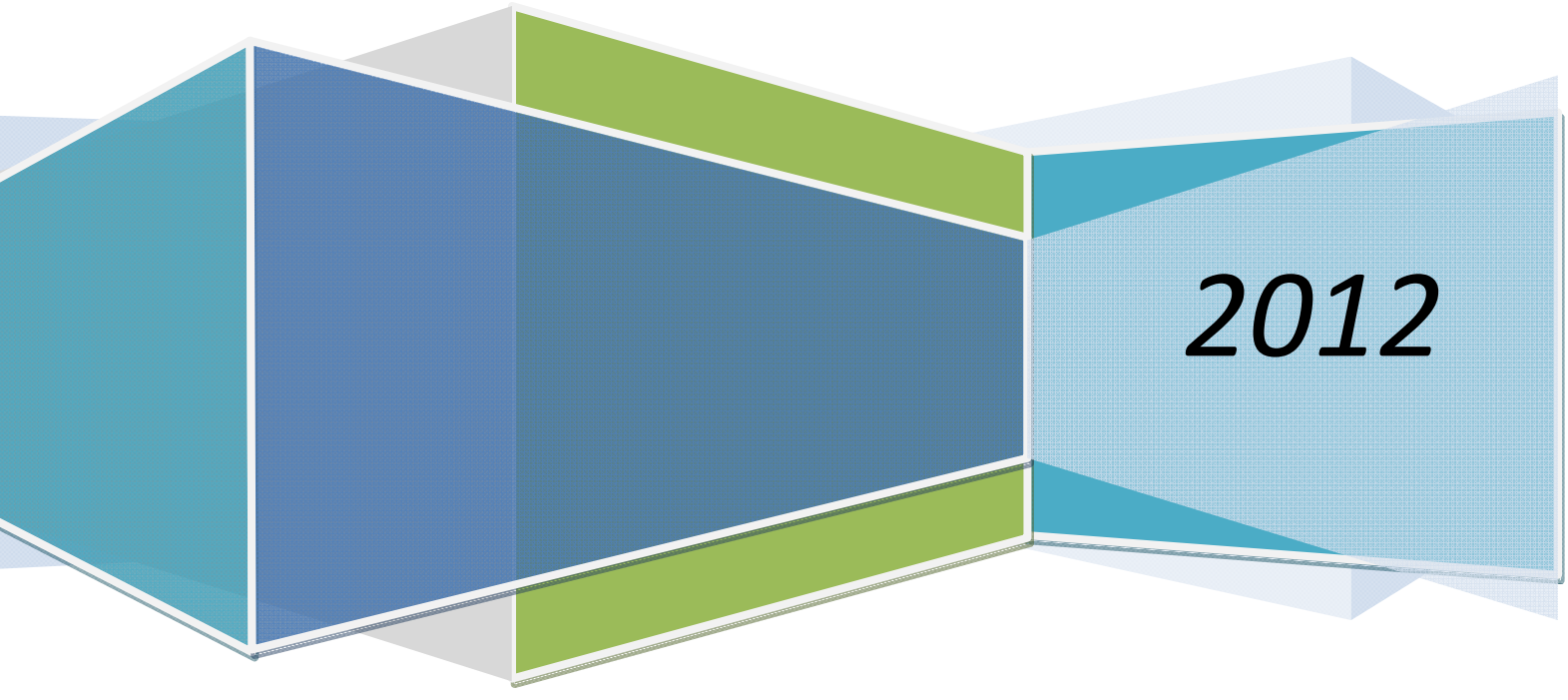
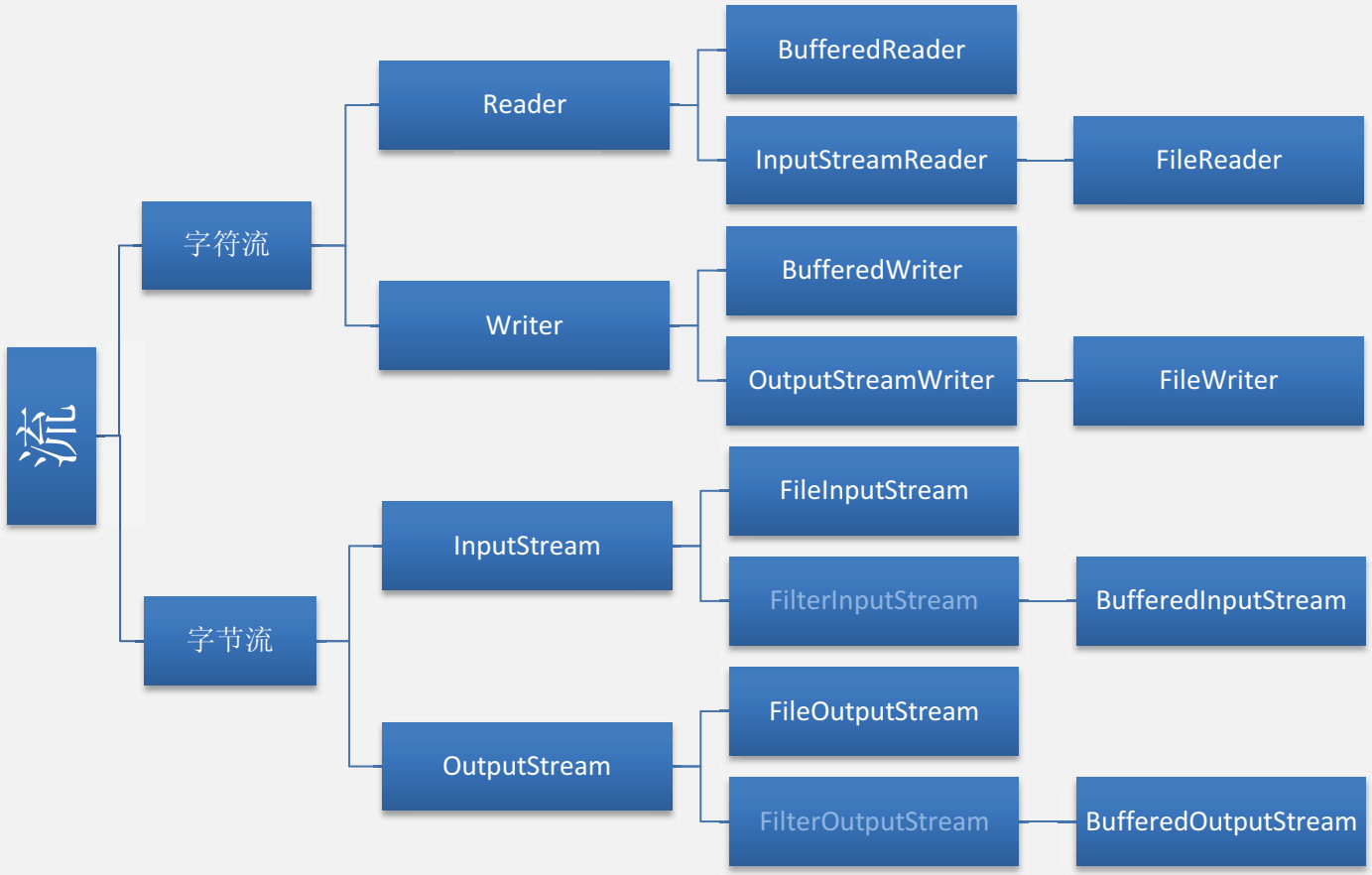
JAVA IO 总结

2012 最新

MyLewis





IO 流对象继承关系

其他常用与流有关的对象：

名称	对应的对象
文件类	File
打印流	PrintStream PrintWriter
管道流	PipedInputStream PipedOutputStream
序列流	SequenceInputStream
对象序列化流	ObjectInputStream ObjectOutputStream

IO 流 学习

毕 向 东 编写
wangboak 整理

IO 流:用于处理设备上的数据。
设备：硬盘，内存，键盘录入。

IO 有具体的分类：
1，根据处理的数据类型不同：字节流和字符流。
2，根据流向不同：输入流和输出流。

字符流的由来：
因为文件编码的不同，而有了对字符进行高效操作的字符流对象。
原理：其实就是基于字节流读取字节时，去查了指定的码表。

字节流和字符流的区别：
1，字节流读取的时候，读到一个字节就返回一个字节。
字符流使用了字节流读到一个或多个字节(中文对应的字节数是两个，在 UTF-8 码表中是 3 个字节)时。先去查指定的编码表，将查到的字符返回。
2，字节流可以处理所有类型数据，如图片，mp3，avi。
而字符流只能处理字符数据。

结论：只要是处理纯文本数据，就要优先考虑使用字符流。除此之外都用字节流。

IO 的体系。所具备的基本功能就有两个：读 和 写。
1，字节流
InputStream（读）,OutputStream（写）。
2，字符流：
Reader（读），Writer（写）。

基本的读写操作方式：
因为数据通常都以文件形式存在。

所以就要找到 IO 体系中可以用于操作文件的流对象。
通过名称可以更容易获取该对象。
因为 IO 体系中的子类名后缀绝大部分是父类名称。而前缀都是体现子类功能的名字。

Reader
|--InputStreamReader
|--FileReader:专门用于处理文件的字符读取流对象。
Writer
|--OutputStreamWriter
|--FileWriter:专门用于处理文件的字符写入流对象。

Reader 中的常见的方法：
1，int read()：
读取一个字符。返回的是读到的那个字符。如果读到流的末尾，返回-1.
2，int read(char[])：
将读到的字符存入指定的数组中，返回的是读到的字符个数，也就是往数组里装的元素的个数。如果读到流的末尾，返回-1.
3，close():
读取字符其实用的是 window 系统的功能，就希望使用完毕后，进行资源的释放。

Writer 中的常见的方法：
1，write(ch): 将一个字符写入到流中。
2，write(char[]): 将一个字符数组写入到流中。
3，write(String): 将一个字符串写入到流中。
4，flush():刷新流，将流中的数据刷新到目的地中，流还存在。
5，close():关闭资源：在关闭前会先调用 flush()，刷新流中的数据去目的地。然流关闭。

FileWriter:
该类没有特有的方法。只有自己的构造函数。
该类特点在于，
1，用于处理文本文件。
2，该类中有默认的编码表，
3，该类中有临时缓冲。

构造函数：在写入流对象初始化时，必须要有一个存储数据的目的地。
FileWriter(String filename):
该构造函数做了什么事情呢？
1，调用系统资源。
2，在指定位置，创建一个文件。
注意：如果该文件已存在，将会被覆盖。
FileWriter(String filename,boolean append):
该构造函数：当传入的 boolean 类型值为 true 时，会在指定文件末尾处进行数据的续写。

FileReader：

1，用于读取文本文件的流对象。

2，用于关联文本文件。

构造函数：在读取流对象初始化的时候，必须要指定一个被读取的文件。

如果该文件不存在会发生 FileNotFoundException.

FileReader(String filename);

清单 1：

1,将文本数据存储到一个文件中。

```
class Demo
{
	public static void main(String[] args) throws IOException
	{
		FileWriter fw = new FileWrier("demo.txt");

		fw.write("abcdec");

		fw.flush();
		fw.write("kkkk");

		fw.close();
	}
}
```

对于读取或者写入流对象的构造函数，以及读写方法，还有刷新关闭功能都会抛出 IOException 或其子类。

所以都要进行处理。或者 throws 抛出，或者 try　catch 处理。

清单 2：

完整的异常处理方式。

```
class Demo
{
	public static void main(String[] args)
	{
		FileWriter fw = null;
		try
		{

			fw = new FileWrier("z:\\demo.txt");
			fw.write("abcdec");
```

```
                fw.flush();
                fw.write("kkkk");

            }
            catch(IOException e)
            {
                System.out.println(e.toString());
            }
            finally
            {
                if(fw!=null)
                    try
                    {
                        fw.close();
                    }
                    catch(IOException e)
                    {
                        System.out.println("close:"+e.toString());
                    }
             }
        }
}
```

另一个小细节：

当指定绝对路径时，定义目录分隔符有两种方式：

1，反斜线 但是一定要写两个。\\ new FileWriter("c:\\demo.txt");

2，斜线 / 写一个即可。new FileWriter("c:/demo.txt");

清单 3：

读取一个已有的文本文件，将文本数据打印出来。

一次读一个字符就打印出来，效率不高。

```
class Demo
{
    public static void main(String[] args) throws IOException
    {
        FileReader fr = new FileReader("demo.txt");

        int ch = 0;
        //一次读一个字符。
```

```
            while((ch=fr.read())!=-1)
            {
                System.out.print((char)ch);
            }

            fr.close();

        }
    }
```

读一个字符就存入字符数组里，读完 1Kb 再打印。

```
    class Demo
    {
        public static void main(String[] args)
        {
            FileReader fr = null;
            try
            {
                fr = new FileReader("demo.txt");

                char[] buf = new char[1024];//该长度通常都是 1024 的整数倍。

                int len = 0;

                while((len=fr.read(buf))!=-1)
                {
                    System.out.println(new String(buf,0,len));
                }

            }
            catch(IOException e)
            {
                System.out.println(e.toString());
            }
            finally
            {
                if(fr!=null)
                    try
                    {
                        fr.close();
                    }
                    catch(IOException e)
                    {
```

```
                    System.out.println("close:"+e.toString());
                }
            }

        }
    }
```

字符流的缓冲区：

缓冲区的出现提高了对流的操作效率。
原理：其实就是将数组进行封装。
对应的对象：
BufferedWriter：
特有方法：
newLine()：跨平台的换行符。
BufferedReader：
特有方法：
readLine():一次读一行，到行标记时，将行标记之前的字符数据作为字符串返回。当读到末尾时，返回 null。

在使用缓冲区对象时，要明确，缓冲的存在是为了增强流的功能而存在，
所以在建立缓冲区对象时，要先有流对象存在。
其实缓冲内部就是在使用流对象的方法，只不过加入了数组对数据进行了临时存储。为了提高操作数据的效率。

代码上的体现：
写入缓冲区对象。

```
//建立缓冲区对象必须把流对象作为参数传递给缓冲区的构造函数。
BufferedWriter bufw = new BufferedWriter(new FileWriter("buf.txt"));
bufw.write("abce");//将数据写入到了缓冲区。
bufw.flush();//对缓冲区的数据进行刷新。将数据刷到目的地中。
bufw.close();//关闭缓冲区，其实关闭的是被包装在内部的流对象。
```

读取缓冲区对象。

```
BufferedReader bufr = new BufferedReader(new FileReader("buf.txt"));
String line = null;



//按照行的形式取出数据。取出的每一个行数据不包含回车符。
while((line=bufr.readLine())!=null)

{
	System.out.println(line);
}
bufr.close();
```

练习：通过缓冲区的形式，对文本文件进行拷贝。

```
public static void main(String[] args)
{
	BufferedReader bufr = new BufferedReader(new FileReader("a.txt"));

	BufferedWriter bufw = new BufferedWriter(new FileWriter("b.txt"));

	String line = null;

	while((line=bufr.readLine())!=null)
	{
		bufw.write(line);
		bufw.newLine();
		bufw.flush();
	}

	bufw.close();
	bufr.close();
}
```

readLine():方法的原理：

其实缓冲区中的该方法，用的还是与缓冲区关联的流对象的 read 方法。
只不过，每一次读到一个字符，先不进行具体操作，先进行临时存储。
当读取到回车标记时，将临时容器中存储的数据一次性返回。

既然明确了原理，我们也可以实现一个类似功能的方法。

```
class MyBufferedReader
{
	private Reader r;

	MyBufferedReader(Reader r)
	{
		this.r = r;
	}
	public String myReadLine()throws IOException
	{
		//1,创建临时容器。
		StringBuilder sb = new StringBuilder();

		//2,循环的使用 read 方法不断读取字符。
		int ch = 0;
		while((ch=r.read())!=-1)
		{
			if(ch=='\r')
				continue;
			if(ch=='\n')
				return sb.toString();
			else
				sb.append((char)ch);
		}
			if(sb.length()!=0)
			return sb.toString();
		return null;
	}
	public void myClose()throws IOException
	{
		r.close();
	}
}
main()
{
	MyBufferedReader myBufr = new MyBufferedReader(new FileReader("a.txt"));
	String line = null;

	while((line=myBufr.myReadLine())!=null)
	{
		System.out.println(line);
	}
}
```

它的出现基于流并增强了流的功能。
这也是一种设计模式的体现：**装饰设计模式**。
对一组对象进行功能的增强。

该模式和继承有什么区别呢？
它比继承有更好的灵活性。

通常装饰类和被装饰类都同属与一个父类或者接口。

```
Writer
    |--MediaWriter
    |--TextWriter
```

（注：MediaWriter 与 TextWtiter 两个类在 JDK 中并不存在，为了更形象的举例说明而“创建”的，不要误解。）
需求：想要对数据的操作提高效率，就用到了缓冲技术。
    通过所学习的继承特性。可以建立子类复写父类中的 write 方法。即可

```
Writer：（注：不要误解，以下两个对象不存在，只为举例。）
    |--MediaWriter
        |--BufferedMediaWriter
    |--TextWriter
        |--BufferedTextWriter
```

当 Writer 中子类对象过多，那么为了提高每一个对象效率，每一个对象都有一个自己的子类 Buffered。
虽然可以实现，但是继承体系变的很臃肿。

那么是否可以对其进行一下优化呢？
其实子类都是在使用缓冲技术。
可不可以对缓冲技术进行描述，将需要增强的对象传递给缓冲区即可。

```
class BufferdWriter
{
    BufferedWriter(MediaWriter mw)
    { }
    BufferedWriter(TextWriter mw)
    { }
}
```

该类虽然完成了对已有两个对象的增强。
但是当有新的对象出现时，还要继续在该类中添加构造函数。这样不利于扩展和维护。
将对这些对象父类型进行操作即可。这就是多态，提高了程序的扩展性。

同时 BufferedWriter 中一样具体 write 方法，只不过是增强后的 write。

所以 BuferedWriter 也应该是 Writer 中的一个子类。

```
class BufferedWriter extends Writer
{
	private Writer w;
	BufferedWriter(Writer w)
	{
		this.w = w;
	}
}
```

```
Writer
	|--MediaWriter
	|--TextWriter
	|--BufferedWriter
```

这样就会发现装饰设计模式，优化增强功能的部分。比继承要灵活很多。

可以在读一行的基础上添加一个行号。

```
class MyLineNumberReader extends MyBufferedReader
{
	private int number;
	MyLineNumberReader(Reader r)
	{
		super(r);
	}
	public String myReadLine()
	{
		number++;
		return super.myReadLine();
	}

	public void setNumber(int number)
	{
		this.number = number;
	}
	public int getNumber()
	{
		return number;
	}
}
```

字节流：

抽象基类：InputStream，OutputStream。

字节流可以操作任何数据。

注意：字符流使用的数组是字符数组。char [] chs
　　字节流使用的数组是字节数组。byte [] bt

```
FileOutputStream fos = new FileOutputStream("a.txt");

fos.write("abcde");//直接将数据写入到了目的地。

fos.close();//只关闭资源。

FileInputStream fis = new FileInputStream("a.txt");

//fis.available();//获取关联的文件的字节数。

//如果文件体积不是很大。
//可以这样操作。

byte[] buf = new byte[fis.available()];//创建一个刚刚好的缓冲区。
//但是这有一个弊端，就是文件过大，大小超出 jvm 的内容空间时，会内存溢出。

fis.read(buf);

System.out.println(new String(buf));
```

需求：copy 一个图片。

```
BufferedInputStream bufis = new BufferedInputStream(new FileInputStream("1.jpg"));

BufferedOutputStream bufos =
      new BufferedOutputStream(new FileOutptStream("2.jpg"));

int by = 0;

while((by=bufis.read())!=-1)
{
      bufos.write(by);
}
bufos.close();
bufis.close();
```

目前学习的流对象：

字符流：

FileReader.

FileWriter.

BufferedReader

BufferedWriter.

字节流：

FileInputStream

FileOutputStream

BufferedInputStream

BufferedOutputStream

字节流的 read()方法读取一个字节。为什么返回的不是 byte 类型，而是 int 类型呢？

因为 read 方法读到末尾时返回的是-1.
而在所操作的数据中的很容易出现连续多个 1 的情况，而连续读到 8 个 1，就是-1.
导致读取会提前停止。

所以将读到的一个字节给提升为一个 int 类型的数值，但是只保留原字节，并在剩余二进制位补 0.
具体操作是：byte&255　or　byte&0xff

对于 write 方法，可以一次写入一个字节，但接收的是一个 int 类型数值。
只写入该 int 类型的数值的最低一个字节（8 位）。

简单说:read 方法对读到的数据进行提升。write 对操作的数据进行转换。

转换流：

特点：

1，是字节流和字符流之间的桥梁。

2，该流对象中可以对读取到的字节数据进行指定编码表的编码转换。

什么时候使用呢?

1，当字节和字符之间有转换动作时。

2，流操作的数据需要进行编码表的指定时。

具体的对象体现：

1，InputStreamReader：字节到字符的桥梁。

2，OutputStreamWriter：字符到字节的桥梁。

这两个流对象是字符流体系中的成员。

那么它们有转换作用，而本身又是字符流。所以在构造的时候，需要传入字节流对象进来。

构造函数：
InputStreamReader(InputStream):通过该构造函数初始化，使用的是本系统默认的编码表 GBK。

InputStreamReader(InputStream,String charSet):通过该构造函数初始化，可以指定编码表。

OutputStreamWriter(OutputStream):通过该构造函数初始化，使用的是本系统默认的编码表 GBK。

OutputStreamWriter(OutputStream,String charSet):通过该构造函数初始化，可以指定编码表。

操作文件的字符流对象是转换流的子类。
Reader
|--InputStreamReader
|--FileReader

Writer
|--OutputStreamWriter
|--FileWriter

转换流中的 read 方法。已经融入了编码表，
在底层调用字节流的 read 方法时将获取的一个或者多个字节数据进行临时存储，
并去查指定的编码表，如果编码表没有指定，
查的是默认码表。那么转流的 read 方法就可以返回一个字符比如中文。

转换流已经完成了编码转换的动作，对于直接操作的文本文件的 FileReaer 而言，就不用在重新定义了，
只要继承该转换流，获取其方法，就可以直接操作文本文件中的字符数据了。

注意：
在使用 FileReader 操作文本数据时，该对象使用的是默认的编码表。
如果要使用指定编码表时，必须使用转换流。

FileReader fr = new FileReader("a.txt");//操作 a.txt 的中的数据使用的本系统默认的 GBK。

操作 a.txt 中的数据使用的也是本系统默认的 GBK。
InputStreamReader isr = new InputStreamReader(new FileInputStream("a.txt"));

这两句的代码的意义相同。

如果 a.txt 中的文件中的字符数据是通过 utf-8 的形式编码。
那么在读取时，就必须指定编码表。
那么转换流必须使用。

InputStreamReader isr = new InputStreamReader(new FileInputStream("a.txt"),"utf-8");

流操作的基本规律。

1，明确数据源和数据汇（数据目的）。
其实是为了明确输入流还是输出流。
2，明确操作的数据是否是纯文本数据。
其实是为了明确字符流还是字节流。

数据源：键盘 System.in,硬盘 File 开头的流对象，内存(数组)。
数据汇：控制台 System.out,硬盘 File 开头的流对象，内存(数组)。

需求：
1，将键盘录入的数据存储到一个文件中。
数据源：System.in

既然是源，使用的就是输入流，可用的体系有 InputStream，Reader。
因为键盘录入进来的一定是纯文本数据，所以可以使用专门操作字符数据的 Reader。
发现 System.in 对应的流是字节读取流。所以要将其进行转换，将字节转成字符即可。
所以要使用 Reader 体系中：InputStreamReader
接下来，是否需要提高效率呢？如果需要，那么就加入字符流的缓冲区：BufferedReader

BufferedReader bur = new BufferedReader(new InputStreamReader(System.in));

数据汇：一个文件，硬盘。

既然是数据汇，那么一定是输出流，可以用的 OutputStream,Writer。
往文件中存储的都是文本数据，那么可以使用字符流较为方便:Writer.

因为操作的是一个文件。所以使用 Writer 中的 FileWriter。
是否要提高效率呢？是，那就使用 BufferedWriter.

```
BufferedWriter bufr = new BufferedWriter(new FileWriter("a.txt"));
```

附加需求：希望将这些文本数据按照指定的编码表存入文件中。
既然是文本数据，而是还是写入到文件中，那么使用的体系还是 Writer。
因为要指定编码表，所以要使用 Writer 中的转换流，OutputStreamWriter。
是否要提高效率，是，选择 BufferedWriter。
注意：虽然是最终是文件，但是不可以选择 FileWriter。因为该对象是使用默认编码表。
输出转换流要接收一个字节输出流进来，所以要是用 OutputStram 体系，而最终输出到一个文件中，
那么就要使用 OutputStream 体系中可以操作的文件的字节流对象：
FileOutputStream。

```
//String charSet = System.getProperty("file.encoding");
String charSet = "utf-8";
BufferedWriter bufw = new BufferedWriter(new OutputStreamWriter(new FileOutputStream("a.txt"),charSet);
```

2，将一个文本文件的数据展示在控制台上。
同上，自己一定要动手。

3，复制文件。
同上，自己一定要动手。
这三个都可以独立完成，并写对过程，和代码。那么 IO 流的操作，哦了！

可以和流相关联的集合对象 Properties.

Map
  |--Hashtable
    |--Properties
Properties:该集合不需要泛型，因为该集合中的键值对都是 String 类型。

1，存入键值对：setProperty(key,value);
2，获取指定键对应的值：value getProperty(key);
3，获取集合中所有键元素：
  Enumeration propertyNames();
  在 jdk1.6 版本给该类提供一个新的方法。
  Set<String> stringPropertyNames();
4，列出该集合中的所有键值对，可以通过参数打印流指定列出到的目的地。
  list(PrintStream);
  list(PrintWriter);
  例：list(System.out):将集合中的键值对打印到控制台。
    list(new PrintStream("prop.txt")):将集合中的键值对存储到 prop.txt 文件中。
5，可以将流中的规则数据加载进行集合，并称为键值对。
  load(InputStream):
  jdk1.6 版本。提供了新的方法。
  load(Reader):
  注意：流中的数据要是 "键=值" 的规则数据。
6，可以将集合中的数据进行指定目的的存储。
  store(OutputStram,String comment)方法。
  jdk1.6 版本。提供了新的方法。
  store(Writer ,String comment):
  使用该方法存储时，会带着当时存储的时间。

练习：记录一个程序运行的次数，当满足指定次数时，该程序就不可以再继续运行了。
  通常可用于软件使用次数的限定。

File 类：

该类的出现是对文件系统的中的文件以及文件夹进行对象的封装。

可以通过对象的思想来操作文件以及文件夹。

1，构造函数：

File(String filename):将一个字符串路径(相对或者绝对)封装成 File 对象，该路径是可存在的，也可以是不存在。

File(String parent,String child);

File(File parent,String child);

2，特别的字段：separator:跨平台的目录分隔符。

例子：File file = new File("c:"+File.separator+"abc"+File.separator+"a.txt");

3，常见方法：

1，创建：

boolean createNewFile()throws IOException:创建文件，如果被创建的文件已经存在，则不创建。

boolean mkdir(): 创建文件夹。

boolean mkdirs(): 创建多级文件夹。

2，删除：

boolean delete():可用于删除文件或者文件夹。

注意：对于文件夹只能删除不带内容的空文件夹，

对于带有内容的文件夹，不可以直接删除，必须要从里往外删除。

void deleteOnExit(): 删除动作交给系统完成。无论是否反生异常，系统在退出时执行删除动作。

3，判断：

boolean canExecute():

boolean canWrite():

boolean canRead();

boolean exists()：判断文件或者文件夹是否存在。

boolean isFile(): 判断 File 对象中封装的是否是文件。

boolean isDirectory():判断 File 对象中封装的是否是文件夹。

boolean isHidden():判断文件或者文件夹是否隐藏。在获取硬盘文件或者文件夹时，

对于系统目录中的文件，java 是无法访问的，所以在遍历，可以避免遍历隐藏文件。

4，获取：

getName():获取文件或者文件夹的名称。
getPath():File 对象中封装的路径是什么，获取的就是什么。
getAbsolutePath():无论 File 对象中封装的路径是什么，获取的都是绝对路径。
getParent(): 获取 File 对象封装文件或者文件夹的父目录。
注意：如果封装的是相对路径，那么返回的是 null.
long length():获取文件大小。
long lastModified()：获取文件或者文件最后一次修改的时间。

static File[] listRoots():获取的是被系统中有效的盘符。
String[] list():获取指定目录下当前的文件以及文件夹名称。
String[] list(Filenamefilter): 可以根据指定的过滤器，过滤后的文件及文件夹名称。
File[] listFiles():获取指定目录下的文件以及文件夹对象。

5，重命名：
renameTo(File):

```
File f1 = new File("c:\\a.txt");
File f2 = new File("c:\\b.txt");
f1.renameTo(f2);//将 c 盘下的 a.txt 文件改名为 b.txt 文件。
```

递归：

其实就是在使用一个功能过程中，又对该功能有需求。
就出现了函数自身调用自身。

注意：
1，一定要限定条件，否则内存溢出。
2，使用递归时，调用次数不要过多，否则也会出现内存溢出。

需求：
想要列出指定目录下的文件以及文件夹中的文件(子文件)。

```
//列出指定目录下的当前的文件或者文件夹。
//想要列出当前目录下的文件夹中的文件，其实就是在重新使用该功能 。
public void listDir(File dir,int level)
{
	System.out.println(getLevel(level)+dir.getName());
	level++;
	File[] files = dir.listFiles();
	for(int x =0; x<files.length; x++)
	{
		if(files[x].isDirecotry())//如果遍历到的是目录。
			listDir(files[x],level);//那么就行该功能的重复使用。递归。
		else
			System.out.println(getLevel(level)+files[x].getName());
	}
}

//想要对列出的目录有一些层次关系。
public String getLevel(int level)
{
	StringBuilder sb = new StringBuilder();
	sb.append("|--");
	for(int x=0; x<level; x++)
	{
		sb.insert(0,"|    ");
	}
	return sb.toString();
}
```

2，需求：删除一个带内容的目录。

原理：从里往外删除，所以需要使用递归完成。

```
public void deleteAll(File dir)
{
	File[] files = dir.listFiles();
	for(int x=0; x<files.length; x++)
	{
		if(files[x].isDirectory())
			deleteAll(files[x]);
		else
			files[x].delete();
	}
	dir.delete();

}
```

IO 包中的常见对象。

字节流：

- FileInputStream
- FileOutputStream
- BufferedInputStream
- BufferedOutputStream

字符流:

- FileReader
- FileWriter
- BufferedReader
- BufferedWriter

转换流:

- InputStreamReader
- OutputStreamWriter

文件对象:

- File

打印流：

- PrintStream
- PrintWriter

所有的带 File 的流对象都可以直接操作 File 对象。

IO 包中的其他对象：

1，打印流。

PrintStream：

是一个字节打印流，System.out 对应的类型就是 PrintStream。

它的构造函数可以接收三种数据类型的值。

1，字符串路径。

2，File 对象。

3，OutputStream。

PrintWriter：

是一个字符打印流。构造函数可以接收四种类型的值。

1，字符串路径。

2，File 对象。

对于 1，2 类型的数据，还可以指定编码表。也就是字符集。

3，OutputStream

4，Writer

对于 3，4 类型的数据，可以指定自动刷新。

注意：该自动刷新值为 true 时，只有三个方法可以用：println,printf,format.

```
//如果想要既有自动刷新，又可执行编码。如何完成流对象的包装？
PrintWrter pw =
    new PrintWriter(new OutputSteamWriter(new FileOutputStream("a.txt"),"utf-8"),true);

//如果想要提高效率。还要使用打印方法。
PrintWrter pw =
new PrintWriter(new BufferdWriter(new OutputSteamWriter(new
    FileOutputStream("a.txt"),"utf-8")),true);
```

2，管道流。

PipedInputStream

PipedOutputStream

特点：

读取管道流和写入管道流可以进行连接。

连接方式：

1，通过两个流对象的构造函数。

2，通过两个对象的 connect 方法。

通常两个流在使用时，需要加入多线程技术，也就是让读写同时运行。

注意；对于 read 方法。该方法是阻塞式的，也就是没有数据的情况，该方法会等待。

参考：Pipedstream.java

3，RandomAccessFile：

该对象并不是流体系中的一员。

该对象中封装了字节流，同时还封装了一个缓冲区(字节数组)，通过内部的指针来操作数组中的数据。

该对象特点：

1，该对象只能操作文件，所以构造函数接收两种类型的参数。

a，字符串路径。

b，File 对象。

2，该对象既可以对文件进行读取，也可以写入。

在进行对象实例化时，必须要指定的该对象的操作模式，r rw 等。

该对象中有可以直接操作基本数据类型的方法。

该对象最有特点的方法：

skipBytes():跳过指定的字节数。

seek()：指定指针的位置。

getFilePointer()：获取指针的位置。

通过这些方法，就可以完成对一个文件数据的随机的访问。

想读哪里就读哪里，想往哪里写就往哪里写。

该对象功能，可以读数据，可以写入数据，如果写入位置已有数据，会发生数据覆盖。也就是可以对数据进行修改。

在使用该对象时，建议数据都是有规则的。或者是分段的。

注意；该对象在实例化时，如果要操作的文件不存在，会自动建立。

如果要操作的文件存在，则不会建立，如果存在的文件有数据。

那么在没有指定指针位置的情况下，写入数据，会将文件开头的数据覆盖。

可以用于多线程的下载，也就是通过多线程往一个文件中同时存储数据。

序列流。也称为合并流。

SequenceInputStream:

特点：可以将多个读取流合并成一个流。这样操作起来很方便。

原理：其实就是将每一个读取流对象存储到一个集合中。最后一个流对象结尾作为这个流的结尾。

两个构造函数：

1，SequenceInputStream(InputStream in1,InputStream in2)

可以将两个读取流合并成一个流。

2，SequenceInputStream(Enumeration<? extends InputStream> en)

可以将枚举中的多个流合并成一个流。

作用：可以用于多个数据的合并。

注意：因为 Enumeration 是 Vector 中特有的取出方式。而 Vector 被 ArrayList 取代。所以要使用 ArrayList 集合效率更高一些。那么如何获取 Enumeration 呢？

```
ArrayList<FileInputStream > al = new ArrayList<FileInputStream>();
for(int x=1; x<4; x++)
	al.add(new FileInputStream(x+".txt"));
	Iterator<FileInputStream> it = al.iterator();

Enumeration<FileInputStream> en = new Enumeration<FileInputStream>()
{
	public boolean hasMoreElements()
	{
		return it.hasNext();
	}
	public FileInputStream nextElement()
	{
		return it.next();
	}
};

//多个流就变成了一个流，这就是数据源。
SequenceInputStream sis = new SequenceInputStream(en);

//创建数据目的。
FileOutputStream fos = new FileOutputStream("4.txt");

byte[] buf = new byte[1024*4];

int len = 0;

while((len=sis.read(buf))!=-1)
{
	fos.write(buf,0,len);
}
fos.close();
sis.close();
```

```
//如果要一个对文件数据切割。

一个读取对应多了输出。
FileInputStream fis = new FileInputStream("1.mp3");

FileOutputStream fos    = null;

byte[] buf = new byte[1024*1024];//是一个 1MB 的缓冲区。

int len = 0;
int count = 1;

while((len=fis.read(buf))!=-1)
{
        fos = new FileOutputStream((count++)+".part);
        fos.write(buf,0,len);

        fos.close();
}
fis.close();

//这样就是将 1.mp3 文件切割成多个碎片文件。

想要合并使用 SequenceInputStream 即可。


对于切割后，合并是需要的一些源文件的信息。
可以通过配置文件进行存储。该配置可以通过键=值的形式存在。
然后通过 Properties 对象进行数据的加载和获取。
```

对象的序列化。

ObjectInputStream
ObjectOutputStream

可以通过这两个流对象直接操作已有对象并将对象进行本地持久化存储。
存储后的对象可以进行网络传输。

两个对象的特有方法：

ObjectInputStream

Object readObject():该方法抛出异常：ClassNotFountException。

ObjectOutputStream

void writeObject(Object)：被写入的对象必须实现一个接口:Serializable

否则会抛出：NotSerializableException

Serializable：该接口其实就是一个没有方法的标记接口。

用于给类指定一个 UID。该 UID 是通过类中的可序列化成员的数字签名运算出来的一个 long 型的值。

只要是这些成员没有变化，那么该值每次运算都一样。

该值用于判断被序列化的对象和类文件是否兼容。

如果被序列化的对象需要被不同的类版本所兼容。可以在类中自定义 UID。

定义方式：static final long serialVersionUID = 42L;

注意：对应静态的成员变量，不会被序列化。

对应非静态也不想被序列化的成员而言，可以通过 transient 关键字修饰。

通常，这两个对象成对使用。

操作基本数据类型的流对象。

DataInputStream

DataInputStream(InputStream);

操作基本数据类型的方法：

int readInt():一次读取四个字节，并将其转成 int 值。

boolean readBoolean():一次读取一个字节。

short readShort();

long readLong();

剩下的数据类型一样。

String readUTF():按照 utf-8 修改版读取字符。注意，它只能读 writeUTF()写入的字符数据。

DataOutputStream

DataOutputStream(OutputStream):

操作基本数据类型的方法:

writeInt(int)：一次写入四个字节。

注意和 write(int)不同。write(int)只将该整数的最低一个 8 位写入。剩余三个 8 位丢弃。

writeBoolean(boolean);

writeShort(short);

writeLong(long);

剩下是数据类型也也一样。

writeUTF(String):按照 utf-8 修改版将字符数据进行存储。只能通过 readUTF 读取。

通常只要操作基本数据类型的数据。就需要通过 DataStram 进行包装。

通常成对使用。

操作数组的流对象。

1，操作字节数组

ByteArrayInputStream

ByteArrayOutputStream

toByteArray();

toString();

writeTo(OutputStream);

2，操作字符数组。

CharArrayReader

CharArrayWriter

对于这些流，源是内存。目的也是内存。

而且这些流并未调用系统资源。使用的就是内存中的数组。
所以这些在使用的时候不需要 close。

操作数组的读取流在构造时，必须要明确一个数据源。所以要传入相对应的数组。

对于操作数组的写入流，在构造函数可以使用空参数。因为它内置了一个可变长度数组作为缓冲区。

这几个流的出现其实就是通过流的读写思想在操作数组。

类似的对象同理：
StringReader，
StringWriter。

编码转换：

在 io 中涉及到编码转换的流是转换流和打印流。
但是打印流只有输出。

在转换流中是可以指定编码表的。
默认情况下，都是本机默认的码表。GBK. 这个编码表怎么来的？
System.getProperty("file.encoding");

常见码表：
ASCII:美国标准信息交换码。使用的是 1 个字节的 7 位来表示该表中的字符。

ISO8859-1:拉丁码表。使用 1 个字节来表示。

GB2312:简体中文码表。
GBK：简体中文码表，比 GB2312 融入更多的中文文件和符号。

unicode:国际标准码表。都用两个字节表示一个字符。
UTF-8：对 unicode 进行优化，每一个字节都加入了标识头。

编码转换：
字符串 -->字节数组 ：编码。通过 getBytes(charset);
字节数组-->字符串 ： 解码。通过 String 类的构造函数完成。String(byte[],charset);

如果编错了，没救！
如果编对了，解错了，有可能还有救！

```
String s = "你好";
//编码。
byte[] b = s.getBytes("GBK");

//解码。
String s1 = new String(b,"iso8859-1");
System.out.println(s1);//????
```

```
//想要还原。
/*
对 s1 先进行一次解码码表的编码。获取原字节数据。
然后在对原字节数据进行指定编码表的解码。
*/
byte[] b1 = s1.getBytes("iso8859-1");

String s2 = new String(b1,"gbk");

System.out.println(s2);//你好。
```

这种情况在 tomcat 服务器会出现。
因为 tomcat 服务器默认是 iso8859-1 的编码表。
所以客户端通过浏览器向服务端通过 get 提交方式提交中文数据时，
服务端获取到会使用 ISO8859-1 进行中文数据的解码。会出现乱码。
这时就必须要对获取的数据进行 iso8859-1 编码。然后在按照页面指定的编码表进行解码即可

而对于 post 提交，这种方法也通用。但是 post 有更好的解决方式。
request.setCharacterEncoding("utf-8");即可。
所以建立客户端提交使用 post 提交方式。