

# Java I/O

## 本节目标

1. File文件操作类
2. 字节流与字符流
3. 转换流
4. 字符编码
5. 内存操作流
6. 打印流
7. System类对IO的支持
8. 两种输入流
9. 序列化

如果要学好IO，必须清楚抽象类；IO的操作部分重点掌握两个代码模型。IO的核心组成就是五个类(File、OutputStream、InputStream、Reader、Writer)一个接口(Serializable)

## 1. File文件操作类

在Java.io包之中，File类是唯一一个与文件本身操作(创建、删除、取得信息..)有关的程序类。

### 1.1 File类使用

java.io.File类是一个普通的类，直接产生实例化对象即可。如果要实例化对象则需要使用到两个构造方法：

- public File(String pathname);
- public File(String parent, String child)，设置父路径和子路径。

如果要想进行文件的基本操作，可以使用File类的如下方法：

- 创建一个新文件：

```
public boolean createNewFile() throws IOException
```

范例：创建新文件

```
package www.bit.java.io;

import java.io.File;
import java.io.IOException;

public class TestIO {

    public static void main(String[] args) {
        // 定义要操作的文件路径
    }
}
```

```

File file = new File("/Users/yuisama/Desktop/TestIO.java");
try {
    file.createNewFile() ;
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}
}

```

File类只是创建文件本身，但是对于其内容并不做处理。

- 判断文件是否存在:

```
public boolean exists()
```

- 删除文件:

```
public boolean delete()
```

范例：编写文件的基本操作(如果文件不存在则进行创建；存在则删除)

```

package www.bit.java.io;

import java.io.File;
import java.io.IOException;

public class TestIO {

    public static void main(String[] args) throws IOException {
        // 定义要操作的文件路径
        File file = new File("/Users/yuisama/Desktop/TestIO.java");
        if (file.exists()) {
            // 文件存在，进行删除
            file.delete() ;
        }else {
            file.createNewFile() ;
        }
    }
}

```

以上实现了最简化的文件处理操作，但是代码存在两个问题：

- 实际项目部署环境可能与开发环境不同。那么这个时候路径的问题就很麻烦了。windows下使用的是"\"，而Unix系统下使用的是"/"。所以在使用路径分隔符时都会采用File类的一个常量"public static final String separator "来描述。

```
// separator由不同操作系统下的JVM来决定到底是哪个杠杠!
File file = new File(File.separator + "Users" + File.separator + "yuisama" +
File.separator + "Desktop"
+ File.separator + "TestIO.java");
```

- 在Java中要进行文件的处理操作是要通过本地操作系统支持的，在这之中如果操作的是同名文件，就可能出现延迟的问题。（开发之中尽可能避免文件重名问题）

## 1.2 目录操作

File类中关于目录有如下方法：

- 取得父路径或父File对象：

```
public String getParent()
```

```
public File getParentFile()
```

若想创建父路径，此时最好取得父路径的File类对象。

- 创建目录(无论有多少级父目录，都会创建)

```
public boolean mkdirs()
```

范例：Java文件目录操作

```
package www.bit.java.io;

import java.io.File;
import java.io.IOException;

public class TestIO {

    public static void main(String[] args) throws IOException {
        // separator由不同操作系统下的JVM来决定到底是哪个杠杠!
        File file = new File(File.separator + "Users" + File.separator
+ "yuisama" + File.separator + "Desktop"
+ File.separator +
"javaIO"+File.separator+"bit"+File.separator+"TestIO.java");
        if (!file.getParentFile().exists()) { // 创建父目录
            file.getParentFile().mkdirs() ; // 有多少级父目录就创建多
少级
        }
        if (file.exists()) {
            // 文件存在，进行删除
            file.delete();
        } else {
            file.createNewFile();
        }
    }
}
```

```
        }  
    }  
}
```

以上判断父目录是否存在以及父目录的创建操作非常重要，对于后续开发很重要。

## 1.3 文件信息

在File类里面提供有一系列取得文件信息的操作：

1. 判断路径是否是文件: `public boolean isFile()`
2. 判断路径是否是目录: `public boolean isDirectory()`
3. 取得文件大小(字节): `public long length()`
4. 最后一次修改日期 : `public long lastModified()`

范例：取得文件信息

```
package www.bit.java.io;  
  
import java.io.File;  
import java.io.IOException;  
import java.util.Date;  
  
public class TestIO {  
  
    public static void main(String[] args) throws IOException {  
        // 要操作的文件  
        File file = new File(File.separator + "Users" + File.separator  
+ "yuisama" + File.separator + "Desktop"  
        + File.separator + "test.png");  
        // 保证文件存在再进行操作  
        if (file.exists() && file.isFile()) {  
            System.out.println("文件大小: " + file.length());  
            System.out.println("最后一次修改日期: " + new  
Date(file.lastModified()));  
        }  
    }  
}
```

以上操作都是针对文件进行信息取得，Java里也提供有如下方法列出一个目录的全部组成：`public File[] listFiles()`

范例：列出目录中的全部组成

```
package www.bit.java.io;  
  
import java.io.File;
```

```

import java.io.IOException;

public class TestIO {

    public static void main(String[] args) throws IOException {
        // 要操作的文件
        File file = new File(File.separator + "Users" + File.separator
+ "yuisama" + File.separator + "Desktop");
        // 保证是个目录且存在
        if (file.exists() && file.isDirectory()) {
            // 列出目录中的全部内容
            File[] result = file.listFiles() ;
            for (File file2 : result) {
                System.out.println(file2);
            }
        }
    }
}

```

以上这些操作可以取得本地文件的相关信息

## 1.4 综合案例（目录列表）

虽然File提供有listFiles()方法，但是这个方法本身只能够列出本目录中的第一级信息。如果要求列出目录中所有级的信息，必须自己来处理。这种操作就必须通过递归的模式来完成。

范例：递归打印当前目录下所有层级的文件信息

```

package www.bit.java.io;

import java.io.File;

public class TestIO {

    public static void main(String[] args) {
        File file = new File(File.separator + "Users" + File.separator
+ "yuisama" + File.separator + "Desktop") ;
        listAllFiles(file) ; // 从此处开始递归
    }
    /**
     *
     * this methods is used for 列出指定目录中的全部子目录信息 yuisama 2017年11
月27日
     *
     * @param file
     */
    public static void listAllFiles(File file) {
        if (file.isDirectory()) { // 现在给定的file对象属于目录

```

```

        File[] result = file.listFiles() ; // 继续列出子目录内容
        if (result != null) {
            for (File file2 : result) {
                listAllFiles(file2) ;
            }
        }
    }else {
        // 给定的file是文件，直接打印
        System.out.println(file) ;
    }
}
}

```

### 线程阻塞问题：

现在所有代码都是在main线程下完成的，如果listAllFiles()方法没有完成，那么对于main后续的执行将无法完成。这种耗时的操作让主线程出现了阻塞，而导致后续代码无法正常执行完毕。如果不想让阻塞产生，最好再产生一个新的线程进行处理。

范例：新增子线程进行耗时操作

```

package www.bit.java.io;

import java.io.File;

public class TestIO {

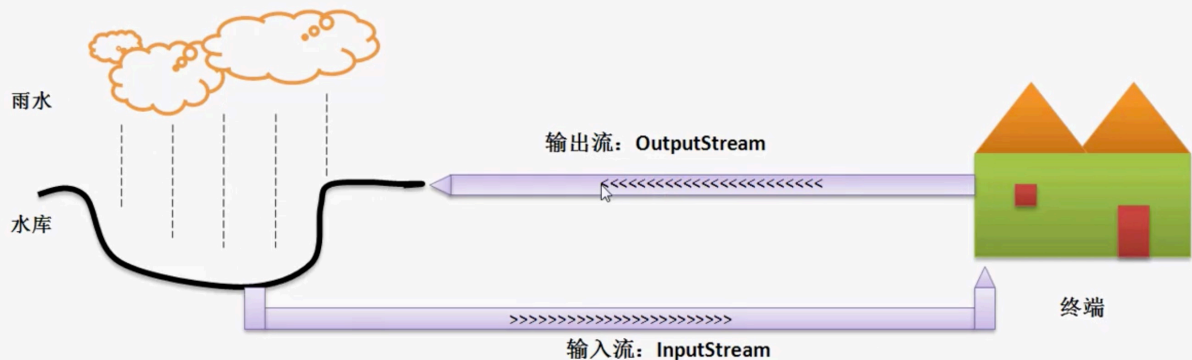
    public static void main(String[] args) {
        // 开启子线程进行列出处理
        new Thread()->{
            File file = new File(File.separator + "Users" +
File.separator + "yuisama" + File.separator + "Desktop") ;
            listAllFiles(file) ; // 从此处开始递归
        },"输出线程").start();
        System.out.println("开始进行文件输出...");
    }
    /**
     *
     * this methods is used for 列出指定目录中的全部子目录信息 yuisama 2017年11
月27日
     *
     * @param file
     */
    public static void listAllFiles(File file) {
        if (file.isDirectory()) { // 现在给定的file对象属于目录
            File[] result = file.listFiles() ; // 继续列出子目录内容
            if (result != null) {
                for (File file2 : result) {

```



# 输入流与输出流

## ➤ 字节操作流



# 输入流与输出流

## ➤ 字符操作流



字节流与字符流操作的本质区别只有一个：字节流是原生的操作，而字符流是经过处理后的操作。

在进行网络数据传输、磁盘数据保存所保存所支持的数据类型只有：字节。

而所有磁盘中的数据必须先读取到内存后才能进行操作，而内存中会帮助我们帮我们把字节变为字符。字符更加适合处理中文。

不管使用的是字节流还是字符流，其基本的操作流程几乎是一样的，以文件操作为例。

1. 根据文件路径创建File类对象；
2. 根据字节流或字符流的子类实例化父类对象；
3. 进行数据的读取或写入操作
4. 关闭流(close())。

对于IO操作属于资源处理，所有的资源处理操作(IO操作、数据库操作、网络)最后必须要进行关闭。

## 2.2 字节输出流(OutputStream)



如果要想通过程序进行内容输出，则可以使用java.io.OutputStream。

来观察OutputStream类的定义结构：

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

OutputStream类实现了Closeable, Flushable两个接口，这两个接口中的方法：

1. Closeable: public void close() throws IOException;
2. Flushable: public void flush() throws IOException;

在OutputStream类中还定义有其他方法：

1. 将给定的字节数组内容全部输出： public void write(byte b[]) throws IOException
2. 将部分字节数组内容输出： public void write(byte b[], int off, int len) throws IOException
3. 输出单个字节： public abstract void write(int b) throws IOException;

由于OutputStream是一个抽象类，所以要想为父类实例化，就必须使用子类。由于方法名称都由父类声明好了，所以我们在此处只需要关系子类的构造方法。如果要进行文件的操作，可以使用FileOutputStream类来处理，这个类的构造方法如下：

1. 接收File类（覆盖）： public FileOutputStream(File file) throws FileNotFoundException
2. 接收File类（追加）： public FileOutputStream(File file, boolean append)

范例：实现文件的内容输出

```
package www.bit.java.io;

import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;

public class TestIO {
    public static void main(String[] args) throws Exception{
        File file = new File(File.separator + "Users" + File.separator
+ "yuisama" + File.separator + "Desktop"
        + File.separator + "hello.txt");
        if (!file.getParentFile().exists()) { // 必须保证父目录存在
            file.getParentFile().mkdirs() ; // 创建多级父目录
        }
        // OutputStream是一个抽象类，所以需要通过子类进行实例化，此时只能操作
        File类

        OutputStream output = new FileOutputStream(file) ;
        // 要求输出到文件的内容
        String msg = "比特科技Java公开课" ;
        // 将内容变为字节数组
        output.write(msg.getBytes());
        // 关闭输出
        output.close();
    }
}
```

在进行文件输出的时候，所有的文件会自动帮助用户创建，不在需要调用createFile()方法手工创建。

这个时候程序如果重复执行，并不会出现内容追加的情况而是一直在覆盖。如果需要文件内容追加，则需要调用FileOutputStream提供的另外一种构造方法。

范例：文件内容追加

```
package www.bit.java.io;

import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;

public class TestIO {
    public static void main(String[] args) throws Exception{
        File file = new File(File.separator + "Users" + File.separator
+ "yuisama" + File.separator + "Desktop"
        + File.separator + "hello.txt");
        if (!file.getParentFile().exists()) { // 必须保证父目录存在
            file.getParentFile().mkdirs(); // 创建多级父目录
        }
        // OutputStream是一个抽象类，所以需要通过子类进行实例化，此时只能操作
File类

        OutputStream output = new FileOutputStream(file,true) ;
        // 要求输出到文件的内容
        String msg = "比特科技Java公开课\n" ;
        // 将内容变为字节数组
        output.write(msg.getBytes());
        // 关闭输出
        output.close();
    }
}
```

范例：部分内容输出

```
package www.bit.java.io;

import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;

public class TestIO {
    public static void main(String[] args) throws Exception{
        File file = new File(File.separator + "Users" + File.separator
+ "yuisama" + File.separator + "Desktop"
        + File.separator + "hello.txt");
```

File类

```
        if (!file.getParentFile().exists()) { // 必须保证父目录存在
            file.getParentFile().mkdirs(); // 创建多级父目录
        }
        // OutputStream是一个抽象类，所以需要通过子类进行实例化，此时只能操作

        OutputStream output = new FileOutputStream(file,true);
        // 要求输出到文件的内容
        String msg = "比特科技Java公开课\n";
        // 将内容变为字节数组
        output.write(msg.getBytes(),0,5);
        // 关闭输出
        output.close();
    }
}
```

## 2.3 AutoCloseable自动关闭支持

从Jdk1.7开始追加了一个AutoCloseable接口，这个接口的主要目的是自动进行关闭处理，但是这种处理一般不好用，因为使用它必须结合try..catch

范例：观察AutoCloseable接口使用

```
package www.bit.java.io;

class Message implements AutoCloseable {
    public Message() {
        System.out.println("创建一条新的消息");
    }
    @Override
    public void close() throws Exception {
        System.out.println("[AutoCloseable]自动关闭方法");
    }
    public void print() {
        System.out.println("www.bit.java");
    }
}

public class TestAutoClose {
    public static void main(String[] args) {
        try(Message msg = new Message()) { // 必须在try中定义对象
            msg.print();
        } catch (Exception e) {
        }
    }
}
```

这种自动关闭处理是需要结合try语句进行调用。

范例：使用自动关闭处理之前的操作

```
package www.bit.java.io;

import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;

public class TestIO {
    public static void main(String[] args) throws Exception{
        File file = new File(File.separator + "Users" + File.separator
+ "yuisama" + File.separator + "Desktop"
        + File.separator + "hello.txt");
        if (!file.getParentFile().exists()) { // 必须保证父目录存在
            file.getParentFile().mkdirs(); // 创建多级父目录
        }
        // OutputStream是一个抽象类，所以需要通过子类进行实例化，此时只能操作
        File类
        try(OutputStream output = new FileOutputStream(file,true)){
            // 要求输出到文件的内容
            String msg = "比特科技Java公开课\n" ;
            // 将内容变为字节数组
            output.write(msg.getBytes());
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

是否去使用由大家决定，但是一般不推荐使用，因为语法结构比较混乱，还是推荐大家使用close方法手工关闭资源。

## 2.3 字节输入流：InputStream

利用了OutputStream实现了程序输出内容到文件的处理，下面使用InputStream类在程序中读取文件内容。InputStream类的定义如下：

```
public abstract class InputStream implements Closeable
```

发现InputStream类只实现了Closeable接口，在InputStream类中提供有如下方法：

1. 读取数据到字节数组中,返回数据的读取个数。如果此时开辟的字节数组大小大于读取的数据大小，则返回的就是读取个数；如果要读取的数据大于数组的内容，那么这个时候返回的就是数组长度；如果没有数据了还在读，则返回-1: **public int read(byte b[]) throws IOException.最常用方法**
2. 读取部分数据到字节数组中，每次只读取传递数组的部分内容，如果读取满了则返回长度(len),如果没有读取满则返回读取的数据个数，如果读取到最后没有数据了返回-1: **public int**

read(byte b[], int off, int len) throws IOException

3. 读取单个字节，每次读取一个字节的內容，直到没有数据了返回-1： public abstract int read() throws IOException;

同OutputStream的使用一样，InputStream是一个抽象类，如果要对其实例化，同样也需要使用子类。如果要对文件进行处理，则使用FileInputStream类。

范例：实现文件信息的读取

```
package www.bit.java.io;

import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;

public class TestInputStream {

    public static void main(String[] args) throws Exception {
        // 1.定义文件路径
        File file = new File(File.separator + "Users" + File.separator
+ "yuisama" + File.separator + "Desktop"
+ File.separator + "hello.txt");
        // 2.必须保证文件存在才能进行处理
        if (file.exists()) {
            InputStream input = new FileInputStream(file) ;
            byte[] data = new byte[1024] ; // 每次可以读取的最大数量
            int len = input.read(data) ; // 此时的数据读取到了数组之中
            String result = new String(data,0,len) ; // 将字节数组转
为String
            System.out.println("读取内容【"+result+"】") ;
            input.close();
        }
    }
}
```

整个的操作流程可以发现OutputStream、InputStream类的使用形式上是非常类似的。

## 2.4 字符输出流：Writer

字符适合于处理中文数据，Writer是字符输出流的处理类，这个类的定义如下：

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

与OutputStream相比多了一个Appendable接口。

在Writer类里面也提供write()方法，而且该方法接收的类型都是char型，要注意的是，Writer类提供了一个直接输出字符串的方法：

```
public void write(String str) throws IOException
```

如果要操作文件使用FileWriter子类。

范例：通过Writer实现输出

```
package www.bit.java.io;

import java.io.File;
import java.io.FileWriter;
import java.io.Writer;

public class TestWriter {

    public static void main(String[] args) throws Exception {
        File file = new File(File.separator + "Users" + File.separator
+ "yuisama" + File.separator + "Desktop"
+ File.separator + "hello.txt");
        if (!file.getParentFile().exists()) { // 必须保证父目录存在
            file.getParentFile().mkdirs(); // 创建多级父目录
        }
        String msg = "比特科技java公开课" ;
        Writer out = new FileWriter(file) ;
        out.write(msg) ;
        out.close() ;
    }
}
```

Writer类的结构与方法的使用与OutputStream非常相似，只是Writer类对于中文的支持很好并且提供了直接写入String的方法而已。

## 2.5 字符输入流：Reader

Reader依然也是一个抽象类。如果要进行文件读取，同样的，使用FileReader。

在上面讲到的Writer类中提供有方法直接向目标源写入字符串，而在Reader类中没有方法可以直接读取字符串类型，这个时候只能通过字符数组进行读取操作

范例：通过文件读取数据

```
package www.bit.java.io;

import java.io.File;
import java.io.FileReader;
import java.io.Reader;

public class TestReader {
```

```

    public static void main(String[] args) throws Exception{
        // 1.定义文件路径
        File file = new File(File.separator + "Users" + File.separator
+ "yuisama" + File.separator + "Desktop"
        + File.separator + "hello.txt");
        // 2.必须保证文件存在才能进行处理
        if (file.exists()) {
            Reader in = new FileReader(file) ;
            char[] data = new char[1024] ;
            int len = in.read(data) ; // 将数据读取到字符数组中
            String result = new String(data, 0, len) ;
            System.out.println("读取内容【"+result+"】") ;
            in.close();
        }
    }
}

```

字符流适合处理中文，字节流适合处理一切数据类型（对中文支持不好）

## 2.6 字节流vs字符流

通过上述这一系列流的讲解可以发现，使用字节流和字符流从代码形式上区别不大。但是如果从实际开发来讲，字节流一定是优先考虑的，只有在处理中文时才会考虑字符流。因为所有的字符都需要通过内存缓冲来进行处理。

所有字符流的操作，无论是写入还是输出，数据都先保存在缓存中。

范例：示范字节流输出与字符流输出区别

如果字符流不关闭，数据就有可能保存在缓存中并没有输出到目标源。这种情况下就必须强制刷新才能够得到完整数据。

范例：字符流刷新操作

```

package www.bit.java.io;

import java.io.File;
import java.io.FileWriter;
import java.io.Writer;

public class TestWriter {

    public static void main(String[] args) throws Exception {
        File file = new File(File.separator + "Users" + File.separator
+ "yuisama" + File.separator + "Desktop"
        + File.separator + "hello.txt");
        if (!file.getParentFile().exists()) { // 必须保证父目录存在
            file.getParentFile().mkdirs() ; // 创建多级父目录
        }
    }
}

```

```

    }
    String msg = "比特科技java公开课\n" ;
    Writer out = new FileWriter(file,true) ;
    out.write(msg) ;
    out.flush(); // 写上此语句表示强制清空缓冲内容，所有内容都输出。
}

}

```

在以后进行IO处理的时候，如果处理的是图片、音乐、文字都可以使用字节流，而只有处理中文的时候才会使用字符流

## 3.转换流

### 3.1 转换流的基本使用

现在为止已经知道了两种数据流：字节流和字符流。实际上这两泪流是可以进行互相转换处理的。

- OutputStreamWriter:将字节输出流变为字符输出流（Writer对于文字的输出来要比OutputStream方便）
- InputStreamReader:将字节输入流变为字符输入流（InputStream读取的是字节，不方便中文的处理）

要想知道这两个类的实际意义，我们首先来看这两个类的继承关系以及构造方法：

```

public class OutputStreamWriter extends Writer

public OutputStreamWriter(OutputStream out)

```

```

public class InputStreamReader extends Reader

public InputStreamReader(InputStream in)

```

范例：观察字节流与字符流的转换

```

package www.bit.java.io;

import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;

public class TestOutputStreamWriter {
    public static void main(String[] args) throws Exception{
        File file = new File(File.separator + "Users" + File.separator
+ "yuisama" + File.separator + "Desktop"
+ File.separator + "hello.txt");
    }
}

```



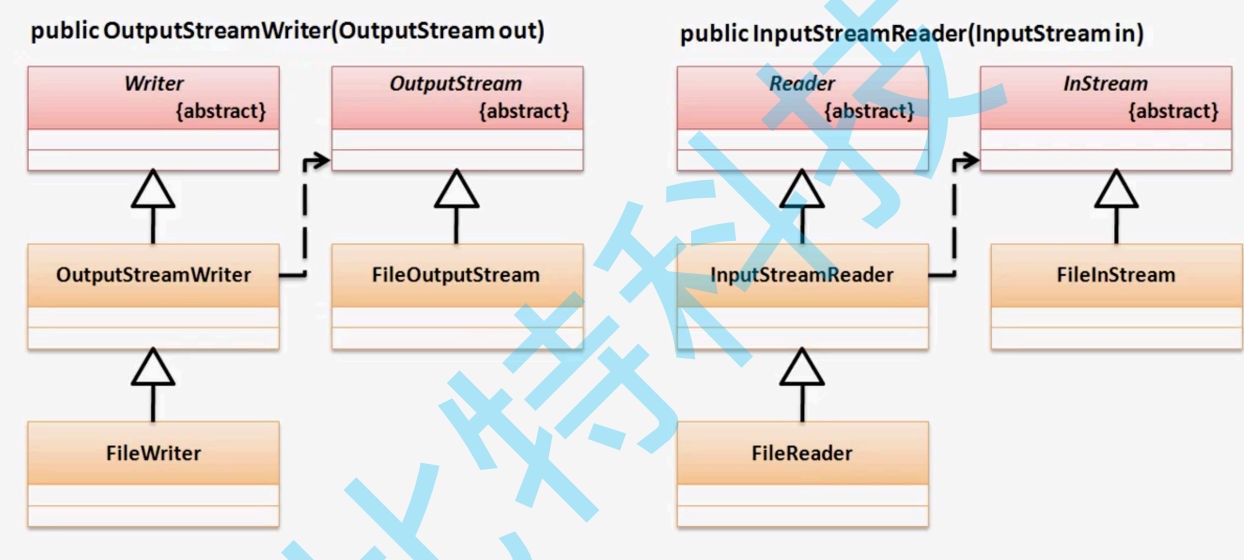
```

        if (!file.getParentFile().exists()) { // 必须保证父目录存在
            file.getParentFile().mkdirs(); // 创建多级父目录
        }
        OutputStream output = new FileOutputStream(file);
        Writer out = new OutputStreamWriter(output); // 字节流转为字符流
        String msg = "你好啊 比特";
        out.write(msg);
        out.close();
    }
}

```

这种操作在实际开发中并没有什么意义，我们主要用它来分析FileOutputStream、FileInputStream、FileWriter、FileReader之间的继承关系。

## OutputStreamWriter、InputStreamReader



从整个继承结构来讲，发现所有字符流处理的时候是经过转换后得来的。

### 3.2 综合演练：文件拷贝（重要）

linux下文件拷贝命令:"cp 源文件路径 目标文件路径"

现在希望通过程序来实现这样的操作。即，建立一个CopyFile程序类，这个类通过初始化参数接收源文件与目标文件路径。

分析：

1. 要想实现数据的拷贝肯定是要通过流的方式来完成，对于流有两类，由于要拷贝的内容不一定是文字数据，所以此处我们采用字节流。
2. 在进行拷贝的时候需要确定模式：a.在程序中开辟一个数组，该数组长度为文件长度，将所有数据一次性读取到该数组中随后进行输出保存。b.采用同边读边写的方式完成。

范例：初期模型

```
package www.bit.java.io;
```

```

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

/**
 * This class is used for 建立一个专门负责文件拷贝处理的类, 该类具备如下功能: 1.判断拷
 * 贝的源文件是否存在
 * 2.判断目标文件的父路径是否存在, 如果不存在则创建父目录 3.进行文件拷贝的处理
 *
 * @author yuisama created on 2017年11月28日
 */
class CopyFileUtil { // 此时这个工具类不需要任何属性, 建议将构造方法私有化, 并且文件操作
方法均为类方法
    private CopyFileUtil() {
    } // 构造方法私有化

    /**
     * this methods is used for 判断要拷贝的源路径是否存在 yuisama 2017年11月28
     *
     * @param path
     *          输入的源路径信息
     * @return 如果该路径真实存在返回true, 否则返回false
     */
    public static boolean fileIsExists(String path) {
        return new File(path).exists();
    }

    /**
     * this methods is used for 根据传入的路径判断父路径是否存在, 如果不存在则创建
     * yuisama 2017年11月28日
     *
     * @param path
     *          输出的目标地址, 根据此地址判断父路径是否存在。不存在则创建
     */
    public static void createParentsDir(String path) {
        File file = new File(path);
        if (!file.getParentFile().exists()) { // 路径不存在
            file.getParentFile().mkdirs(); // 创建多级父目录
        }
    }

    /**
     *
     * this methods is used for 文件拷贝 yuisama 2017年11月28日

```

```

*
* @param sourcePath
*         源文件路径
* @param destPath
*         目标文件路径
* @return 是否拷贝成功
*/
public static boolean copyFile(String sourcePath, String destPath) {
    File inFile = new File(sourcePath) ;
    File outFile = new File(destPath) ;
    FileInputStream fileInputStream = null ;
    FileOutputStream fileOutputStream = null ;
    try {
        fileInputStream = new FileInputStream(inFile) ;
        fileOutputStream = new FileOutputStream(outFile) ;
        copyFileHandle(fileInputStream, fileOutputStream) ; //

```

完成具体文件拷贝处理

```

        } catch (IOException e) {
            e.printStackTrace() ;
            return false ;
        } finally {
            try {
                fileInputStream.close() ;
                fileOutputStream.close() ;
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        return true;
    }
}
/**
*
* this methods is used for 实现具体的文件拷贝操作
* yuisama
* 2017年11月28日
* @param inputStream 输入流对象
* @param outputStream 输出流对象
*/
private static void copyFileHandle(InputStream
inputStream,OutputStream outputStream) throws IOException {
    long start = System.currentTimeMillis() ;
    // InputStream有读取单个字节的方法
    // OutputStream有输出单个字节的方法
    int temp = 0 ;
    do {
        temp = inputStream.read() ; // 读取单个字节数据
        outputStream.write(temp) ; // 通过输出流输出
    } while (temp != -1); // 如果有数据继续读取

```

```

        long end = System.currentTimeMillis() ;
        System.out.println("拷贝文件所花费的时间: "+(end-start)) ;
    }
}

public class CopyFile {
    public static void main(String[] args) {
        if (args.length != 2) { // 现在参数不是两个
            System.out.println("非法操作, 命令为: java CopyFile 源文件
路径 目标文件路径");
            return;
        }
        String sourcePath = args[0]; // 取得源文件路径
        String destPath = args[1]; // 取得目标路径
        if (CopyFileUtil.fileExists(sourcePath)) {
            CopyFileUtil.createParentsDir(sourcePath); // 创建目录
            System.out.println(CopyFileUtil.copyFile(sourcePath,
destPath) ? "文件拷贝成功" : "文件拷贝失败");
        } else {
            System.out.println("源文件不存在, 无法进行拷贝");
        }
    }
}

```

这个时候的确成功执行了，但是这个代码有两个问题：

1. 在开发里尽量不要去使用do..while，尽量去使用while。
2. 拷贝的速度特别慢

范例：解决do..while

```

        while((temp = inputStream.read()) != -1 ) { // 判断这个读取后的字节(保存在temp)中是否为-1,如果不是表示有内容
            outputStream.write(temp) ;
        }
    }
}

```

以上的方式还是针对于一个字节的方式完成的，如果文件太大，这种做法实在太慢。

范例：解决读取慢的问题

如果要想解决读取慢的问题，那么就要一次性读取多个字节内容

```

package www.bit.java.io;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;

```

```

import java.io.OutputStream;

/**
 * This class is used for 建立一个专门负责文件拷贝处理的类，该类具备如下功能： 1.判断拷贝的源文件是否存在
 * 2.判断目标文件的父路径是否存在，如果不存在则创建父目录 3.进行文件拷贝的处理
 *
 * @author yuisama created on 2017年11月28日
 */
class CopyFileUtil { // 此时这个工具类不需要任何属性，建议将构造方法私有化，并且文件操作方法均为类方法
    private CopyFileUtil() {
    } // 构造方法私有化

    /**
     * this methods is used for 判断要拷贝的源路径是否存在 yuisama 2017年11月28日
     *
     * @param path
     *          输入的源路径信息
     * @return 如果该路径真实存在返回true，否则返回false
     */
    public static boolean fileIsExists(String path) {
        return new File(path).exists();
    }

    /**
     * this methods is used for 根据传入的路径判断父路径是否存在，如果不存在则创建
     yuisama 2017年11月28日
     *
     * @param path
     *          输出的目标地址，根据此地址判断父路径是否存在。不存在则创建
     */
    public static void createParentsDir(String path) {
        File file = new File(path);
        if (!file.getParentFile().exists()) { // 路径不存在
            file.getParentFile().mkdirs(); // 创建多级父目录
        }
    }

    /**
     *
     * this methods is used for 文件拷贝 yuisama 2017年11月28日
     *
     * @param sourcePath
     *          源文件路径
     * @param destPath
     *          目标文件路径
     * @return 是否拷贝成功

```

```

    */
    public static boolean copyFile(String sourcePath, String destPath) {
        File inFile = new File(sourcePath) ;
        File outFile = new File(destPath) ;
        FileInputStream fileInputStream = null ;
        FileOutputStream fileOutputStream = null ;
        try {
            fileInputStream = new FileInputStream(inFile) ;
            fileOutputStream = new FileOutputStream(outFile) ;
            copyFileHandle(fileInputStream, fileOutputStream) ; //
            完成具体文件拷贝处理
        } catch (IOException e) {
            e.printStackTrace() ;
            return false ;
        } finally {
            try {
                fileInputStream.close() ;
                fileOutputStream.close() ;
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        return true;
    }
    /**
     *
     * this methods is used for 实现具体的文件拷贝操作
     * yuisama
     * 2017年11月28日
     * @param inputStream 输入流对象
     * @param outputStream 输出流对象
     */
    private static void copyFileHandle(InputStream
inputStream,OutputStream outputStream) throws IOException {
        long start = System.currentTimeMillis() ;
        // InputStream有读取单个字节的方法
        // OutputStream有输出单个字节的方法
        byte[] data = new byte[1024] ; // 开辟缓冲区一次性读入多个内容

        int len = 0 ;
        // len = inputStream.read(data)) != -1,表示将数据读取到字节数组之
        中,而后返回读取个数
        while((len = inputStream.read(data)) != -1) {
            outputStream.write(data,0,len) ; // 将字节数组的部分内容写
            到目标文件中
        }
        long end = System.currentTimeMillis() ;
        System.out.println("拷贝文件所花费的时间: "+(end-start)) ;
    }

```

```

    }
}

public class CopyFile {
    public static void main(String[] args) {
        if (args.length != 2) { // 现在参数不是两个
            System.out.println("非法操作, 命令为: java CopyFile 源文件
路径 目标文件路径");
            return;
        }
        String sourcePath = args[0]; // 取得源文件路径
        String destPath = args[1]; // 取得目标路径
        if (CopyFileUtil.fileIsExists(sourcePath)) {
            CopyFileUtil.createParentsDir(sourcePath); // 创建目录
            System.out.println(CopyFileUtil.copyFile(sourcePath,
destPath) ? "文件拷贝成功" : "文件拷贝失败");
        } else {
            System.out.println("源文件不存在, 无法进行拷贝");
        }
    }
}

```

这个程序最为核心的部分就是:

```

byte[] data = new byte[1024] ; // 开辟缓冲区一次性读入多个内容
int len = 0 ;
// len = inputStream.read(data)) != -1, 表示将数据读取到字节数组之
中, 而后返回读取个数
while((len = inputStream.read(data)) != -1) {
    outputStream.write(data, 0, len) ; // 将字节数组的部分内容写
到目标文件中
}

```

这也是以后用到最多的代码（后续在文件上传之中都是这种模式）

如果现在要求你处理的数据都在InputStream里面，就采用以上的模型。

## 4. 字符编码

### 4.1 常用字符编码

在计算机的世界里面，所有的文字都是通过编码来描述的。对于编码而言，如果没有正确的解码，那么就会产生乱码。

那么要想避免乱码问题，就必须清楚常见的编码有哪些

1. GBK、GB2312：表示的是国标编码，GBK包含简体中文和繁体中文，而GB2312只包含简体中文。也就是说，这两种编码都是描述中文的编码。

2. UNICODE编码: java提供的16进制编码, 可以描述世界上任意的文字信息, 但是有个问题, 如果现在所有的字母也都使用16进制编码, 那么这个编码太庞大了, 会造成网络传输的负担。
3. ISO8859-1: 国际通用编码, 但是所有的编码都需要进行转换。
4. UTF编码: 相当于结合了UNICODE、ISO8859-1, 也就是说需要使用到16进制文字使用UNICODE, 而如果只是字母就使用ISO8859-1, 而常用的就是**UTF-8**编码形式。

在以后的开发之中使用的编码只有一个:UTF-8编码

## 4.2 乱码产生分析

清楚了常用编码后, 下面就可以观察一下乱码的产生。要想观察出乱码, 就必须首先知道当前操作系统中默认支持的编码是什么(java默认编码)

范例: 读取java运行属性

```
System.getProperties().list(System.out);
```

如果说现在本地系统所用的编码与程序所用编码不同, 那么强制转换就会出现乱码。

范例: 观察乱码产生

```
package www.bit.java.testthread;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.UnsupportedEncodingException;

public class Test {
    public static void main(String[] args) throws
        UnsupportedEncodingException, IOException {
        OutputStream out = new FileOutputStream(new
        File("/Users/yuisama/Desktop/hello.txt"));
        out.write("比特欢迎您".getBytes("ISO8859-1"));
        out.close();
    }
}
```

乱码产生的本质: 编码和解码不统一产生的问题。

以后就使用UTF-8

## 5.内存操作流

### 5.1 内存流概念



在之前所有的操作都是针对于文件进行的IO处理。除了文件之外，IO的操作也可以发生在内存之中，这种流称之为内存操作流。文件流的操作里面一定会产生一个文件数据(不管最后这个文件数据是否被保留)。

如果现在需求是：需要进行IO处理，但是又不希望产生文件。这种情况下就可以使用内存作为操作终端。

对于内存流也分为两类：

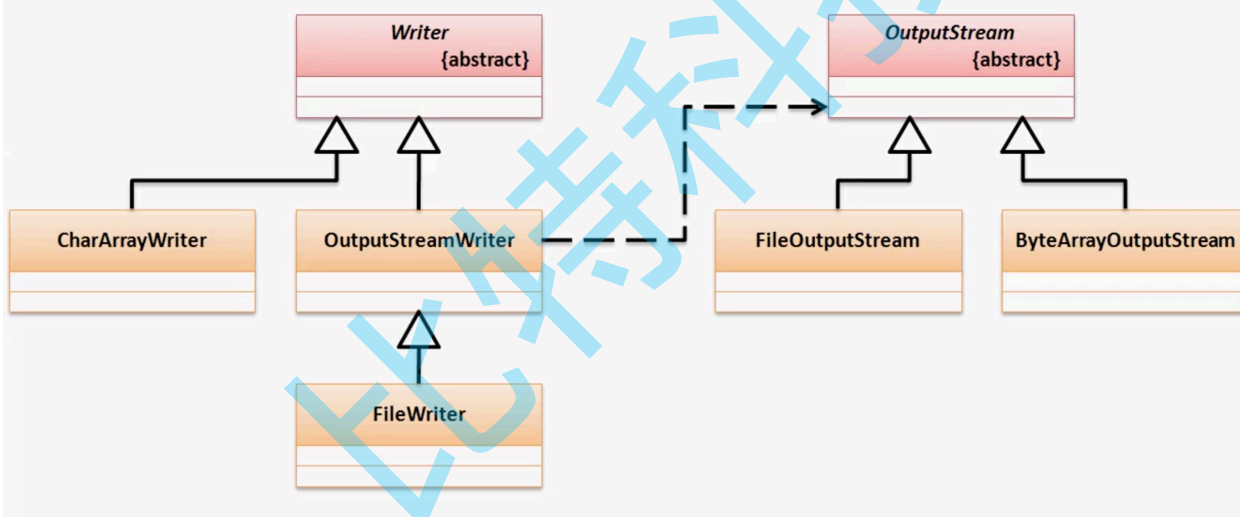
1. 字节内存流:ByteArrayInputStream、ByteArrayOutputStream
2. 字符内存流:CharArrayReader、CharArrayWriter

首先来观察ByteArrayInputStream和ByteArrayOutputStream的构造方法：

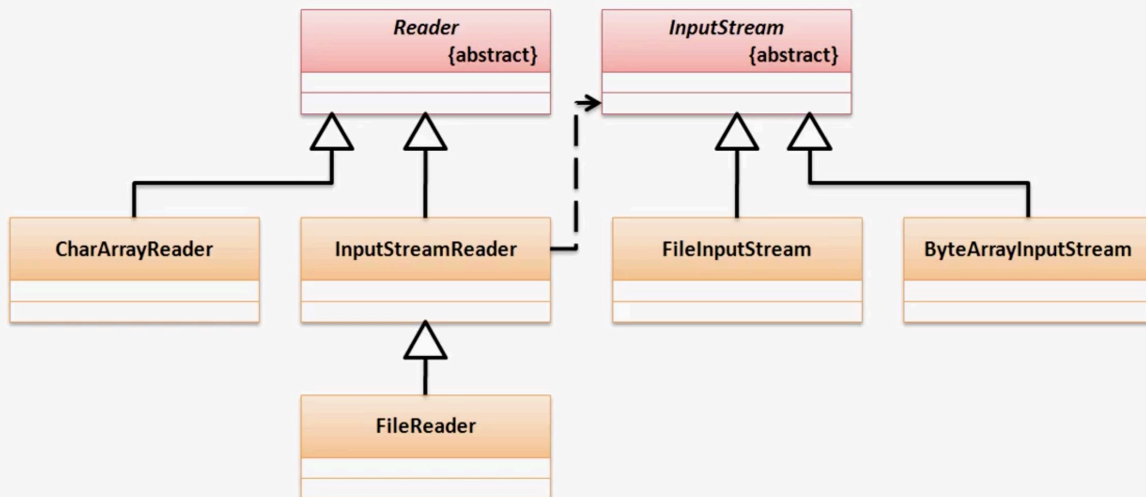
```
public ByteArrayInputStream(byte buf[])  
  
public ByteArrayOutputStream()
```

接下来看内存流的继承关系：

## 内存流继承关系



## 内存流继承关系



范例：通过内存流实现大小写转换。

```
package www.bit.java.test.intern;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class TestDemo {
    public static void main(String[] args) throws IOException {
        String msg = "hello world";
        // 实例化InputStream类对象，实例化的时候需要把你操作的数据保存到内存之中
        // 最终读取的就是你设置的内容。
        InputStream input = new ByteArrayInputStream(msg.getBytes());
        OutputStream output = new ByteArrayOutputStream();
        int temp = 0;
        while((temp = input.read()) != -1) {
            // 每个字节进处理,处理之后所有数据都在OutputStream类中
            output.write(Character.toUpperCase(temp));
        }
        // 直接输出output对象
        System.out.println(output);
        input.close();
        output.close();
    }
}
```

这个时候发生了IO操作，但是没有文件产生，可以理解为一个临时文件处理。

## 5.2 内存流操作

内存操作流还有一个很小的功能，可以实现两个文件的合并处理(文件量不大)。

内存操作流最为核心的部分就是：将所有OutputStream输出的程序保存在了程序里面，所以可以通过这一特征实现处理。

需求：现在有两个文件：data-a.txt、data-b.txt。现在要求将这两个文件的内容做一个合并处理。

范例：内存流实现文件合并处理

```
package www.bit.java.test.intern;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class TestDemo {
    public static void main(String[] args) throws IOException {
        File[] files = new File[] { new
File("/Users/yuisama/Desktop/data-a.txt"),
        new File("/Users/yuisama/Desktop/data-b.txt")
};

        String[] data = new String[2] ;
        for (int i = 0; i < files.length ; i++) {
            data[i] = readFile(files[i]) ;
        }
        StringBuffer buf = new StringBuffer() ; // 组合操作
        String contentA [] = data[0].split(" ") ;
        String contentB [] = data[1].split(" ") ;
        for (int i = 0; i < contentA.length ; i++) {
            buf.append(contentA[i]).append("
").append(contentB[i]).append(" ") ;
        }
        System.out.println(buf);
    }

    /**
     *
     * this methods is used for 读取文件内容。使用File对象因为其包含有完整的路径信息
     * yuisama 2017年11月30日
     *
     * @param file
     * @return
     * @throws IOException
     */
    public static String readFile(File file) throws IOException {
        if (file.exists()) {
```

```

        InputStream input = new FileInputStream(file);
        // 没有向上转型，因为稍后要使用到toByteArray()方法。
        ByteArrayOutputStream bos = new
ByteArrayOutputStream();
        int temp = 0;
        byte[] data = new byte[10];
        while ((temp = input.read(data)) != -1) {
            // 将数据保存在bos中
            bos.write(data, 0, temp);
        }
        bos.close();
        input.close();
        // 将读取内容返回
        return new String(bos.toByteArray());
    }
    return null;
}
}

```

如果只是使用InputStream类，在进行数据完整读取的时候会很不方便，结合内存流的使用会好很多。

## 6.打印流

打印流解决的就是OutputStream的设计缺陷，属于OutputStream功能的加强版。如果操作的不是二进制数据，只是想通过程序向终端目标输出信息的话，OutputStream不是很方便，其缺点有两个：

1. 所有的数据必须转换为字节数组。
2. 如果要输出的是int、double等类型就不方便了

### 6.1 打印流概念

打印流设计的主要目的是为了了解决OutputStream的设计问题，其本质不会脱离OutputStream。

范例：自己设计一个简单打印流

```

package www.bit.java.test.intern;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;

class PrintUtil {
    private OutputStream out ;
    // 由外部传入要输出的目标终端
    public PrintUtil(OutputStream out) {
        this.out = out ;
    }
}

```

```

// 核心功能就一个
public void print(String str) {
    try {
        this.out.write(str.getBytes());
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

public void println(String str) {
    this.print(str+"\n");
}

public void print(int data) {
    this.print(String.valueOf(data));
}

public void println(int data) {
    this.println(String.valueOf(data));
}

public void print(double data) {
    this.print(String.valueOf(data));
}

public void println(double data) {
    this.println(String.valueOf(data));
}
}

public class TestPrint {
    public static void main(String[] args) throws Exception {
        PrintUtil printUtil = new PrintUtil(new FileOutputStream(new
File("/Users/yuisama/Desktop/test.txt")));
        printUtil.print("姓名:");
        printUtil.println("yuisama");
        printUtil.print("年龄: ");
        printUtil.println(27);
        printUtil.print("工资: ");
        printUtil.println(0.0000000000000001);
    }
}

```

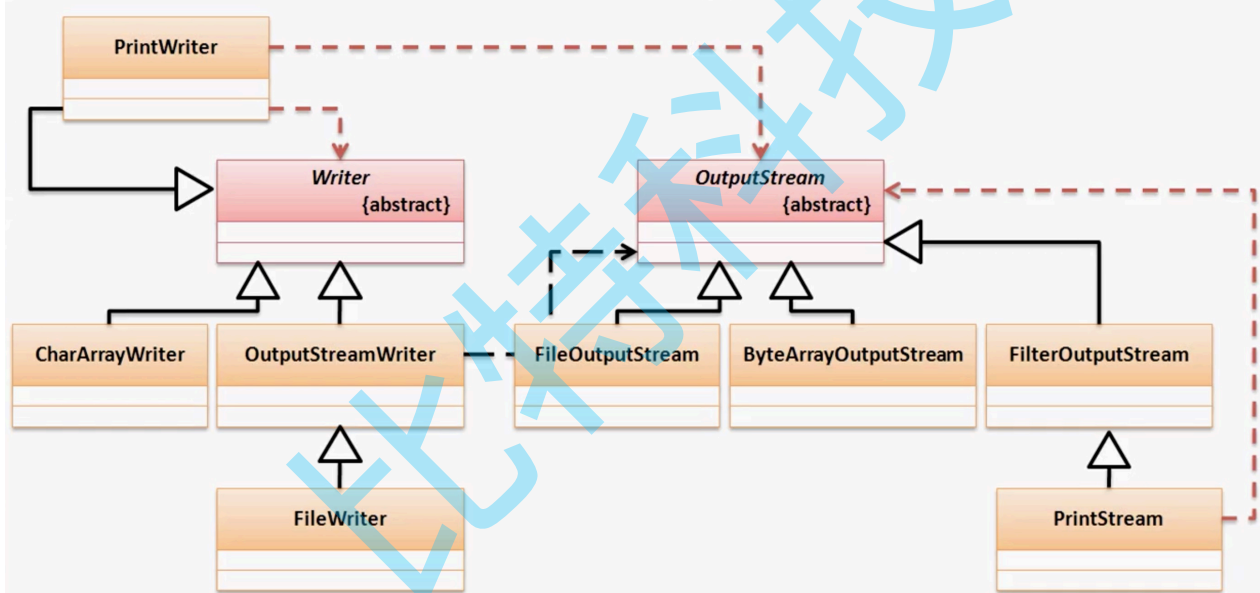
经过简单处理之后，让OutputStream的功能变的更加强大了，其实本质就只是对OutputStream的功能做了一个封装而已。java提供有专门的打印流处理类:PrintStream、PrintWriter

## 6.2 使用系统提供的打印流

打印流分为字节打印流: PrintStream、字符打印流:PrintWriter，以后使用PrintWriter几率较高。首先来观察这两个类的继承结构与构造方法:

<u>PrintStream 类:</u>	<u>PrintWriter 类:</u>
<u>java.lang.Object</u> →  - <u>java.io.OutputStream</u> → →  - <u>java.io.FilterOutputStream</u> → → →  - <u>java.io.PrintStream</u>	<u>java.lang.Object</u> →  - <u>java.io.Writer</u> → →  - <u>java.io.PrintWriter</u>
public <u>PrintStream</u> ( <u>OutputStream out</u> )	public <u>PrintWriter</u> ( <u>OutputStream out</u> )  public <u>PrintWriter</u> ( <u>Writer out</u> )

## PrintStream、PrintWriter继承结构



此时看上图继承关系我们会发现，有点像之前讲过的代理设计模式，但是代理设计模式有如下特点：

1. 代理是以接口为使用原则的设计模式。
2. 最终用户可以调用的方法一定是接口定义的方法。

打印流的设计属于装饰设计模式：核心依然是某个类的功能，但是为了得到更好的操作效果，让其支持的功能更多一些。

范例：使用打印流

```

public class TestPrint {
    public static void main(String[] args) throws Exception {
        PrintWriter printUtil = new PrintWriter(new
        FileOutputStream(new File("/Users/yuisama/Desktop/test.txt"))) ;
        printUtil.print("姓名:") ;
        printUtil.println("yuisama") ;
        printUtil.print("年龄: ") ;
        printUtil.println(27) ;
        printUtil.print("工资: ") ;
        printUtil.println(0.0000000000000001) ;
        printUtil.close();
    }
}

```

以后的开发之中一定会用到打印流。

## 6.3 格式化输出

C语言有一个printf()函数，这个函数在输出的时候可以使用一些占位符，例如：字符串(%s)、数字(%d)、小数(%m.nf)、字符(%c)等。从JDK1.5开始，PrintStream类中也追加了此种操作。

格式化输出:

```

public PrintStream printf(String format, Object ... args)

```

范例：观察格式化输出

```

public class TestPrint {
    public static void main(String[] args) throws Exception {
        String name = "yuisama" ;
        int age = 25 ;
        double salary = 1.10700000000001 ;
        PrintWriter printUtil = new PrintWriter(new
        FileOutputStream(new File("/Users/yuisama/Desktop/test.txt"))) ;
        printUtil.printf("姓名: %s,年龄: %d,工资:%1.2f", name,age,salary)
        ;
        printUtil.close();
    }
}

```

同时在String类中也追加有一个格式化字符串方法。

格式化字符串:

```

public static String format(String format, Object... args)

```

范例：格式化字符串

```

public class TestPrint {
    public static void main(String[] args) throws Exception {
        String name = "yuisama" ;
        int age = 25 ;
        double salary = 1.10700000000001 ;
        String str = String.format("姓名: %s,年龄: %d,工资:%1.2f",
name,age,salary, name,age,salary) ;
        System.out.println(str);
    }
}

```

Ajax都是打印流支撑的

## 7.System类对I/O的支持

学习完PrintStream与PrintWriter后，我们发现里面的方法名都很熟悉。例如：print()、println()，实际上我们一直在使用的系统输出就是利用了IO流的模式完成。在System类中定义了三个操作的常量。

1. 标准输出（显示器）：public final static PrintStream out
2. 错误输出：public final static PrintStream err
3. 标准输入(键盘):public final static InputStream in

一直在使用的System.out.println()属于IO的操作范畴

### 7.1 系统输出

系统输出一共有两个常量:out、err,并且这两个常量表示的都是PrintStream类的对象。

1. out输出的是希望用户能看到的内容
2. err输出的是不希望用户看到的内容

这两种输出在实际的开发之中都没用了，取而代之的是"日志"。

```

public class TestPrint {
    public static void main(String[] args) throws Exception {
        try {
            Integer.parseInt("abc") ;
        } catch (Exception e) {
            System.out.println(e) ;
            System.err.println(e) ;
        }
    }
}

```

System.err只是作为一个保留的属性而存在，现在几乎用不到。唯一可能用到的就是System.out。

由于System.out是PrintStream的实例化对象，而PrintStream又是OutputStream的子类，所以可以直接使用System.out直接为OutputStream实例化，这个时候的OutputStream输出的位置将变为屏幕。

范例：使用System.out为OutputStream实例化。



```

public class TestPrint {
    public static void main(String[] args) throws Exception {
        OutputStream out = System.out ;
        out.write("比特欢迎您".getBytes());
    }
}

```

抽象类不同的子类针对于同一方法有不同的实现，而用户调用的时候核心参考的是抽象类。

## 7.2 系统输入

System.in对应的类型是InputStream，而这种输入流指的是由用户通过键盘进行输入(用户输入)。java本身并没有直接的用户输入处理，如果要想实现这种操作，必须使用java.io的模式来完成。

范例：利用InputStream实现数据输入

```

public class TestPrint {
    public static void main(String[] args) throws Exception {
        InputStream in = System.in ;
        byte[] data = new byte[1024] ;
        System.out.print("请输入信息:");
        int temp = in.read(data) ; // 数据读取到字节数组中
        System.out.println("输出内容为：" + new String(data,0,temp)) ;
    }
}

```

现在发现当用户输入数据的时候程序需要暂停执行，也就是程序进入了阻塞状态。直到用户输入完成(按下回车)，程序才能继续向下执行。

以上的程序本身有一个致命的问题，核心点在于：开辟的字节数组长度固定，如果现在输入的长度超过了字节数组长度，那么只能接收部分数据。这个时候是由于一次读取不完所造成的问题，所以此时最好的做法是引入内存操作流来进行控制，这些数据先保存在内存流中而后一次取出。

范例:引入内存流。

```

package www.bit.java.test.intern;

import java.io.ByteArrayOutputStream;
import java.io.InputStream;

public class TestPrint {
    public static void main(String[] args) throws Exception {
        InputStream in = System.in ;
        ByteArrayOutputStream bos = new ByteArrayOutputStream() ;
        byte[] data = new byte[10] ;
        System.out.print("请输入信息:");
        int temp = 0 ;
        while((temp = in.read(data)) != -1) {
            bos.write(data,0,temp) ; // 保存数据到内存输出流中
        }
    }
}

```

```

        // 这里面需要用户判断是否输入结束
        if (temp < data.length) {
            break ;
        }
    }
    in.close() ;
    bos.close() ;
    System.out.println("输出内容为 :" +new
String(bos.toByteArray())) ;
}
}

```

现在虽然实现了键盘输入数据的功能，但是整体的实现逻辑过于混乱了，即java提供的System.in并不好用,还要结合内存流来完成，复杂度很高。

如果要想在IO中进行中文的处理，最好的做法是将所有输入的数据保存在一起再处理，这样才可以保证不出现乱码。

## 8.两种输入流

### 8.1 BufferedReader类

BufferedReader类属于一个缓冲的输入流，而且是一个字符流的操作对象。在java中对于缓冲流也分为两类：字节缓冲流(BufferedInputStream)、字符缓冲流(BufferedReader)。

之所以选择BufferedReader类操作是因为在此类中提供有如下方法（读取一行数据）：

```
String readLine() throws IOException
```

这个方法可以直接读取一行数据(以回车为换行符)

但是这个时候有一个非常重要的问题要解决，来看BufferedReader类的定义与构造方法：

```

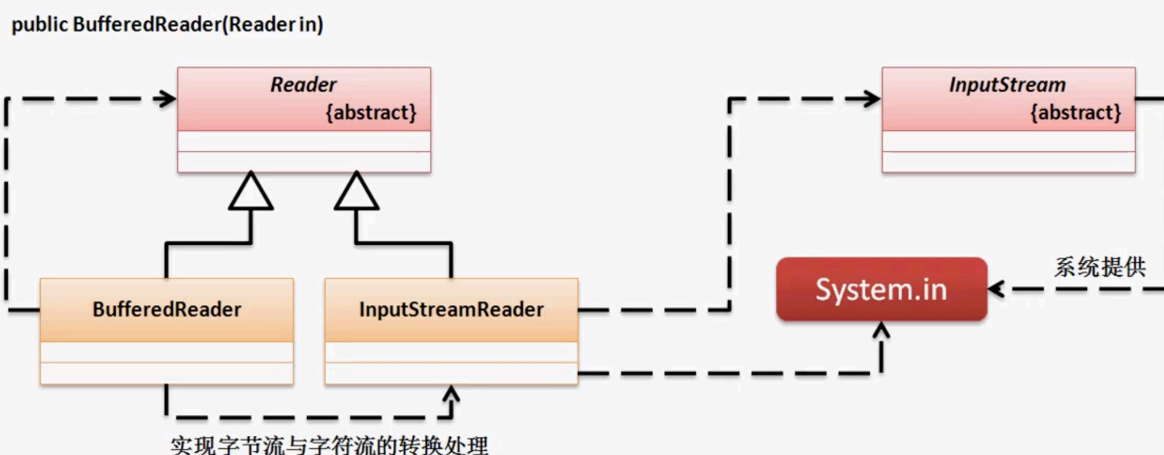
public class BufferedReader extends Reader

    public BufferedReader(Reader in)

```

而System.in是InputStream类的子类，这个时候与Reader没有关系，要建立起联系就要用到InputStreamReader类。如下：

## BufferedReader类



### 范例：利用BufferedReader实现键盘输入

```
package www.bit.java.test.intern;

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class TestPrint {
    public static void main(String[] args) throws Exception {
        BufferedReader buf = new BufferedReader(new
InputStreamReader(System.in)) ;
        System.out.println("请输入信息 :") ;
        // 默认的换行模式是BufferedReader的最大缺点
        String str = buf.readLine() ; // 默认使用回车换行
        System.out.println("输入信息为:" + str );
    }
}
```

以上操作形式是java10多年前输入的标准格式，但是时过境迁，这个类也淹没在历史的潮流之中，被JDK1.5提供的java.util.Scanner类所取代。

使用以上形式实现的键盘输入还有一个最大特点，由于接收的数据类型为String，可以使用String类的各种操作进行数据处理并且可以变为各种常见数据类型。

## 8.2 java.util.Scanner类

打印流解决的是OutputStream类的缺陷，BufferedReader解决的是InputStream类的缺陷。而Scanner解决的是BufferedReader类的缺陷(替换了BufferedReader类)

Scanner是一个专门进行输入流处理的程序类，利用这个类可以方便处理各种数据类型，同时也可以直接结合正则表达式进行各项处理，在这个类中主要关注以下方法:

1. 判断是否有指定类型数据: `public boolean hasNextXxx()`

2. 取得指定类型的数据: public 数据类型 nextXxx()
3. 定义分隔符: public Scanner useDelimiter(Pattern pattern)
4. 构造方法: public Scanner(InputStream source)

范例：使用Scanner实现数据输入

```
package www.bit.java.test.intern;

import java.util.Scanner;

public class TestPrint {
    public static void main(String[] args) throws Exception {
        Scanner scanner = new Scanner(System.in) ;
        System.out.println("请输入数据: ") ;
        if (scanner.hasNext()) { // 有输入内容,不判断空字符串
            System.out.println("输入内容为: "+scanner.next());
        }
        scanner.close() ;
    }
}
```

使用Scanner还可以接收各种数据类型，并且帮助用户减少转型处理。

范例：接收其他类型数据

```
package www.bit.java.test.intern;

import java.util.Scanner;

public class TestPrint {
    public static void main(String[] args) throws Exception {
        Scanner scanner = new Scanner(System.in) ;
        System.out.println("请输入年龄: ") ;
        if (scanner.hasNextInt()) { // 有输入内容,不判断空字符串
            int age = scanner.nextInt() ;
            System.out.println("输入内容为: "+ age );
        }else {
            System.out.println("输入的不是数字!");
        }
        scanner.close() ;
    }
}
```

最为重要的是，Scanner可以对接收的数据类型使用正则表达式判断

范例:利用正则表达式进行判断

```
package www.bit.java.test.intern;
```

```

import java.util.Scanner;

public class TestPrint {
    public static void main(String[] args) throws Exception {
        Scanner scanner = new Scanner(System.in) ;
        System.out.println("请输入生日: ") ;
        if (scanner.hasNext("\\d{4}-\\d{2}-\\d{2}")) {
            String birthday = scanner.next() ;
            System.out.println("输入的生日为:" + birthday);
        }else {
            System.out.println("输入的格式非法, 不是生日");
        }
        scanner.close() ;
    }
}

```

但是以上操作在开发之中基本不会出现, 现在不可能让你编写一个命令程序进行数据输入。

使用Scanner本身能够接收的是一个InputStream对象, 那么也就意味着可以接收任意输入流, 例如: 文件输入流;

Scanner完美的替代了BufferedReader, 而且更好的实现了InputStream的操作。

范例: 使用Scanner操作文件

```

package www.bit.java.test.intern;

import java.io.File;
import java.io.FileInputStream;
import java.util.Scanner;

public class TestPrint {
    public static void main(String[] args) throws Exception {
        Scanner scan = new Scanner(new FileInputStream(new
        File("/Users/yuisama/Desktop/test.txt"))) ;
        scan.useDelimiter("\n") ; // 自定义分隔符
        while (scan.hasNext()) {
            System.out.println(scan.next());
        }
        scan.close();
    }
}

```

**总结:** 以后除了二进制文件拷贝的处理之外, 那么只要是针对程序的信息输出都是用打印流 (PrintStream、PrintWriter), 信息输出使用Scanner。

## 9.序列化

所有的项目开发一定都有序列化的概念存在。

### 9.1 序列化基本概念

对象序列化指的是：将内存中保存的对象变为二进制数据流的形式进行传输，或者是将其保存在文本中。但是并不意味着所有类的对象都可以被序列化，严格来讲，需要被序列化的类对象往往需要传输使用，同时这个类必须实现java.io.Serializable接口。但是这个接口并没有任何的方法定义，只是一个标识而已。

范例：定义可以被序列化对象的类

```
@SuppressWarnings("serial")
class Person implements Serializable{
    private String name ;
    private int age ;
    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}
```

序列化对象时所需要保存的就是对象中的属性，所以默认情况下对象的属性将被转为二进制数据流存在。

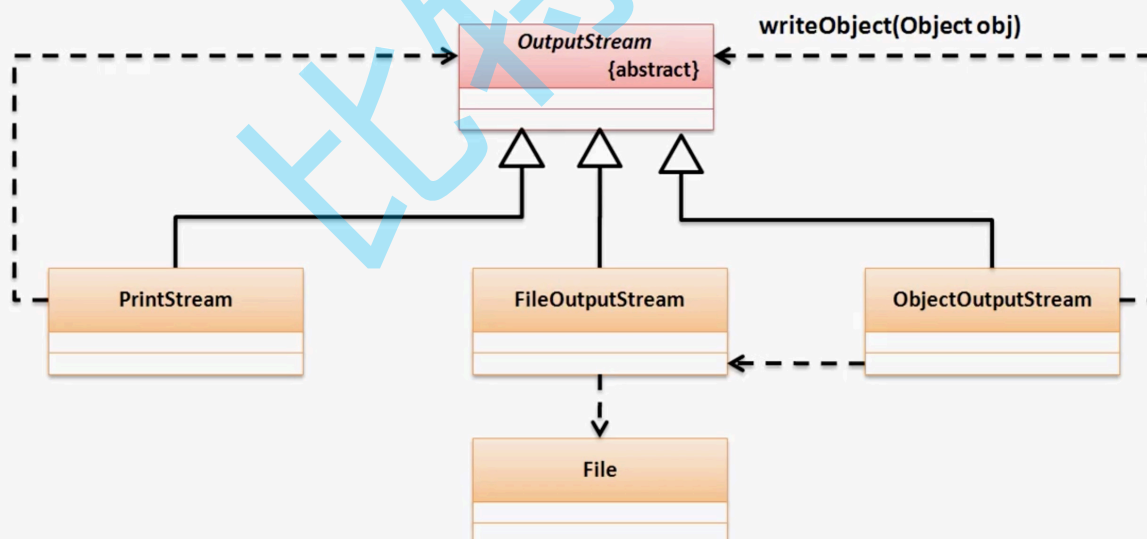
### 9.2 序列化与反序列化(了解)

如果要想实现序列化与反序列化的对象操作，在java.io包中提供有两个处理类:ObjectOutputStream、ObjectInputStream

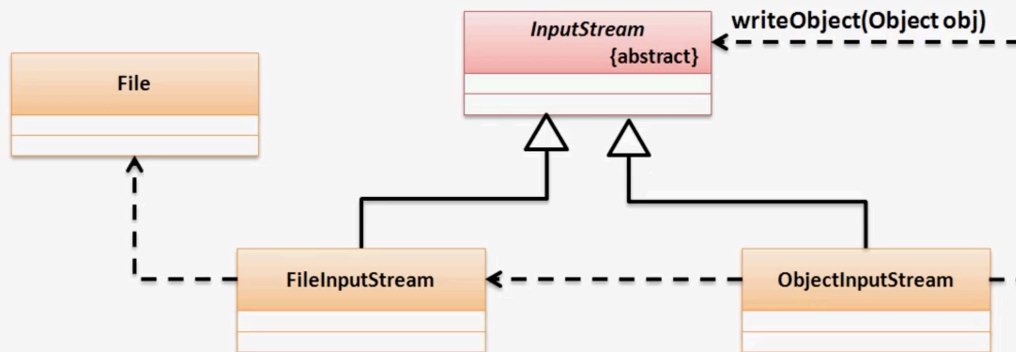
首先来观察这两个类的定义结构及其各自的构造方法

<u>ObjectOutputStream</u> ↩	<u>ObjectInputStream</u> ↩
public class <u>ObjectOutputStream</u> ↩ extends <u>OutputStream</u> implements <u>ObjectOutput</u> , <u>ObjectStreamConstants</u> ↩	public class <u>ObjectInputStream</u> ↩ extends <u>InputStream</u> implements <u>ObjectInput</u> , <u>ObjectStreamConstants</u> ↩
public <u>ObjectOutputStream</u> ( <u>OutputStream</u> out) throws <u>IOException</u> ↩	public <u>ObjectInputStream</u> ( <u>InputStream</u> in) throws <u>IOException</u> ↩
public final void <u>writeObject</u> (Object obj) throws <u>IOException</u> ↩	

## 对象序列化ObjectOutputStream



# 对象反序列化ObjectInputStream



范例：实现对象序列化

```
package www.bit.java.test.intern;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.Serializable;

@SuppressWarnings("serial")
class Person implements Serializable{
    private String name ;
    private int age ;
    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}

public class TestPrint {
    public static final File FILE = new
File("/Users/yuisama/Desktop/testSer.txt") ;
    public static void main(String[] args) throws Exception {
        ser(new Person("yuisama", 25)) ;
    }
    public static void ser(Object obj) throws FileNotFoundException,
```



```

IOException {
    ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(FILE)) ;
    oos.writeObject(obj ) ;
    oos.close() ;
}
}

```

范例：实现对象反序列化

```

public static void dser() throws Exception {
    ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(FILE)) ;
    System.out.println(ois.readObject()) ;
    ois.close() ;
}

```

在以后实际项目的开发过程之中，你们根本就不需要知道如何实现序列化和反序列操作，因为都会由各自的容器帮助你自动处理。

## 9.3 transient关键字(理解)

实际上序列化的处理在java.io包里面有两类，Serializable是使用最多的序列化接口，这种操作采用自动化模式完成，也就是说默认情况下所有的属性都会被序列化下来。

还有一个Externalizable接口是需要用户自己动手来处理序列化，一般很少使用。

但是由于Serializable默认会将对象中所有属性进行序列化保存，如果现在某些属性不希望被保存了，那么就可以使用transient关键字。

范例：使用transient

```

package www.bit.java.test.intern;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

@SuppressWarnings("serial")
class Person implements Serializable{
    private transient String name ;
    private int age ;
    public Person(String name, int age) {

```

```

        super();
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}

public class TestPrint {
    public static final File FILE = new
File("/Users/yuisama/Desktop/testSer.txt") ;
    public static void main(String[] args) throws Exception {
//        ser(new Person("yuisama", 25)) ;
        dser();
    }
    public static void ser(Object obj) throws FileNotFoundException,
IOException {
        ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(FILE)) ;
        oos.writeObject(obj) ;
        oos.close() ;
    }
    public static void dser() throws Exception {
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(FILE)) ;
        System.out.println(ois.readObject()) ;
        ois.close() ;
    }
}

```

大部分情况下使用序列化往往是在简单java类上，其他类上使用序列化的操作模式相对较少。而如果是简单java类，很少去使用transient关键字了。