




Java I/O总结

- [从new BufferedReader\(new InputStreamReader\(conn.getInputStream\(\)\)\)想到的](#) 
- [Java I/O总结——InputStream](#)
- [Java I/O总结——OutputStream](#)
- [Java I/O总结——Reader](#)
- [Java I/O总结——Writer](#) 
- [Java I/O总结——补充说明](#) 

从new BufferedReader(new InputStreamReader(conn.getInputStream()))想到的

从 new BufferedReader(new InputStreamReader(conn.getInputStream()))想到的?晚上睡在床上, 这一小段代码在我的脑海里不断浮现, 因为它 看上去有些相似 (在设计模式中的看到过类似), 但是实在想不起与那个模式相似了?

翻开设计模式书, 已经好久没有看到过本书了, 说实话对这本书中的大多数还不是很了解, 但是此刻看到能让我想到了, 说明这个模式给我留下了深刻的影 响。翻开书找了半天, 我一直以为是 Strategy, 后来看了下不是的, 从头看到尾, 终于找到了 Decorator (装饰)。把这个设计模式又仔细读了一遍, 在这里与大家分享。

设计意图

动态地给一个对象添加一些额外的职责, 就增加功能来说, Decorator 模式相比较生产子类更为灵活。

设计动机

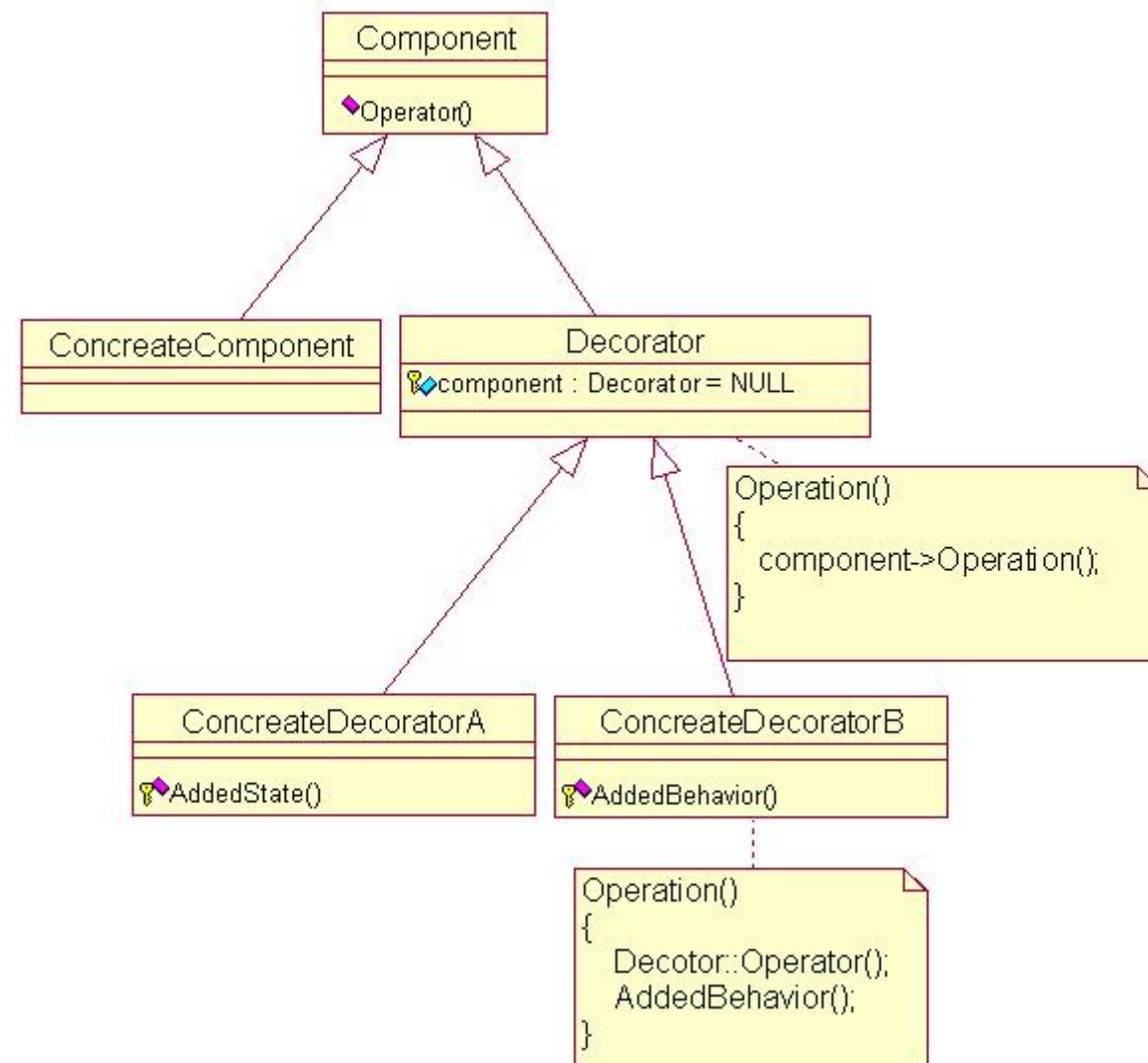
有时候我们希望给某个对象增加而不是整个类增加一些功能, 例如, 给一个图像界面工具箱允许你对人员一个用户界面的组件添加一些特性, 比如说边框, 或者窗口滚动。

使用继承机制是添加功能的一种有效途径, 从其他类继承过来的边框特性可以被多个子类的实例所实现。但是这种方法不够灵活, 因为边框的选择是静态的, 用户不能控制对组件加边框的方式和时机。

一种较为灵活的方式是将组件嵌入另外一个对象中, 由这个对象添加边框, 我们称这个嵌入的对象为装饰。

结构

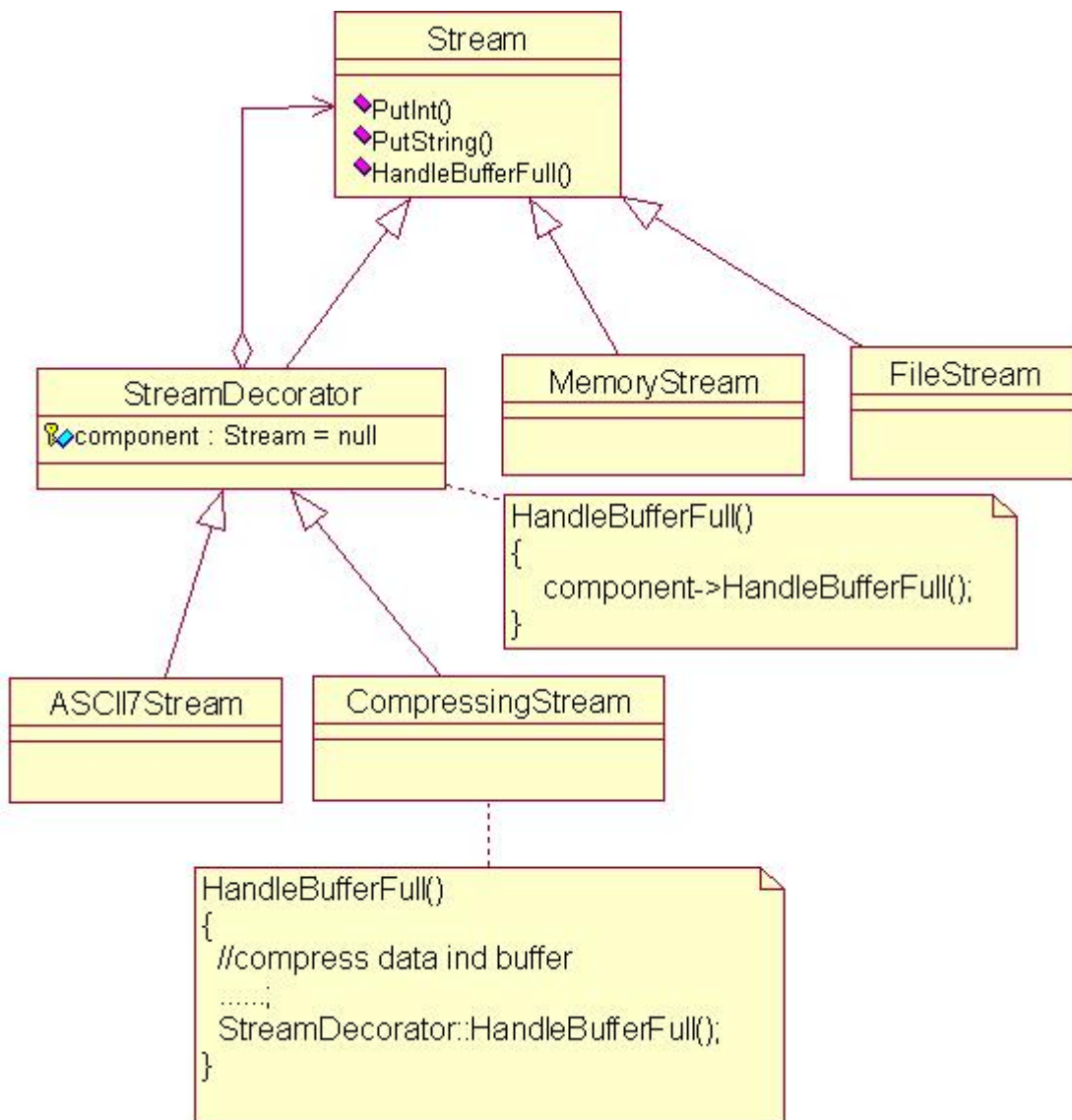
代码部分我们就不详细说明，如下图所示：



这张图当时好像是我学习设计模式中，看的最清楚的一张（难得啊，设计模式中的那么多图，很多当时看的时候都不是很明白），当然这张图不是我最在意的。

Decorator 模式的经典使用

Streams 是大多数 I/O 设备的基础抽象对象，它提供了将对象转换为字符或字符流的操作接口，使得我们可以将一个对象转换成另外一个文件或者内存中的字符串，可以在以后恢复中使用。说了这么多，估计大家也不是很明白，看了下面这张图估计你就明白了：



看了上面这张图，我记得当时顿时对 C++ 中的那么多 XXXInputStream, XXXXOutputStream 明白了很多。对上面这很张图稍 微解释：Stream 抽象类位置一个内部缓冲区并提供一些操作（PutInt, PutString）用于将数据存入流中，一段这个缓冲区满了，Stream 就会调用抽象操作 HandleBufferFull 进行实际数据传递，在 FileStream 中重定义这个操作，将缓冲区的数据传递到文 件中去。

这里的关键类是 StreamDecorator，它维持了一个指向组件流的指针并将请求转发给它，StreamDecorator 子类重定义 HandleBufferFull() 操作并且在调用 StreamDecorator 的 HandleBufferFull 操作之前执行一些额外的动作。

例如：CompressingStream 子类用于压缩数据，而 ASCII7Stream 将数据转换成 7 位 ASICC 码，现在我们创建 FileStream 类，它首先将数据压缩，然后将压缩了的二进制数据转换为 7 位 ASICC 码，我们用 CompressingStream 和 ASCII7Stream 装饰 FileStream：

```
Sream *aStream = new CompressingStream(  
    new ASCII7Stream (
```

```
        new FileStream( "aFileName" )
            )
    );

aStream->PutInt(12);
aStream->PutString( "www.moandroid.com" );
```

记得当时给我印象最深的就是在这个地方，突然间让我明白了很多。

总结说明

很早以前看到这个地方，感受颇深，今天再次看到这个地方，一方面时间已经过去了很多，另外一方面对问题的理解也比较深入了。从 `new BufferedReader(new InputStreamReader(conn.getInputStream()))` 想到的，也许不是这段代码，而是当时那种学习的心境，硬着头皮看（尽管当时很多看不懂），而现在已经很久没有那种心境了，这次写下这篇博客留恋。

随机日志

- [Android 2.0 SDK发布了](#)
- [Android画图学习总结（三）——Drawable](#)
- [\[翻译\]点基础知识——OpenGL ES Common/Common-Lite 规范（版本 1.1.12）](#)
- [Android画图学习总结（一）——类的简介](#)
- [MOTOBLUR模拟演示视频\[Video\]](#)

Java I/O总结——InputStream

在前面介绍了[Decorator（装饰）模式](#)，让我自己想起了刚开始工作时那段“痛并快乐”的学习时光。在学习Android网络方面，也发现网络方面的很多内容都与Java I/O有关，因此暂时先停下Android网络方面的学习，把Java I/O完整的学习下。我们将按照基类的顺序：[InputStream](#)、[OutputStream](#)、[Reader](#)、[Writer](#)来分别对Java I/O加以总结。

在这里强调以下 2 点：

- 如果你对设计模式中的Decorator（装饰）不是很了解，请仔细阅读：[从new BufferedReader\(new InputStreamReader\(conn.getInputStream\(\)\)\)想到的](#)，在这篇博客中，我们详细介绍了Decorator（装饰）模式。在了解了Decorator（装饰）后，我们对学习Java I/O的学习会容易很多，因为Java I/O的核心就是采用了Decorator（装饰）模式。
- Java I/O 系列的博客是我在学习 Java I/O 的基础上，对 Java I/O 的总结，关于 Java I/O 方面的基础知识，我们在这里不详细说明。如果你对 Java I/O 完全不了解，请先找些关于 Java I/O 方面

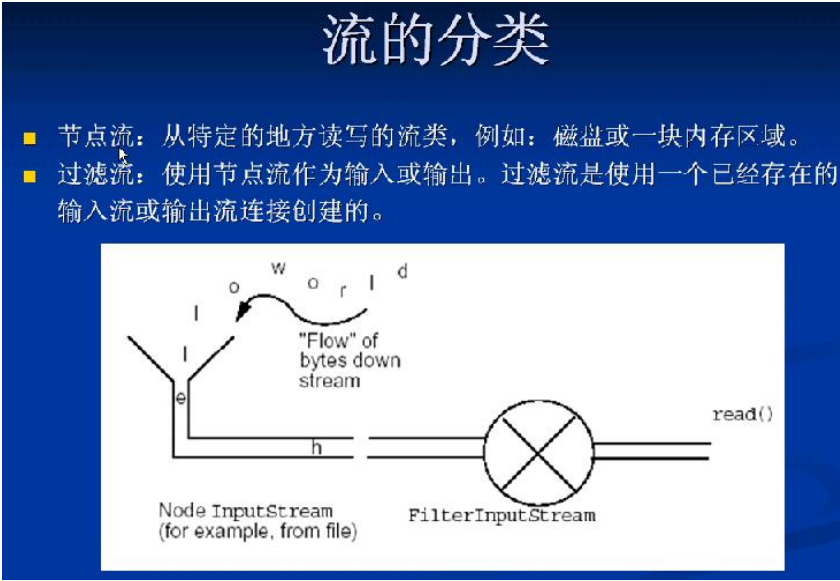
的资料简要阅读下。在简要学习的基础上，我们将 Java I/O 方面的资料整理，帮助大家 Java I/O 的知识更加清晰些。

Java I/O相关的包，如下表格所示：

包	功能说明
java.io	提供输入/输出操作。这个包下的任何数据的读写都会阻塞当前程序的运行，直到数据操作成功或者出现异常。举例说明：程序 read()方法从一个网络输入流中读取数据，由于网络速度的问题，需要等待很长的时间才能得到数据，那么 read()方法所在的程序部分就会停止在那里，直到读取数据或者出现网络传输超时异常。
java.nio	提供输入/输出操作。在 JDK 1.4 总增加了 java.io（意思是 New I/O），NIO 提供更加高效的非阻塞输入/输出操作。

在这里，我们主要介绍 java.io 包。

流的分类，如下图所示：



过滤流就是采用了 Decorator（装饰）模式，后面我们会为大家详细说明。

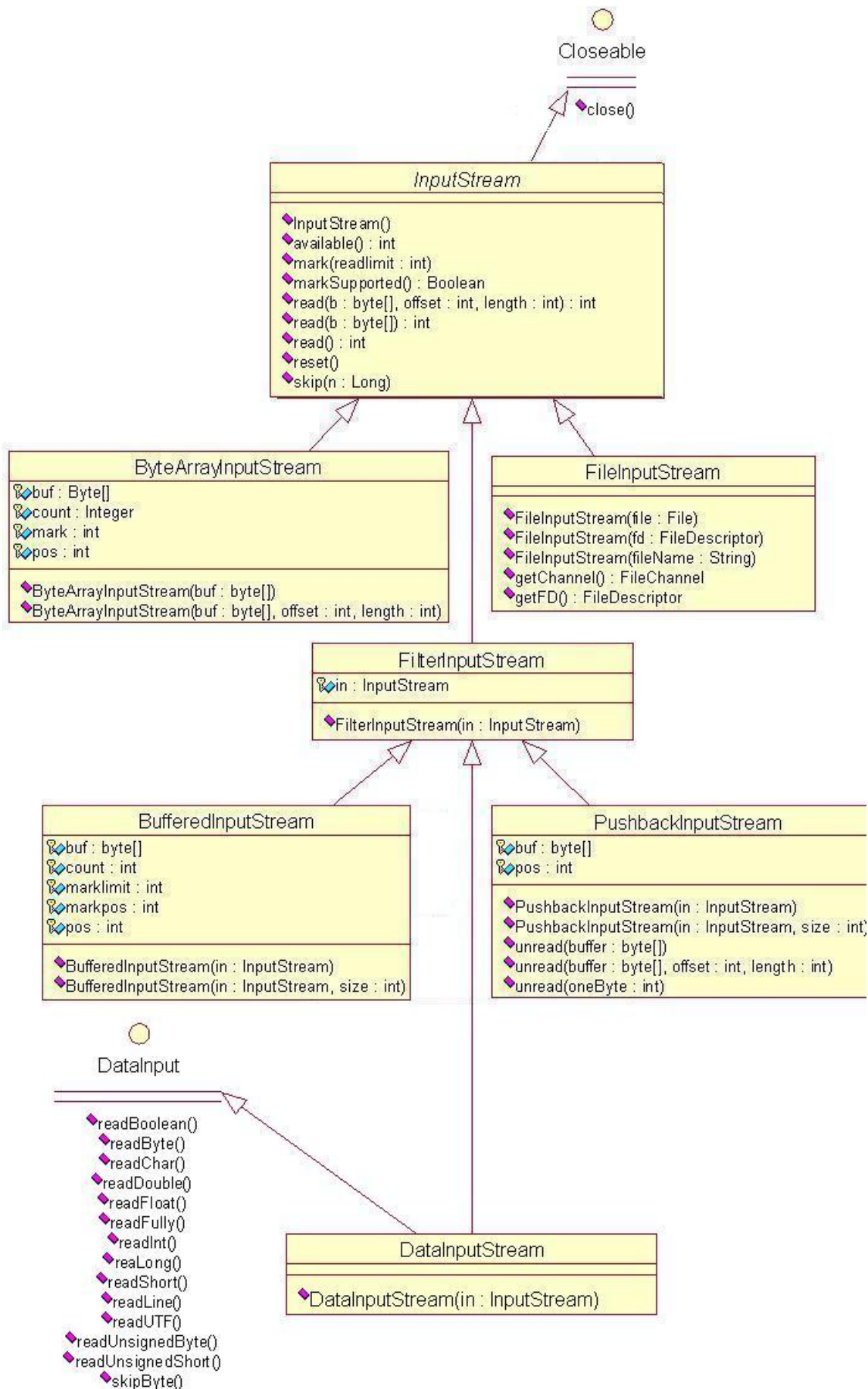
Java I/O的设计原则（Decorator（装饰）模式）

Java I/O库的设计原则

- Java的I/O库提供了一个称做链接的机制，可以将一个流与另一个流首尾相接，形成一个流管道的链接。这种机制实际上是一种被称为Decorator(装饰)设计模式的应用。
- 通过流的链接，可以动态的增加流的功能，而这种功能的增加是通过组合一些流的基本功能而动态获取的。
- 我们要获取一个I/O对象，往往需要产生多个I/O对象，这也是Java I/O库不太容易掌握的原因，但在I/O库中Decorator模式的运用，给我们提供了实现上的灵活性。

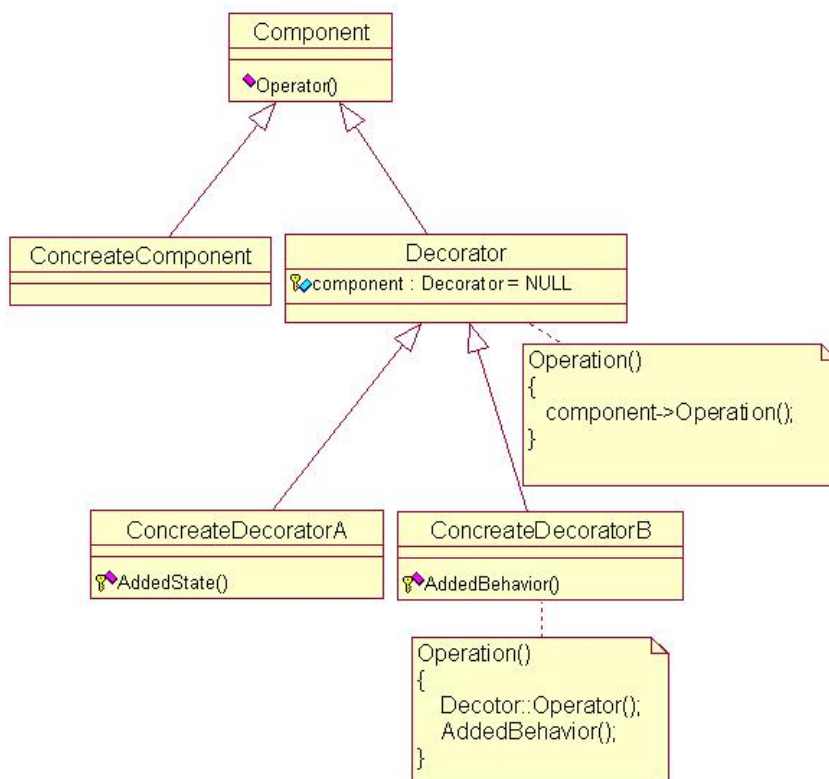
java.io.InputStream类总结

首先学习基类 java.io.InputStream（读取一系列字节的对象），以及在它基础上派生出来的子类，类结构图如下图所示（单击查看大图片）：



InputStream 是一个抽象类，是所有数据形式为字节的输入流的父类，为基于字节的数据输入定义了基本操作方法。实际上，InputStream 的子类大部分都没有增加任何其他接口函数（在上面的类结构图中就可以发现），因此在看 InputStream 子类的时候，我们主要学习其构造函数。

Java I/O 中是如何采用 Decorator（装饰）模式的呢？下面为大家详细说明，看到 FilterInputStream 类（也就前面说的过滤流，后面你会发现更多的过滤流），你是否发现了？对，就是 FilterInputStream 类，她就相当于 Decorator（装饰）模式中的 Decorator 类，而且的 BufferedInputStream、DataInputStream、PushbackInputStream 则相当于是 ConcreteDecorator，如下图所示：



那么 Java I/O 中到底是如何使用的了？Decorator（装饰）模式的主要意图是：动态地给一个对象添加一些额外的职责，这句话很抽象，我们结合 Java I/O 举个具体的例子：比如说我们读取文件，首先打开文件获取到 File，然后我们再创建一个 FileInputStream，然后读取文件。读取文件是一个很费时的操作，尤其是需要多次的读写文件。

自己的一点经历与大家分享：在一次 C++ 编程的时候，需要打开文件然后每次读取一小段数据，后来发现整个程序的效率比较低，通过测试代码发现，对数据的读取方面占用太多的时间。尽管 C++ 对文件的读取中实现了缓冲机制，但是好像这个缓冲区域比较小，然后通过 `_setbuf()` 函数增大缓冲区，发现程序的效率居然提升了很多。

那么在 Java 中采用面向对象的思想：BufferedInputStream 实现了对数据读取的缓冲机制，通过 FileInputStream 来 读取数据，BufferedInputStream 将已经读取的数据存储到缓冲区，BufferedInputStream 相当于对 FileInputStream 进行了“装饰”。

```
File file = new File("c:\\moandroid.txt",true);
InputStream is = new BufferedInputStream(new FileInputStream(file));
long length = file.length();
if(length>Integer.MAX_VALUE)
{
    System.out.println("source file is too large");
    return ;
}
byte[] bytes = new byte[(int)length];
int offset = 0,numRead = 0;
while( offset<bytes.length && (numRead = is.read(bytes,offset,bytes.length-offset))>= 0)
offset += numRead;
if(offset<bytes.length)
    throw new IOException("Could not completely read file"+file.getName());
is.close();
```

DataInputStream 类的功能则更加强大，其在 InputStream 类的基础上增加了很多读取函数的接口，举个例子如下：

```
InputStream is = null;
try
{
    File file = new File("c:\\moandroid.txt",true);
    is = new DataInputStream(new FileInputStream(file));
    int intData = is.readInt();
    boolean boolData = is.readBoolean();
}
catch(FileNotFoundException e)
{
    e.printStackTrace();
}
catch(IOException e)
{
    e.printStackTrace();
}
finally
{
    if(is!=null)
    {
        try{is.close();}
        catch(IOException e){}
    }
}
```

```
}  
}
```

这里补充说明：`DataInputStream` 读取的顺序必须和实际数据存储的顺序一致，否则会出现 `IOException`。

上面的代码可以使用如下图来表示，更加的直观：



大家再去仔细看下 Decorator（装饰）模式中的例子：一个具有边框与滚动条的文本显示窗口，是不是很相似！

总结说明

整个 java.io 的核心都是采用了 [Decorator（装饰）模式](#)，理解了 Decorator（装饰）模式，对 Java I/O 的学习是不是更加简单了，这里我只是将自己的学习的笔记整理后与大家分享。

java.io.InputStream 类结构图，建议大家打印出来后，在以后使用时候在拿出来看下，就知道如何使用了。当然这张类图结构还不完整（缺少其他一些类），打印出来后自己补充说明，把这张类图逐步完善。

相关文章

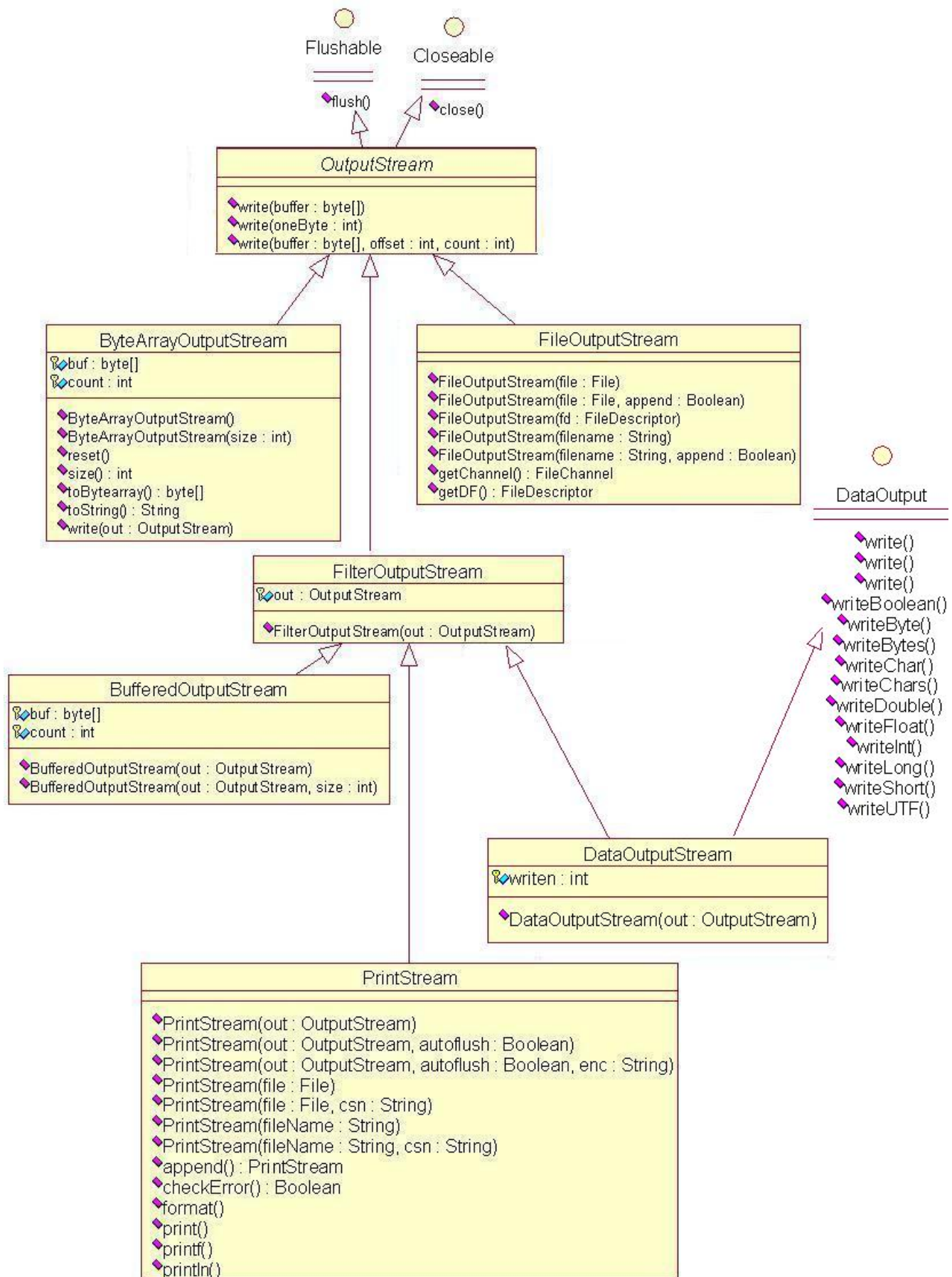
- [Java I/O 总结——补充说明](#)
- [Java I/O 总结——Writer](#)
- [Java I/O 总结——Reader](#)
- [Java I/O 总结——OutputStream](#)

Java I/O 总结——OutputStream

前面我们学习了 [InputStream](#)，`OutputStream` 采用了类似的结构。本篇幅，我们只是介绍 `OutputStream` 中特殊的部分，与 `InputStream` 相同的部分不再重复说明。

java.io. OutputStream 类总结

和 InputStream 类似，OutputStream（写入一系列字节的对象）是所有字节形式输出流的父类。类结构图如下图所示（单击查看大图片）：



上面这张图与前面的 `java.io.InputStream` 的类结构图有些类似，在 `java.io.InputStream/OutputStream` 的类结构中可以找到相互对应的类，这里额外需要说明的是 `PrintStream` 类。

这个类看上去有些陌生，很少使用它。实际上标准输出流：`System.out` 的类型就是 `java.io.PrintStream`。`PrintStream` 作为 `FilterOutputStream` 的子类，其作用也是将某个输出流再次封装，并且提供了一些新的输出特性。说到 `System.out` 估计大家就明白多了，这里就不在说明了。其他标准输出：`System.in` 的类型是 `InputStream`，其默认是由 JRE 指向系统的标准输入流，在控制台默认是键盘的输入，使用 `in.read()` 方法，将返回用户键盘输入的值；`System.err` 的类型也是 `java.io.PrintStream`。

下面举个具体的例子来说明，如何实现重定向标准输入/输出：

```
PrintStream output = new PrintStream(new FileOutputStream("c:/out.log"));
System.setOut(output);
PrintStream errOutput = new PrintStream(new FileOutputStream("c:/err.log"));
System.setErr(errOutput);
System.out.println("Output redirect test");
System.err.println("Error redirect test");
```

原来在控制台输出的文字都将被写入 `out.log` 或 `err.log` 文件中。

对其他的字节流，做个稍微的总结，如下图所示：

基本的流类

- **FileInputStream和FileOutputStream**
节点流，用于从文件中读取或往文件中写入字节流。如果在构造 `FileOutputStream` 时，文件已经存在，则覆盖这个文件。
- **BufferedInputStream和BufferedOutputStream**
过滤流，需要使用已经存在的节点流来构造，提供带缓冲的读写，提高了读写的效率。
- **DataInputStream和DataOutputStream**
过滤流，需要使用已经存在的节点流来构造，提供了读写 `Java` 中的基本数据类型的功能。
- **PipedInputStream和PipedOutputStream**
管道流，用于线程间的通信。一个线程的 `PipedInputStream` 对象从另一个线程的 `PipedOutputStream` 对象读取输入。要使管道流有用，必须同时构造管道输入流和管道输出流。

总结说明

java.io 学习起来也相当的简单，我个人认为主要原因是 JAVA 是完全面向对象的，而 C++ 由于包含了一些 C 语言的元素，在很多方面显得比较复杂，估计这也是大部分人认为 C++ 比 JAVA 复杂的主要原因吧！

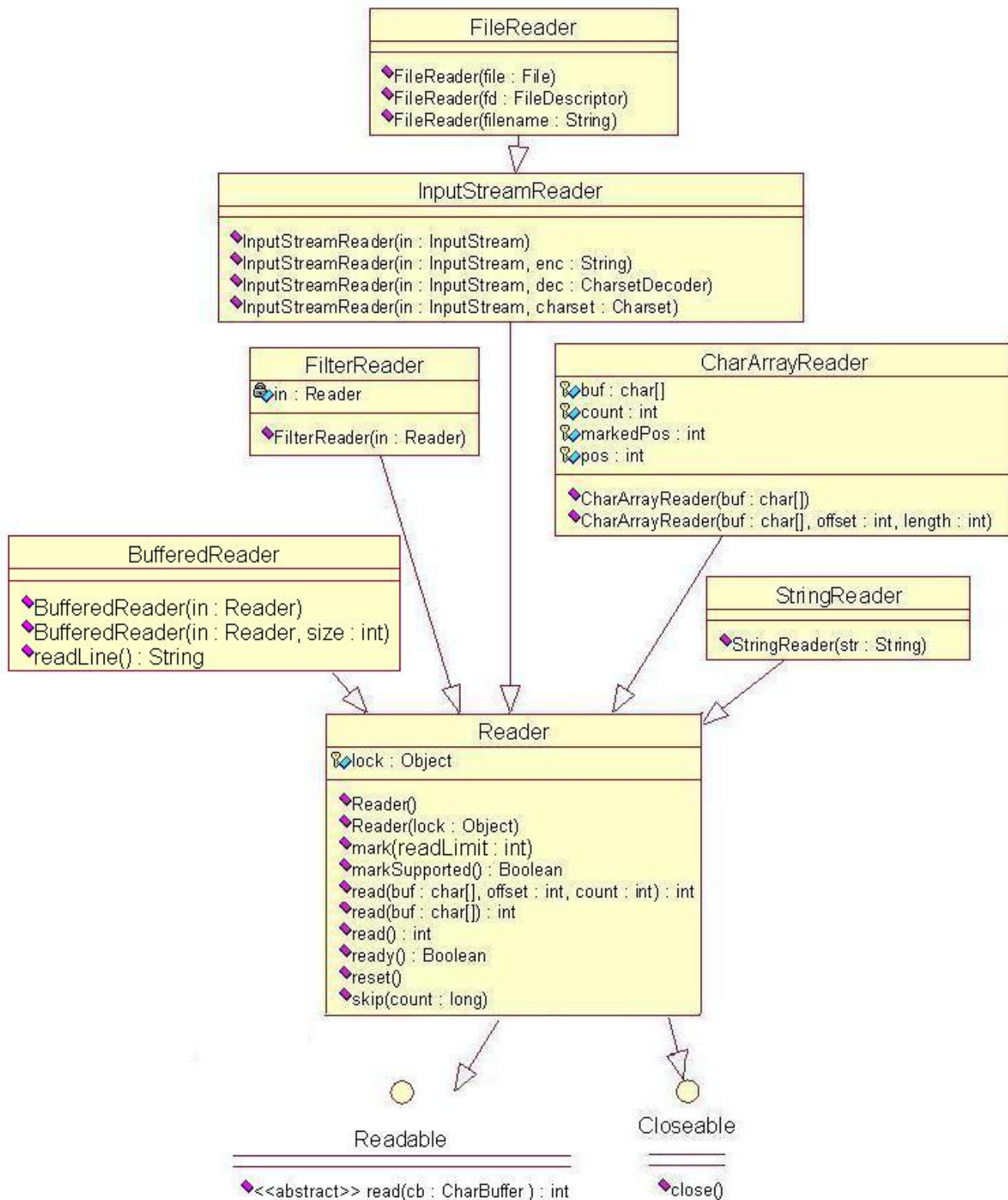
相关文章

- [Java I/O总结——补充说明](#)
- [Java I/O总结——Writer](#)
- [Java I/O总结——Reader](#)
- [Java I/O总结——InputStream](#)

Java I/O总结——Reader

[InputStream](#)和[OutputStream](#)是针对基于字节（byte）输入输出设计的，实际应用中常常需要读写的是基于字符（char ,Unicode 2 个字节）的，java.io.Reader和java.io.Writer就是所有读写字符数据流的父类。

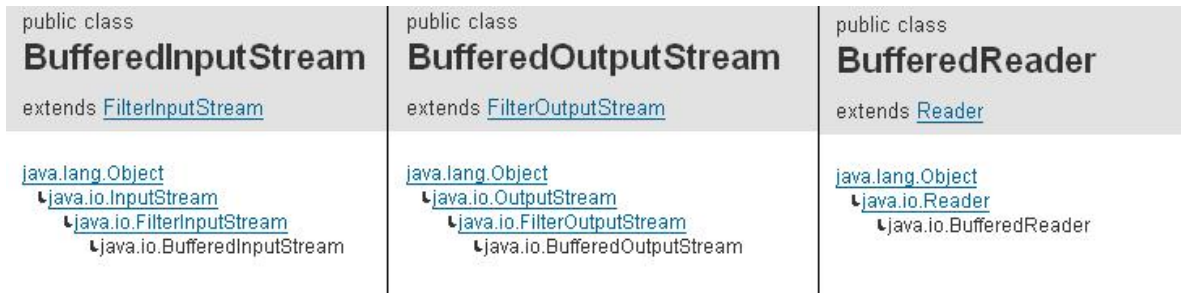
Reader 提供的方法和 InputStream 提供的几乎是一样的，不同之处在于 Reader 的操作多数是 char 类型的。



对上图（从下往上看）总结说明如下：

- Reader 类中定义了成员变量 lock，顾名思义，lock 的用途是解决实现对流操作的同步问题。Reader 的子类可以使用这个成员变量来实现流操作的同步操作。

- FilterReader 也是采用了 Decorator (装饰) 模式, 与我们在前面学习 InputStream、OutputStream 相比较, 如下图所示:



前面的 BufferedInputStream、BufferedOutputStream 都是采用了 Decorator (装饰) 模式, 而这里的 BufferedReader 是直接 from Reader 继承下来了, 而不是从 FilterReader 继承的, 不符合 Decorator (装饰) 模式据说这个 JAVA SDK 中的一个 BUG, 但是好像一直也没有修改, 估计对使用没有大的影响吧。此外 BufferedReader 还提供了一个 String readLine() 函数, 方便我们从文件中读取一行数据, 这个函数也是比较有用的。

- InputStreamReader 就像是一个桥: 把字节流 (byte) 转换为字符流 (char), 它读取字节流, 并使用指定的字符集 (charset) 将这些字节流转换为字符。常见的字符集有: GBK、ISO-8859-1、UTF-8, Java 的字符流本身采用的就是 Unicode, 关于字符集的详细 说明请看文档。给大家举个具体的例子, 如下:

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String strLine;
while(strLine=in.readLine()!=null)
{
    System.out.println(strLine);
}
in.close();
```

具体的含义就是: 将标准输入流转换为字符流, 并缓存到 BufferedReader 的缓冲区中, 并将缓冲区中的数据显示在标准输出设备上。
- StringReader、CharArrayReader 使用起来都非常简单;

总结说明

Reader 子类中较常用的还是: BufferedInputStream、InputStreamReader, BufferedInputStream 为我们提供了数据缓存机制, 对于读取大块的数据比较方便, 尤其是在读取文件 的时候; InputStreamReader 作为流的转换, 转换后的字符流是什么编码, 与虚拟机默认的字符集有关, 关于字符集部分大家还需要去深入学习。

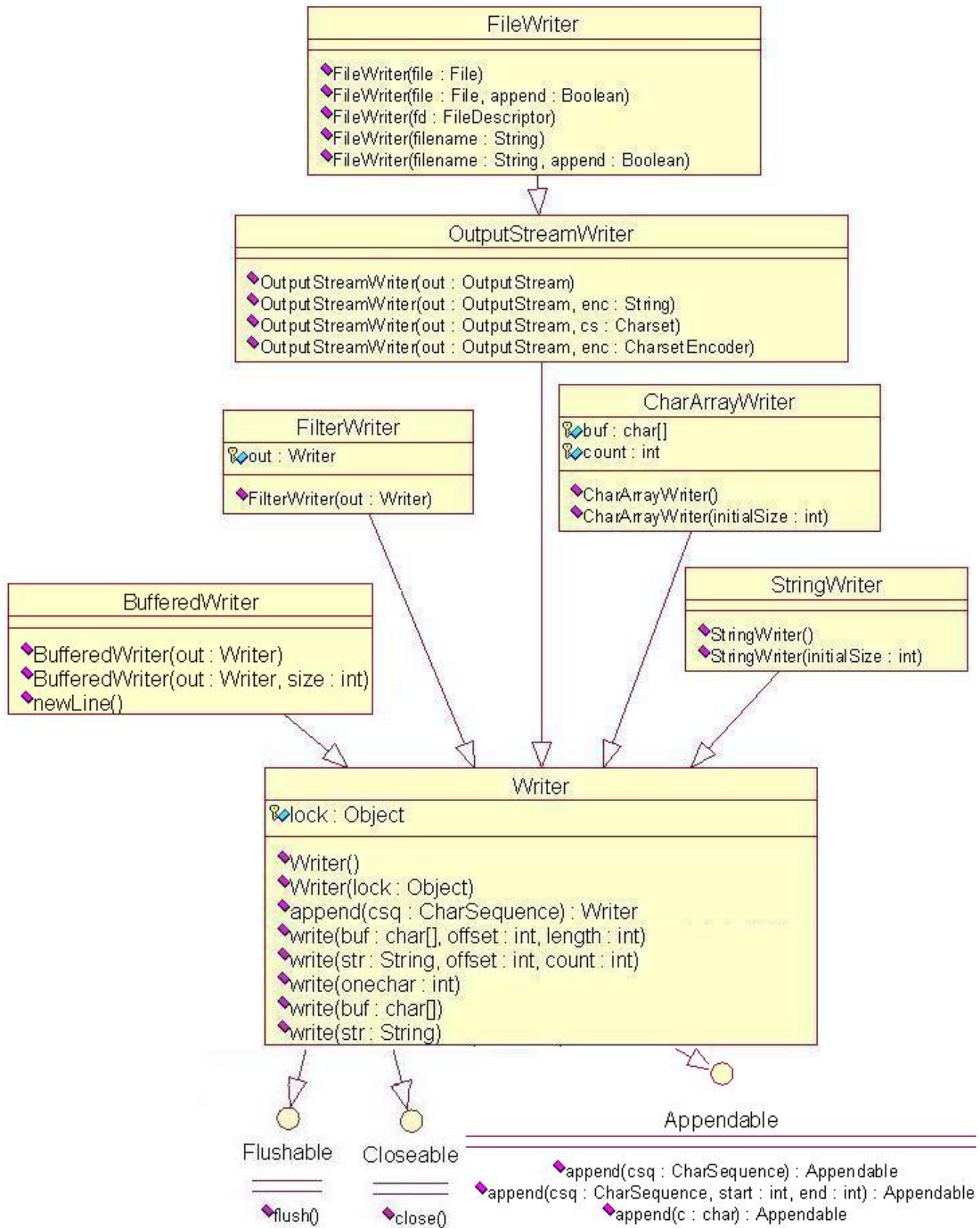
相关文章

- [Java I/O总结——补充说明](#)
- [Java I/O总结——Writer](#)
- [Java I/O总结——OutputStream](#)

- [Java I/O总结——InputStream](#)

Java I/O总结——Writer

前一篇幅我们总结了[Reader](#), Writer与Reader几乎是“心心相印”的, 此外Writer提供的方法和[OutputStream](#)提供的几乎是一样的, 不同之处在于Writer的操作多数是char类型的, 类结构如下图所示:



对比前面的 Reader 的类结构图，相信大家看得会更清楚。在这里我一直想说的是：Java I/O 在使用的时候很容易混淆，给大家看个表，如下：

流		说明
字节流	InputStream	读取（read）一系列字节（byte）的对象
	OutputStream	写入（write）一系列字节（byte）的对象
字符流	Reader	读取（read）一系列字符（char）的对象
	Writer	写入（write）一系列字符（char）的对象

- **字节与字符的主要区别是：**字节是二进制的数字，字符也是二进制数据，但是这种数据包含有区域信息（字符集），需要翻译才可以获取实际的信息。
- **输入流与输出流的主要区别是：**输入流的主要作用是从流中读取数据，输出流的主要作用是向流中写入数据；

在这里在给大家一条很好的建议：把[InputStream](#)、[OutputStream](#)、[Reader](#)、[Writer](#)的类图分别打印出来，在自己模糊的时候拿出来看下，时间长了你就清楚了。人的记忆就是需要不断的加深再加深，这样你回忆的速度会越来越快。

相关文章

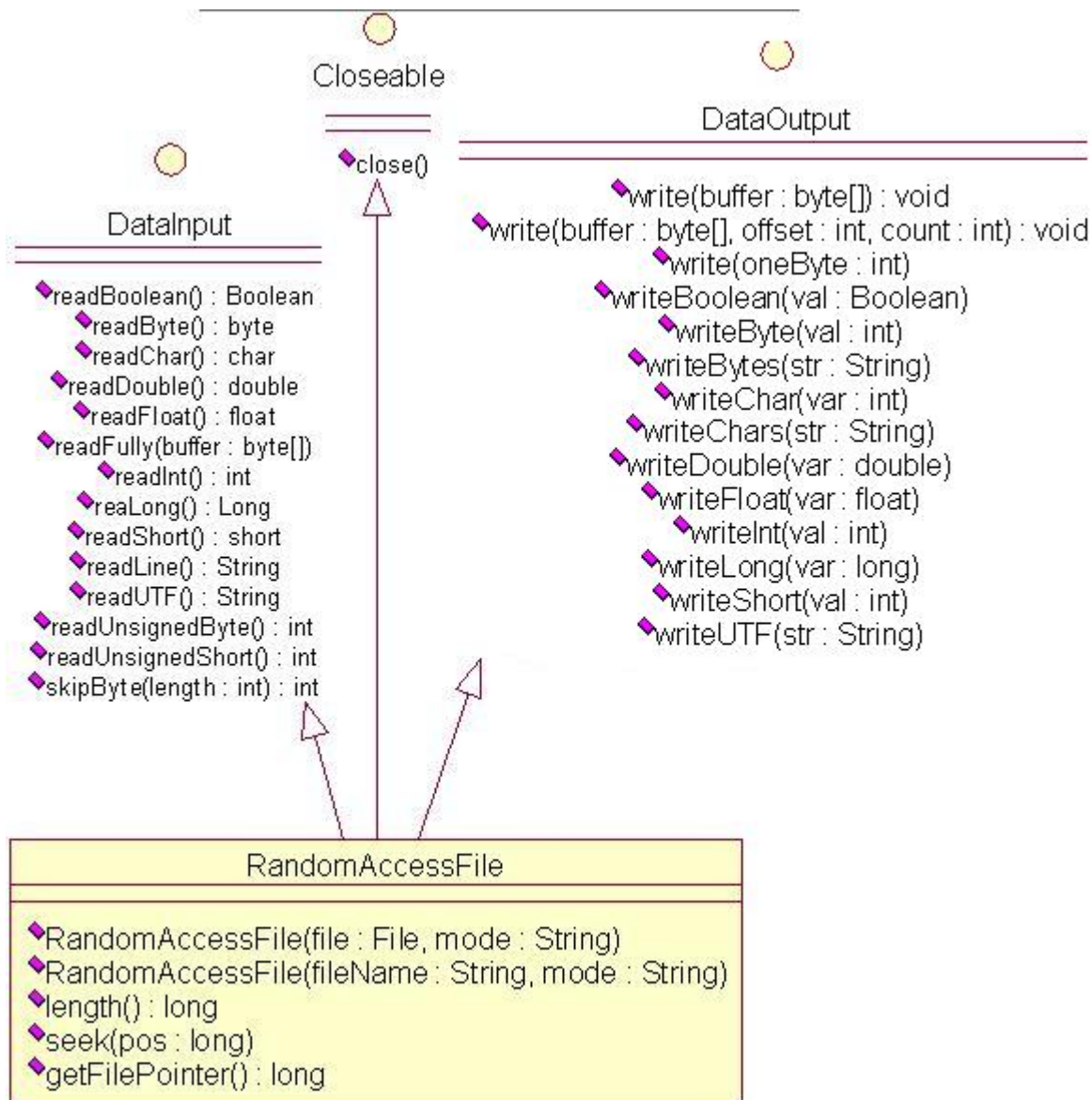
- [Java I/O总结——补充说明](#)
- [Java I/O总结——Reader](#)
- [Java I/O总结——OutputStream](#)
- [Java I/O总结——InputStream](#)

Java I/O总结——补充说明

前面总结了那么多，我在准备完成 Java I/O 总结系列文章的时候，还是发现文章中遗漏了不少内容，本篇幅把这些遗漏的内容补充说明。

RandomAccessFile

前面对文件的读写都是流的顺序数据读写，实际文件的操作却往往会从文件的某个的特定位置开始读写操作，，因而之前这些居于流思想的 I/O 类无法满足 需求。RandomAccessFile 实现了任意位置读写的功能，此外还实现了 DataInput、DataOutput 接口，支持字节数据，字符数据 和 Java 基本数据类型的读写，类结构如下图所示：



其接口，我们在这里就不详细介绍了，需要强调说的是其构造函数，需要我们去仔细阅读，如下图所示：

```
public RandomAccessFile (File file, String mode) Since: API Level 1
```

Constructs a new `RandomAccessFile` based on `file` and opens it according to the access string in `mode`.

`mode` may have one of following values:

"r"	The file is opened in <u>read-only mode</u> . An <code>IOException</code> is thrown if any of the write methods is called.
"rw"	The file is opened for <u>reading and writing</u> . If the file does not exist, it will be created.
"rws"	The file is opened for <u>reading and writing</u> . Every change of <u>the file's content or metadata</u> must be written <u>synchronously to the target device</u> .
"rwd"	The file is opened <u>for reading and writing</u> . Every change of <u>the file's content</u> must be written <u>synchronously to the target device</u> .

Parameters

- `file` the file to open.
- `mode` the file access [mode](#), either "r", "rw", "rws" or "rwd".

Throws

- [FileNotFoundException](#) if the file cannot be opened or created according to `mode`.
- [IllegalArgumentException](#) if `mode` is not "r", "rw", "rws" or "rwd".
- [SecurityException](#) if a `SecurityManager` is installed and it denies access request according to `mode`.

看了上面的介绍，下面我们列举一个具体的例子，如下：

```
import java.io.*;
class PipedStreamTest
{
    public static void main(String[] args)
    {
        PipedOutputStream pos=new PipedOutputStream();
        PipedInputStream pis=new PipedInputStream();
        try
        {
            pos.connect(pis);
            new Producer(pos).start();
            new Consumer(pis).start();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

class Producer extends Thread
{
    private PipedOutputStream pos;
    public Producer(PipedOutputStream pos)
    {
        this.pos=pos;
    }
    public void run()
    {
        try
        {
            pos.write("Hello,welcome you!".getBytes());
            pos.close();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

class Consumer extends Thread
{
    private PipedInputStream pis;
    public Consumer(PipedInputStream pis)
    {
        this.pis=pis;
    }
    public void run()
    {
        try
        {
            byte[] buf=new byte[100];
            int len=pis.read(buf);
            System.out.println(new String(buf,0,len));
            pis.close();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

对象的可串行化

实现了 `DataInputStream` 和 `OutputStream` 接口的类可以做到对 Java 基本类型数据的读写,有事我们会遇到需要将程序运行过程中的某个对象保存下来,下次程序运行时通过读入保存的数据,可以恢复这个对象。由于对象是运行期的产物,因此对对象的读写需要 Java 虚拟机的底层支持,这在 Java 中称做对象的串行化,一个可串行化的对象可以被存储成特定形式的二进制数据。将对象串行化也非常简单,只需要类实现 `java.io.Serializable` 接口即可。找个借口没有定义任何的方法,仅仅用来表示实现这个接口的类的对象是可串行化,列举一个具体的例子,如下:

```
import java.io.*;

class ObjectSerialTest
{
    public static void main(String[] args) throws Exception
    {
        Employee e1=new Employee("zhangsan",25,3000.50);
        Employee e2=new Employee("lisi",24,3200.40);
        Employee e3=new Employee("wangwu",27,3800.55);

        FileOutputStream fos=new FileOutputStream("employee.txt");
        ObjectOutputStream oos=new ObjectOutputStream(fos);
        oos.writeObject(e1);
        oos.writeObject(e2);
        oos.writeObject(e3);
        oos.close();

        FileInputStream fis=new FileInputStream("employee.txt");
        ObjectInputStream ois=new ObjectInputStream(fis);
        Employee e;
        for(int i=0;i<3;i++)
        {
            e=(Employee)ois.readObject();
            System.out.println(e.name+" "+e.age+" "+e.salary);
        }
        ois.close();
    }
}

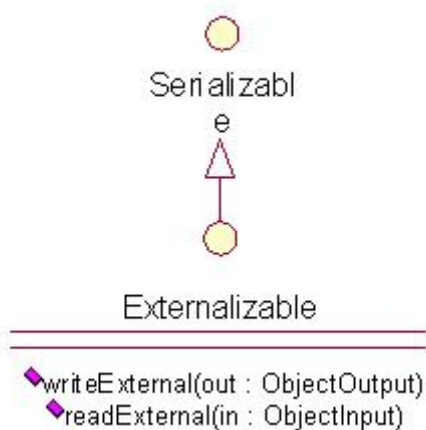
class Employee implements Serializable
{
    String name;
    int age;
    double salary;
    public Employee(String name,int age,double salary)
    {
        this.name=name;
        this.age=age;
        this.salary=salary;
    }
}
```


在这里需要强调说的是：

对象序列化

- 当一个对象被序列化时，只保存对象的非静态成员变量，不能保存任何的成员方法和静态的成员变量。
- 如果一个对象的成员变量是一个对象，那么这个对象的数据成员也会被保存。
- 如果一个可序列化的对象包含对某个不可序列化的对象的引用，那么整个序列化操作将会失败，并且会抛出一个`NotSerializableException`。我们可以将这个引用标记为`transient`，那么对象仍然可以序列化。

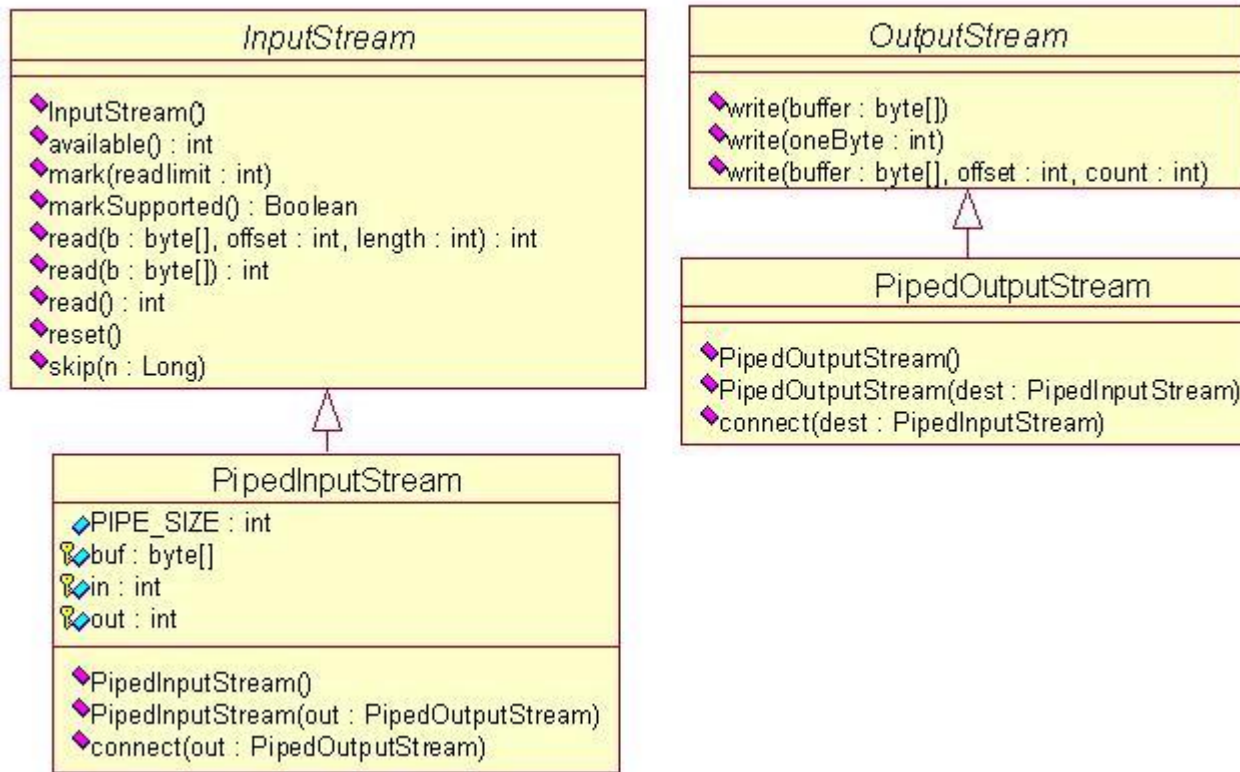
在一些特殊情况下，比如说数据存储时需要加密，这个时候需要自己实现 `Externalizable` 接口，这个接口如下图所示：



这个接口的实现非常简单，具体的实例就不列举了。

PipedInputStream和PipedOutputStream

在 Window 中可以使用 Pipe 实现数据传递,在 Android 中也有类似的方法:2 个线程之间通过 Pipe 交换数据, PipedInputStream 向管道中读取数据, PipedOutputStream 读取管道的数据。下面我们来看这 2 个类的结构图, 如下图所示:



下面我们列举一个具体的例子, 如下:

```
import java.io.*;
class PipedStreamTest
{
    public static void main(String[] args)
    {
        PipedOutputStream pos=new PipedOutputStream();
        PipedInputStream pis=new PipedInputStream();
        try
        {
            pos.connect(pis);
            new Producer(pos).start();
            new Consumer(pis).start();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

class Producer extends Thread
{
    private PipedOutputStream pos;
    public Producer(PipedOutputStream pos)
    {
        this.pos=pos;
    }
    public void run()
    {
        try
        {
            pos.write("Hello,welcome you!".getBytes());
            pos.close();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

class Consumer extends Thread
{
    private PipedInputStream pis;
    public Consumer(PipedInputStream pis)
    {
        this.pis=pis;
    }
    public void run()
    {
        try
        {
            byte[] buf=new byte[100];
            int len=pis.read(buf);
            System.out.println(new String(buf,0,len));
            pis.close();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

总结说明

Java I/O 的总结总算完成了，中间由于工作比较忙，中断了一个星期左右。在这里需要特殊说明的，Java I/O 与 Android SDK 中似乎有些区别，比如说 Java I/O 中提供了对压缩文件的读写支持：ZipOutputStream、GZIPOutputStream、ZipInputStream、GZIPInputStream，在 Android SDK 中都没有看到，不知道是遗漏了还是真的没有，在使用的过程中大家还需要仔细看下 Android SDK 中的说明。

相关文章

- [Java I/O总结——Writer](#)
- [Java I/O总结——Reader](#)
- [Java I/O总结——OutputStream](#)
- [Java I/O总结——InputStream](#)