**I/O类体系**

在JDK API中，基础的IO类都位于java.io包，而新实现的IO类则位于一系列以java.nio开头的包名中，这里首先介绍java.io包中类的体系结构。

按照前面的说明，流是有方向的，则整个流的结构按照流的方向可以划分为两类：

1、输入流：

该类流将外部数据源的数据转换为流，程序通过读取该类流中的数据，完成对于外部数据源中数据的读入。

2、输出流：

该类流完成将流中的数据转换到对应的数据源中，程序通过向该类流中写入数据，完成将数据写入到对应的外部数据源中。

而在实际实现时，由于JDK API历史的原因，在java.io包中又实现了两类流：字节流(byte stream)和字符流(char stream)。这两种流实现的是流中数据序列的单位，在字节流中，数据序列以byte为单位，也就是流中的数据按照一个byte一个byte的顺序实现成流，对于该类流操作的基本单位是一个byte，而对于字节流，数据序列以char为单位，也就是流中的数据按照一个char一个插入的顺序实现成流，对于该类流操作的基本单位是一个char。

另外字节流是从JDK1.0开始加入到API中的，而字符流则是从JDK1.1开始才加入到API中的，对于现在使用的JDK版本来说，这两类流都包含在API的内部。在实际使用时，字符流的效率要比字节流高一些。

在实际使用时，字符流中的类基本上和字节流中的类对应，所以在开始学习IO类时，可以从最基础的字节流开始学习。

在SUN设计JDK的IO类时，按照以上的分类，为每个系列的类设计了一个父类，而实现具体操作的类都作为该系列类的子类，则IO类设计时的四个体系中每个体系中对应的父类分别是：

**字节输入流InputStream**

该类是IO编程中所有字节输入流的父类，熟悉该类的使用将对使用字节输入流产生很大的帮助，下面做一下详细的介绍。

按照前面介绍的流的概念，字节输入流完成的是按照字节形式构造读取数据的输入流的结构，每个该类的对象就是一个实际的输入流，在构造时由API完成将外部数据源转换为流对象的操作，这种转换对程序员来说是透明的。在程序使用时，程序员只需要读取该流对象，就可以完成对于外部数据的读取了。

InputStream是所有字节输入流的父类，所以在InputStream类中包含的每个方法都会被所有字节输入流类继承，通过将读取以及操作数据的基本方法都声明在InputStream类内部，使每个子类根据需要覆盖对应的方法，这样的设计可以保证每个字节输入流子类在进行实际使用时，开放给程序员使用的功能方法是一致的。这样将简化IO类学习的难度，方便程序员进行实际的编程。

默认情况下，对于输入流内部数据的读取都是单向的，也就是只能从输入流从前向后读，已经读取的数据将从输入流内部删除掉。如果需要重复读取流中同一段内容，则需要使用流类中的mark方法进行标记，然后才能重复读取。这种设计在使用流类时，需要深刻进行体会。

在InputStream类中，常见的方法有：

a、available方法

public int available() throws IOException

该方法的作用是返回当前流对象中还没有被读取的字节数量。也就是获得流中数据的长度。

假设初始情况下流内部包含100个字节的数据，程序调用对应的方法读取了一个字节，则当前流中剩余的字节数量将变成99个。

另外，该方法不是在所有字节输入流内部都得到正确的实现，所以使用该方法获得流中数据的个数是不可靠的。

b、close方法

public void close() throws IOException

该方法的作用是关闭当前流对象，并释放该流对象占用的资源。

在IO操作结束以后，关闭流是进行IO操作时都需要实现的功能，这样既可以保证数据源的安全，也可以减少内存的占用。

c、markSupported方法

public boolean markSupported()

该方法的作用是判断流是否支持标记(mark)。标记类似于读书时的书签，可以很方便的回到原来读过的位置继续向下读取。

d、reset方法

public void reset() throws IOException

该方法的作用是使流读取的位置回到设定标记的位置。可以从该位置开始继续向后读取。

e、mark方法

public void mark(int readlimit)

为流中当前的位置设置标志，使得以后可以从该位置继续读取。变量readlimit指设置该标志以后可以读取的流中最大数据的个数。当设置标志以后，读取的字节数量超过该限制，则标志会失效。

f、read方法

read方法是输入流类使用时最核心的方法，能够熟练使用该方法就代表IO基本使用已经入门。所以在学习以及后期的使用中都需要深刻理解该方法的使用。

在实际读取流中的数据时，只能按照流中的数据存储顺序依次进行读取，在使用字节输入流时，读取数据的最小单位是字节(byte)。

另外，需要注意的是，read方法是阻塞方法，也就是如果流对象中无数据可以读取时，则read方法会阻止程序继续向下运行，一直到有数据可以读取为止。

read方法总计有三个，依次是：

public abstract int read() throws IOException

该方法的作用是读取当前流对象中的第一个字节。当该字节被读取出来以后，则该字节将被从流对象中删除，原来流对象中的第二个字节将变成流中的第一个字节，而使用流对象的available方法获得的数值也将减少1。如果需要读取流中的所以数据，只要使用一个循环依次读取每个数据即可。当读取到流的末尾时，该方法返回-1。该返回值的int中只有最后一个字节是流中的有效数据，所以在获得流中的数值时需要进行强制转换。返回值作成int的目的主要是处理好-1的问题。

由于该方法是抽象的，所以会在子类中被覆盖，从而实现最基础的读数据的功能。

public int read(byte[] b) throws IOException

该方法的作用是读取当前流对象中的数据，并将读取到的数据依次存储到数组b(b需要提前初始化完成)中，也就是把当前流中的第一个字节的数据存储到b[0]，第二个字节的数据存储到b[1]，依次类推。流中已经读取过的数据也会被删除，后续的数据会变成流中的第一个字节。而实际读取的字节数量则作为方法的返回值返回。

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

该方法的作用和上面的方法类似，也是将读取的数据存储到b中，只是将流中的第一个数据存储到b中下标为off的位置，最多读取len个数据，而实际读取的字节数量则作为方法的返回值返回。

g、skip方法

public long skip(long n) throws IOException

该方法的作用是跳过当前流对象中的n个字节，而实际跳过的字节数量则以返回值的方式返回。

跳过n个字节以后，如果需要读取则是从新的位置开始读取了。使用该方法可以跳过流中指定的字节数，而不用依次进行读取了。

从流中读取出数据以后，获得的是一个byte数组，还需要根据以前的数据格式，实现对于该byte数组的解析。

由于InputStream类是字节输入流的父类，所以该体系中的每个子类都包含以上的方法，这些方法是实现IO流数据读取的基础。

**字节输出流OutputStream**

该类是所有的字节输出流的父类，在实际使用时，一般使用该类的子类进行编程，但是该类内部的方法是实现字节输出流的基础。

该体系中的类完成把对应的数据写入到数据源中，在写数据时，进行的操作分两步实现：第一步，将需要输出的数据写入流对象中，数据的格式由程序员进行设定，该步骤需要编写代码实现；第二步，将流中的数据输出到数据源中，该步骤由API实现，程序员不需要了解内部实现的细节，只需要构造对应的流对象即可。

在实际写入流时，流内部会保留一个缓冲区，会将程序员写入流对象的数据首先暂存起来，然后在缓冲区满时将数据输出到数据源。当然，当流关闭时，输出流内部的数据会被强制输出。

字节输出流中数据的单位是字节，在将数据写入流时，一般情况下需要将数据转换为字节数组进行写入。

在OutputStream中，常见的方法有：

a、close方法

public void close() throws IOException

该方法的作用是关闭流，释放流占用的资源。

b、flush方法

public void flush() throws IOException

该方法的作用是将当前流对象中的缓冲数据强制输出出去。使用该方法可以实现立即输出。

c、write方法

write方法是输出流中的核心方法，该方法实现将数据写入流中。在实际写入前，需要实现对应的格式，然后依次写入到流中。写入流的顺序就是实际数据输出的顺序。

write方法总计有3个，依次是：

public abstract void write(int b) throws IOException

该方法的作用是向流的末尾写入一个字节的数据。写入的数据为参数b的最后一个字节。在实际向流中写数据时需要按照逻辑的顺序进行写入。该方法在OutputStream的子类内部进行实现。

public void write(byte[] b) throws IOException

该方法的作用是将数组b中的数据依次写入当前的流对象中。

public void write(byte[] b, int off, int len) throws IOException

该方法的作用是将数组b中从下标为off(包含)开始，后续长度为len个的数据依次写入到流对象中。

在实际写入时，还需要根据逻辑的需要设定byte数值的格式，这个根据不同的需要实现不同的格式。

**字符输入流Reader**

字符输入流体系是对字节输入流体系的升级，在子类的功能上基本和字节输入流体系中的子类一一对应，但是由于字符输入流内部设计方式的不同，使得字符输入流的执行效率要比字节输入流体系高一些，在遇到类似功能的类时，可以优先选择使用字符输入流体系中的类，从而提高程序的执行效率。

Reader体系中的类和InputStream体系中的类，在功能上是一致的，最大的区别就是Reader体系中的类读取数据的单位是字符(char)，也就是每次最少读入一个字符(两个字节)的数据，在Reader体系中的读数据的方法都以字符作为最基本的单位。

Reader类和InputStream类中的很多方法，无论声明还是功能都是一样的，但是也增加了两个方法,依次介绍如下：

a、read方法

public int read(CharBuffer target) throws IOException

该方法的作用是将流内部的数据依次读入CharBuffer对象中，实际读入的char个数作为返回值返回。

b、ready方法

public boolean ready() throws IOException

该方法的作用是返回当前流对象是否准备完成，也就是流内部是否包含可以被读取的数据。

其它和InputStream类一样的方法可以参看上面的介绍。

**字符输出流Writer**

字符输出流体系是对字节输出流体系的升级，在子类的功能实现上基本上和字节输出流保持一一对应。但由于该体系中的类设计的比较晚，所以该体系中的类执行的效率要比字节输出流中对应的类效率高一些。在遇到类似功能的类时，可以优先选择使用该体系中的类进行使用，从而提高程序的执行效率。

Writer体系中的类和OutputStream体系中的类，在功能上是一致的，最大的区别就是Writer体系中的类写入数据的单位是字符(char)，也就是每次最少写入一个字符(两个字节)的数据，在Writer体系中的写数据的方法都以字符作为最基本的操作单位。

Writer类和OutputStream类中的很多方法，无论声明还是功能都是一样的，但是还是增加了一些方法，依次介绍如下：

a、append方法

将数据写入流的末尾。总计有3个方法，依次是：

public Writer append(char c) throws IOException

该方法的作用和write(int c)的作用完全一样，既将字符c写入流的末尾。

public Writer append(CharSequence csq) throws IOException

该方法的作用是将CharSequence对象csq写入流的末尾，在写入时会调用csq的toString方法将该对象转换为字符串，然后再将该字符串写入流的末尾。

public Writer append(CharSequence csq, int start, int end)

throws IOException

该方法的作用和上面的方法类似，只是将转换后字符串从索引值为start(包含)到索引值为end(不包含)的部分写入流中。

b、write方法

除了基本的write方法以外，在Writer类中又新增了两个，依次是：

public void write(String str) throws IOException

该方法的作用是将字符串str写入流中。写入时首先将str使用getChars方法转换成对应的char数组，然后实现依次写入流的末尾。

public void write(String str, int off, int len)

throws IOException

该方法的作用是将字符串str中索引值为off(包含)开始，后续长度为len个字符写入到流的末尾。

使用这两个方法将更方便将字符串写入流的末尾。

其它和OutputStream类一样的方法可以参看上面的介绍。

**小结**

在实际使用IO类时，根据逻辑上的需要，挑选对应体系中的类进行实际的使用，从而实现程序中IO的相关功能。

熟悉了IO类的体系以后，就可以首先熟悉基本的IO类的使用，然后再按照IO类体系中相关类的使用方式逐步去了解相关的IO类的使用，从而逐步熟悉java.io包中类的使用，然后再掌握IO编程。

在实际使用时，一般都使用这4个类中对应的子类，每个子类完成相关的功能。对于这些子类，也可以根据这些类是否直接连接数据源，将这些IO类分类为：

1、实体流

指直接连接数据源的IO流类

2、装饰流

指不直接连接数据源，而是建立在其它实体流对象的基础之上。

下面IO类的使用中将分别介绍这些体系中的类。在实际使用时也应该根据流属于实体流还是装饰流进行不同的使用。

## I/O类使用

由于在IO操作中，需要使用的数据源有很多，作为一个IO技术的初学者，从读写文件开始学习IO技术是一个比较好的选择。因为文件是一种常见的数据源，而且读写文件也是程序员进行IO编程的一个基本能力。本章IO类的使用就从读写文件开始。

### 文件操作

文件(File)是最常见的数据源之一，在程序中经常需要将数据存储到文件中，例如图片文件、声音文件等数据文件，也经常需要根据需要从指定的文件中进行数据的读取。当然，在实际使用时，文件都包含一个的格式，这个格式需要程序员根据需要进行设计，读取已有的文件时也需要熟悉对应的文件格式，才能把数据从文件中正确的读取出来。

文件的存储介质有很多，例如硬盘、光盘和U盘等，由于IO类设计时，从数据源转换为流对象的操作由API实现了，所以存储介质的不同对于程序员来说是透明的，和实际编写代码无关。

#### 文件的概念

文件是计算机中一种基本的数据存储形式，在实际存储数据时，如果对于数据的读写速度要求不是很高，存储的数据量不是很大时，使用文件作为一种持久数据存储的方式是比较好的选择。

存储在文件内部的数据和内存中的数据不同，存储在文件中的数据是一种“持久存储”，也就是当程序退出或计算机关机以后，数据还是存在的，而内存内部的数据在程序退出或计算机关机以后，数据就丢失了。

在不同的存储介质中，文件中的数据都是以一定的顺序依次存储起来，在实际读取时由硬件以及操作系统完成对于数据的控制，保证程序读取到的数据和存储的顺序保持一致。

每个文件以一个文件路径和文件名称进行表示，在需要访问该文件的时，只需要知道该文件的路径以及文件的全名即可。在不同的操作系统环境下，文件路径的表示形式是不一样的，例如在Windows操作系统中一般的表示形式为C:/windows/system，而Unix上的表示形式为/user/my。所以如果需要让Java程序能够在不同的操作系统下运行，书写文件路径时还需要比较注意。

##### 绝对路径和相对路径

绝对路径是指书写文件的完整路径，例如d:/java/Hello.java，该路径中包含文件的完整路径d:/java以及文件的全名Hello.java。使用该路径可以唯一的找到一个文件，不会产生歧义。但是使用绝对路径在表示文件时，受到的限制很大，且不能在不同的操作系统下运行，因为不同操作系统下绝对路径的表达形式存在不同。

相对路径是指书写文件的部分路径，例如/test/Hello.java，该路径中只包含文件的部分路径/test和文件的全名Hello.java，部分路径是指当前路径下的子路径，例如当前程序在d:/abc下运行，则该文件的完整路径就是d:/abc/test。使用这种形式，可以更加通用的代表文件的位置，使得文件路径产生一定的灵活性。

在Eclipse项目中运行程序时，当前路径是项目的根目录，例如工作空间存储在d:/javaproject，当前项目名称是Test，则当前路径是：d:/javaproject/Test。在控制台下面运行程序时，当前路径是class文件所在的目录，如果class文件包含包名，则以该class文件最顶层的包名作为当前路径。

另外在Java语言的代码内部书写文件路径时，需要注意大小写，大小写需要保持一致，路径中的文件夹名称区分大小写。由于’/’是Java语言中的特殊字符，所以在代码内部书写文件路径时，例如代表“c:/test/java/Hello.java”时，需要书写成“c://test//java//Hello.java”或“c:/test/java/Hello.java”，这些都需要在代码中注意。

##### 文件名称

文件名称一般采用“文件名.后缀名”的形式进行命名，其中“文件名”用来表示文件的作用，而使用后缀名来表示文件的类型，这是当前操作系统中常见的一种形式，例如“readme.txt”文件，其中readme代表该文件时说明文件，而txt后缀名代表文件时文本文件类型，在操作系统中，还会自动将特定格式的后缀名和对应的程序关联，在双击该文件时使用特定的程序打开。

其实在文件名称只是一个标示，和实际存储的文件内容没有必然的联系，只是使用这种方式方便文件的使用。在程序中需要存储数据时，如果自己设计了特定的文件格式，则可以自定义文件的后缀名，来标示自己的文件类型。

和文件路径一样，在Java代码内部书写文件名称时也区分大小写，文件名称的大小写必须和操作系统中的大小写保持一致。

另外，在书写文件名称时不要忘记书写文件的后缀名。

##### File类

为了很方便的代表文件的概念，以及存储一些对于文件的基本操作，在java.io包中设计了一个专门的类——File类。

在File类中包含了大部分和文件操作的功能方法，该类的对象可以代表一个具体的文件或文件夹，所以以前曾有人建议将该类的类名修改成FilePath，因为该类也可以代表一个文件夹，更准确的说是可以代表一个文件路径。

下面介绍一下File类的基本使用。

1、File对象代表文件路径

File类的对象可以代表一个具体的文件路径，在实际代表时，可以使用绝对路径也可以使用相对路径。

下面是创建的文件对象示例。

public File(String pathname)

该示例中使用一个文件路径表示一个File类的对象，例如：

File f1 = new File(“d://test//1.txt”);

File f2 = new File(“1.txt”);

File f3 = new File(“e://abc”);

这里的f1和f2对象分别代表一个文件，f1是绝对路径，而f2是相对路径，f3则代表一个文件夹，文件夹也是文件路径的一种。

public File(String parent, String child)

也可以使用父路径和子路径结合，实现代表文件路径，例如：

File f4 = new File(“d://test//”,”1.txt”);

这样代表的文件路径是：d:/test/1.txt。

2、File类常用方法

File类中包含了很多获得文件或文件夹属性的方法，使用起来比较方便，下面将常见的方法介绍如下：

a、createNewFile方法

public boolean createNewFile() throws IOException

该方法的作用是创建指定的文件。该方法只能用于创建文件，不能用于创建文件夹，且文件路径中包含的文件夹必须存在。

b、delect方法

public boolean delete()

该方法的作用是删除当前文件或文件夹。如果删除的是文件夹，则该文件夹必须为空。如果需要删除一个非空的文件夹，则需要首先删除该文件夹内部的每个文件和文件夹，然后在可以删除，这个需要书写一定的逻辑代码实现。

c、exists方法

public boolean exists()

该方法的作用是判断当前文件或文件夹是否存在。

d、getAbsolutePath方法

public String getAbsolutePath()

该方法的作用是获得当前文件或文件夹的绝对路径。例如c:/test/1.t则返回c:/test/1.t。

e、getName方法

public String getName()

该方法的作用是获得当前文件或文件夹的名称。例如c:/test/1.t，则返回1.t。

f、getParent方法

public String getParent()

该方法的作用是获得当前路径中的父路径。例如c:/test/1.t则返回c:/test。

g、isDirectory方法

public boolean isDirectory()

该方法的作用是判断当前File对象是否是目录。

h、isFile方法

public boolean isFile()

该方法的作用是判断当前File对象是否是文件。

i、length方法

public long length()

该方法的作用是返回文件存储时占用的字节数。该数值获得的是文件的实际大小，而不是文件在存储时占用的空间数。

j、list方法

public String[] list()

该方法的作用是返回当前文件夹下所有的文件名和文件夹名称。说明，该名称不是绝对路径。

k、listFiles方法

public File[] listFiles()

该方法的作用是返回当前文件夹下所有的文件对象。

l、mkdir方法

public boolean mkdir()

该方法的作用是创建当前文件文件夹，而不创建该路径中的其它文件夹。假设d盘下只有一个test文件夹，则创建d:/test/abc文件夹则成功，如果创建d:/a/b文件夹则创建失败，因为该路径中d:/a文件夹不存在。如果创建成功则返回true，否则返回false。

m、mkdirs方法

public boolean mkdirs()

该方法的作用是创建文件夹，如果当前路径中包含的父目录不存在时，也会自动根据需要创建。

n、renameTo方法

public boolean renameTo(File dest)

该方法的作用是修改文件名。在修改文件名时不能改变文件路径，如果该路径下已有该文件，则会修改失败。

o、setReadOnly方法

public boolean setReadOnly()

该方法的作用是设置当前文件或文件夹为只读。

3、File类基本示例

以上各方法实现的测试代码如下：

import java.io.File;

/\*\*

\* File类使用示例

\*/

public class FileDemo {

public static void main(String[] args) {

//创建File对象

File f1 = new File("d://test");

File f2 = new File("1.txt");

File f3 = new File("e://file.txt");

File f4 = new File("d://","1.txt");

//创建文件

try{

boolean b = f3.createNewFile();

}catch(Exception e){

e.printStackTrace();

}

//判断文件是否存在

System.out.println(f4.exists());

//获得文件的绝对路径

System.out.println(f3.getAbsolutePath());

//获得文件名

System.out.println(f3.getName());

//获得父路径

System.out.println(f3.getParent());

//判断是否是目录

System.out.println(f1.isDirectory());

//判断是否是文件

System.out.println(f3.isFile());

//获得文件长度

System.out.println(f3.length());

//获得当前文件夹下所有文件和文件夹名称

String[] s = f1.list();

for(int i = 0;i < s.length;i++){

System.out.println(s[i]);

}

//获得文件对象

File[] f5 = f1.listFiles();

for(int i = 0;i < f5.length;i++){

System.out.println(f5[i]);

}

//创建文件夹

File f6 = new File("e://test//abc");

boolean b1 = f6.mkdir();

System.out.println(b1);

b1 = f6.mkdirs();

System.out.println(b1);

//修改文件名

File f7 = new File("e://a.txt");

boolean b2 = f3.renameTo(f7);

System.out.println(b2);

//设置文件为只读

f7.setReadOnly();

}

}

4、File类综合示例

下面以两个示例演示File类的综合使用。第一个示例是显示某个文件夹下的所有文件和文件夹，原理是输出当前名称，然后判断当前File对象是文件还是文件夹，如果则获得该文件夹下的所有子文件和子文件夹，并递归调用该方法实现。第二个示例是删除某个文件夹下的所有文件和文件夹，原理是判断是否是文件，如果是文件则直接删除，如果是文件夹，则获得该文件夹下所有的子文件和子文件夹，然后递归调用该方法处理所有子文件和子文件夹，然后将空文件夹删除。则测试时谨慎使用第二个方法，以免删除自己有用的数据文件。示例代码如下：

import java.io.File;

/\*\*

\* 文件综合使用示例

\*/

public class AdvanceFileDemo {

public static void main(String[] args) {

File f = new File("e://Book");

printAllFile(f);

File f1 = new File("e://test");

deleteAll(f1);

}

/\*\*

\* 打印f路径下所有的文件和文件夹

\* @param f 文件对象

\*/

public static void printAllFile(File f){

//打印当前文件名

System.out.println(f.getName());

//是否是文件夹

if(f.isDirectory()){

//获得该文件夹下所有子文件和子文件夹

File[] f1 = f.listFiles();

//循环处理每个对象

int len = f1.length;

for(int i = 0;i < len;i++){

//递归调用，处理每个文件对象

printAllFile(f1[i]);

}

}

}

/\*\*

\* 删除对象f下的所有文件和文件夹

\* @param f 文件路径

\*/

public static void deleteAll(File f){

//文件

if(f.isFile()){

f.delete();

}else{ //文件夹

//获得当前文件夹下的所有子文件和子文件夹

File f1[] = f.listFiles();

//循环处理每个对象

int len = f1.length;

for(int i = 0;i < len;i++){

//递归调用，处理每个文件对象

deleteAll(f1[i]);

}

//删除当前文件夹

f.delete();

}

}

}

关于File类的使用就介绍这么多，其它的方法和使用时需要注意的问题还需要多进行练习和实际使用。

#### 读取文件

虽然前面介绍了流的概念，但是这个概念对于初学者来说，还是比较抽象的，下面以实际的读取文件为例子，介绍流的概念，以及输入流的基本使用。

按照前面介绍的知识，将文件中的数据读入程序，是将程序外部的数据传入程序中，应该使用输入流——InputStream或Reader。而由于读取的是特定的数据源——文件，则可以使用输入对应的子类FileInputStream或FileReader实现。

在实际书写代码时，需要首先熟悉读取文件在程序中实现的过程。在Java语言的IO编程中，读取文件是分两个步骤：1、将文件中的数据转换为流，2、读取流内部的数据。其中第一个步骤由系统完成，只需要创建对应的流对象即可，对象创建完成以后步骤1就完成了，第二个步骤使用输入流对象中的read方法即可实现了。

使用输入流进行编程时，代码一般分为3个部分：1、创建流对象，2、读取流对象内部的数据，3、关闭流对象。下面以读取文件的代码示例：

import java.io.\*;

/\*\*

\* 使用FileInputStream读取文件

\*/

public class ReadFile1 {

public static void main(String[] args) {

//声明流对象

FileInputStream fis = null;

try{

//创建流对象

fis = new FileInputStream("e://a.txt");

//读取数据，并将读取到的数据存储到数组中

byte[] data = new byte[1024]; //数据存储的数组

int i = 0; //当前下标

//读取流中的第一个字节数据

int n = fis.read();

//依次读取后续的数据

while(n != -1){ //未到达流的末尾

//将有效数据存储到数组中

data[i] = (byte)n;

//下标增加

i++;

//读取下一个字节的数据

n = fis.read();

}

//解析数据

String s = new String(data,0,i);

//输出字符串

System.out.println(s);

}catch(Exception e){

e.printStackTrace();

}finally{

try{

//关闭流，释放资源

fis.close();

}catch(Exception e){}

}

}

}

在该示例代码中，首先创建一个FileInputStream类型的对象fis：

fis = new FileInputStream("e://a.txt");

这样建立了一个连接到数据源e:/a.txt的流，并将该数据源中的数据转换为流对象fis，以后程序读取数据源中的数据，只需要从流对象fis中读取即可。

读取流fis中的数据，需要使用read方法，该方法是从InputStream类中继承过来的方法，该方法的作用是每次读取流中的一个字节，如果需要读取流中的所有数据，需要使用循环读取，当到达流的末尾时，read方法的返回值是-1。

在该示例中，首先读取流中的第一个字节：

int n = fis.read();

并将读取的值赋值给int值n，如果流fis为空，则n的值是-1，否则n中的最后一个字节包含的时流fis中的第一个字节，该字节被读取以后，将被从流fis中删除。

然后循环读取流中的其它数据，如果读取到的数据不是-1，则将已经读取到的数据n强制转换为byte，即取n中的有效数据——最后一个字节，并存储到数组data中，然后调用流对象fis中的read方法继续读取流中的下一个字节的数据。一直这样循环下去，直到读取到的数据是-1，也就是读取到流的末尾则循环结束。

这里的数组长度是1024，所以要求流中的数据长度不能超过1024，所以该示例代码在这里具有一定的局限性。如果流的数据个数比较多，则可以将1024扩大到合适的个数即可。

经过上面的循环以后，就可以将流中的数据依次存储到data数组中，存储到data数组中有效数据的个数是i个，即循环次数。

其实截至到这里，IO操作中的读取数据已经完成，然后再按照数据源中的数据格式，这里是文件的格式，解析读取出的byte数组即可。

该示例代码中的解析，只是将从流对象中读取到的有效的数据，也就是data数组中的前n个数据，转换为字符串，然后进行输出。

在该示例代码中，只是在catch语句中输出异常的信息，便于代码的调试，在实际的程序中，需要根据情况进行一定的逻辑处理，例如给出提示信息等。

最后在finally语句块中，关闭流对象fis，释放流对象占用的资源，关闭数据源，实现流操作的结束工作。

上面详细介绍了读取文件的过程，其实在实际读取流数据时，还可以使用其它的read方法，下面的示例代码是使用另外一个read方法实现读取的代码：

import java.io.FileInputStream;

/\*\*

\* 使用FileInputStream读取文件

\*/

public class ReadFile2 {

public static void main(String[] args) {

//声明流对象

FileInputStream fis = null;

try{

//创建流对象

fis = new FileInputStream("e://a.txt");

//读取数据，并将读取到的数据存储到数组中

byte[] data = new byte[1024]; //数据存储的数组

int i = fis.read(data);

//解析数据

String s = new String(data,0,i);

//输出字符串

System.out.println(s);

}catch(Exception e){

e.printStackTrace();

}finally{

try{

//关闭流，释放资源

fis.close();

}catch(Exception e){}

}

}

}

该示例代码中，只使用一行代码：

int i = fis.read(data);

就实现了将流对象fis中的数据读取到字节数组data中。该行代码的作用是将fis流中的数据读取出来，并依次存储到数组data中，返回值为实际读取的有效数据的个数。

使用该中方式在进行读取时，可以简化读取的代码。

当然，在读取文件时，也可以使用Reader类的子类FileReader进行实现，在编写代码时，只需要将上面示例代码中的byte数组替换成char数组即可。

使用FileReader读取文件时，是按照char为单位进行读取的，所以更适合于文本文件的读取，而对于二进制文件或自定义格式的文件来说，还是使用FileInputStream进行读取，方便对于读取到的数据进行解析和操作。

读取其它数据源的操作和读取文件类似，最大的区别在于建立流对象时选择的类不同，而流对象一旦建立，则基本的读取方法是一样，如果只使用最基本的read方法进行读取，则使用基本上是一致的。这也是IO类设计的初衷，使得对于流对象的操作保持一致，简化IO类使用的难度。

#### 写文件

如前所述，将程序内部的数据输出到程序外部的数据源，应该使用IO类体系中的输出流。在实际的编程中，将程序中的数据，例如用户设定或程序运行时生成的内容，存储到外部的文件中，应该使用输出流进行编程。

基本的输出流包含OutputStream和Writer两个，区别是OutputStream体系中的类(也就是OutputStream的子类)是按照字节写入的，而Writer体系中的类(也就是Writer的子类)是按照字符写入的。

使用输出流进行编程的步骤是：

1、建立输出流

建立对应的输出流对象，也就是完成由流对象到外部数据源之间的转换。

2、向流中写入数据

将需要输出的数据，调用对应的write方法写入到流对象中。

3、关闭输出流

在写入完毕以后，调用流对象的close方法关闭输出流，释放资源。

在使用输出流向外部输出数据时，程序员只需要将数据写入流对象即可，底层的API实现将流对象中的内容写入外部数据源，这个写入的过程对于程序员来说是透明的，不需要专门书写代码实现。

在向文件中输出数据，也就是写文件时，使用对应的文件输出流，包括FileOutputStream和FileWriter两个类，下面以FileOutputStream为例子说明输出流的使用。示例代码如下：

import java.io.\*;

/\*\*

\* 使用FileOutputStream写文件示例

\*/

public class WriteFile1 {

public static void main(String[] args) {

String s = "Java语言";

int n = 100;

//声明流对象

FileOutputStream fos = null;

try{

//创建流对象

fos = new FileOutputStream("e://out.txt");

//转换为byte数组

byte[] b1 = s.getBytes();

//换行符

byte[] b2 = "/r/n".getBytes();

byte[] b3 = String.valueOf(n).getBytes();

//依次写入文件

fos.write(b1);

fos.write(b2);

fos.write(b3);

} catch (Exception e) {

e.printStackTrace();

}finally{

try{

fos.close();

}catch(Exception e){}

}

}

}

该示例代码写入的文件使用记事本打开以后，内容为：

Java语言

100

在该示例代码中，演示了将一个字符串和一个int类型的值依次写入到同一个文件中。在写入文件时，首先创建了一个文件输出流对象fos：

fos = new FileOutputStream("e://out.txt");

该对象创建以后，就实现了从流到外部数据源e:/out.txt的连接。说明：当外部文件不存在时，系统会自动创建该文件，但是如果文件路径中包含未创建的目录时将出现异常。这里书写的文件路径可以是绝对路径也可以是相对路径。

在实际写入文件时，有两种写入文件的方式：覆盖和追加。其中“覆盖”是指清除原文件的内容，写入新的内容，默认采用该种形式写文件，“追加”是指在已有文件的末尾写入内容，保留原来的文件内容，例如写日志文件时，一般采用追加。在实际使用时可以根据需要采用适合的形式，可以使用：

public FileOutputStream(String name, boolean append) throws FileNotFoundException

只需要使用该构造方法在构造FileOutputStream对象时，将第二个参数append的值设置为true即可。

流对象创建完成以后，就可以使用OutputStream中提供的wirte方法向流中依次写入数据了。最基本的写入方法只支持byte数组格式的数据，所以如果需要将内容写入文件，则需要把对应的内容首先转换为byte数组。

这里以如下格式写入数据：首先写入字符串s，使用String类的getBytes方法将该字符串转换为byte数组，然后写入字符串“/r/n”，转换方式同上，该字符串的作用是实现文本文件的换行显示，最后写入int数据n，首先将n转换为字符串，再转换为byte数组。这种写入数据的顺序以及转换为byte数组的方式就是流的数据格式，也就是该文件的格式。因为这里写的都是文本文件，所以写入的内容以明文的形式显示出来，也可以根据自己需要存储的数据设定特定的文件格式。

其实，所有的数据文件，包括图片文件、声音文件等等，都是以一定的数据格式存储数据的，在保存该文件时，将需要保存的数据按照该文件的数据格式依次写入即可，而在打开该文件时，将读取到的数据按照该文件的格式解析成对应的逻辑即可。

最后，在数据写入到流内部以后，如果需要立即将写入流内部的数据强制输出到外部的数据源，则可以使用流对象的flush方法实现。如果不需要强制输出，则只需要在写入结束以后，关闭流对象即可。在关闭流对象时，系统首先将流中未输出到数据源中的数据强制输出，然后再释放该流对象占用的内存空间。

使用FileWriter写入文件时，步骤和创建流对象的操作都和该示例代码一致，只是在转换数据时，需要将写入的数据转换为char数组，对于字符串来说，可以使用String中的toCharArray方法实现转换，然后按照文件格式写入数据即可。

对于其它类型的字节输出流/字符输出流来说，只是在逻辑上连接不同的数据源，在创建对象的代码上会存在一定的不同，但是一旦流对象创建完成以后，基本的写入方法都是write方法，也需要首先将需要写入的数据按照一定的格式转换为对应的byte数组/char数组，然后依次写入即可。

所以IO类的这种设计形式，只需要熟悉该体系中的某一个类的使用以后，就可以触类旁通的学会其它相同类型的类的使用，从而简化程序员的学习，使得使用时保持统一。

**读取控制台输入**

前面介绍了使用IO类实现文件读写的示例，其实在很多地方还需要使用到IO类，这里再以读取控制台输入为例子来介绍IO类的使用。

控制台(Console)指无图形界面的程序，运行时显示或输入数据的位置，前面的介绍中可以使用System.out.println将需要输出的内容显示到控制台，本部分将介绍如何接受用户在控制台中的输入。

使用控制台输入是用户在程序运行时和程序进行交互的一种基础手段，这种手段是Windows操作系统出现以前，操作系统位于DOS时代时，用户和程序交互的主要手段。当然，现在这种交互的方式已经被图形界面(GUI)程序取代了。

在读取控制台操作中，操作系统在用户在控制台输入内容，并按回车键提交以后，将用户提交的内容传递给Java运行时系统，Java运行时系统将用户输入的信息构造成一个输入流对象——System.in，在程序员读取控制台输入时，只需要从该流中读取数据即可。至于构造流System.in的过程对于程序员来说是透明的。

查阅JDK API可以发现，System类中的静态属性in是InputStream类型的对象，可以按照输入流的读取方法读取即可。

下面的示例代码实现了输入“回显”的功能，即将用户输入的内容重新显示到控制台，示例代码如下：

/\*\*

\* 读取控制台输入，并将输入的内容显示到控制台

\*/

public class ReadConsole1 {

public static void main(String[] args) {

try{

//提示信息

System.out.println("请输入：");

//数组缓冲

byte[] b = new byte[1024];

//读取数据

int n = System.in.read(b);

//转换为字符串

String s = new String(b,0,n);

//回显内容

System.out.println("输入内容为：" + s);

}catch(Exception e){}

}

}

在该示例代码中，从System.in中读取出用户的输入，然后将用户输入的内容转换为字符串s，然后输出该字符串的内容即可。

下面实现一个简单的逻辑，功能为：回显用户在控制台输入的内容，当用户输入quit时程序运行结束。实现的代码如下：

/\*\*

\* 读取控制台输入

\* 循环回显内容，当输入quit时退出程序

\*/

public class ReadConsole2 {

public static void main(String[] args) {

//数组缓冲

byte[] b = new byte[1024];

//有效数据个数

int n = 0;

try{

while(true){

//提示信息

System.out.println("请输入：");

//读取数据

n = System.in.read(b);

//转换为字符串

String s = new String(b,0,n - 2);

//判断是否是quit

if(s.equalsIgnoreCase("quit")){

break; //结束循环

}

//回显内容

System.out.println("输入内容为：" + s);

}

}catch(Exception e){}

}

}

在该示例代码中，加入了一个while循环，使得用户的输入可以进行多次，在用户输入时，送入输入流的内容除了用户输入的内容以外，还包含”/r/n”这两个字符，所以在将输入的内容和quit比较时，去掉读出的最后2个字符，将剩余的内容转换为字符串。

最后是一个简单的《掷骰子》的控制台小游戏，在该游戏中，玩家初始拥有1000的金钱，每次输入押大还是押小，以及下注金额，随机3个骰子的点数，如果3个骰子的总点数小于等于9，则开小，否则开大，然后判断玩家是否押对，如果未押对则扣除下注金额，如果押对则奖励和玩家下注金额相同的金钱。该程序的示例代码如下：

/\*\*

\* 掷骰子游戏实现

\*/

public class DiceGame {

public static void main(String[] args) {

int money = 1000; //初始金钱数量

int diceNum = 0; // 掷出的骰子数值和

int type = 0; // 玩家押的大小

int cMoney = 0; // 当前下注金额

boolean success; // 胜负

// 游戏过程

while (true) {

// 输入大小

System.out.println("请押大小(1代表大，2代表小):");

type = readKeyboard();

// 校验

if (!checkType(type)) {

System.out.println("输入非法，请重新输入！");

continue;

}

// 输入下注金额

while(true){

System.out.println("你当前的金钱数量是"

+ money + "请下注:");

cMoney = readKeyboard();

// 校验

if (!checkCMoney(money,cMoney)) {

System.out.println("输入非法，请重新输入！");

continue;

}else{

break;

}

}

// 掷骰子

diceNum = doDice();

// 判断胜负

success = isSuccess(type,diceNum);

// 金钱变化

money = changeMoney(money,success,cMoney);

// 游戏结束

if(isEnd(money)){

System.out.println("你输了，bye！");

break;

}

}

}

/\*\*

\* 读取用户输入

\* @return 玩家输入的整数，如果格式非法则返回0

\*/

public static int readKeyboard() {

try {

// 缓冲区数组

byte[] b = new byte[1024];

// 读取用户输入到数组b中，

// 读取的字节数量为n

int n = System.in.read(b);

// 转换为整数

String s = new String(b, 0, n - 2);

int num = Integer.parseInt(s);

return num;

} catch (Exception e) {}

return 0;

}

/\*\*

\* 押的类型校验

\* @param type 类型

\* @return true代表符合要求，false代表不符合

\*/

public static boolean checkType(int type) {

if (type == 1 || type == 2) {

return true;

} else {

return false;

}

}

/\*\*

\* 校验下注金额是否合法

\* @param money 玩家金钱数

\* @param cMoney 下注金额

\* @return true代表符合要求，false代表不符合要求

\*/

public static boolean checkCMoney(int money, int cMoney) {

if (cMoney <= 0) {

return false;

} else if (cMoney <= money) {

return true;

} else {

return false;

}

}

/\*\*

\* 掷骰子

\* @return 骰子的数值之和

\*/

public static int doDice() {

int n = (int) (Math.random() \* 6) + 1;

int n1 = (int) (Math.random() \* 6) + 1;

int n2 = (int) (Math.random() \* 6) + 1;

// 输出随机结果

System.out.println("庄家开：" + n + " " + n1 + " " + n2);

return n + n1 + n2;

}

/\*\*

\* 胜负判断

\* @param type 用户输入类型

\* @param diceNum 骰子点数

\* @return true代表赢，false代表输

\*/

public static boolean isSuccess(int type, int diceNum) {

// 计算庄家类型

int bankerType = 0;

if (diceNum <= 9) {

bankerType = 2;

System.out.println("庄家开小!");

} else {

bankerType = 1;

System.out.println("庄家开大!");

}

if (bankerType == type) { // 赢

return true;

} else { // 输

return false;

}

}

/\*\*

\* 金钱变化

\* @param money 用户钱数

\* @param success 胜负

\* @param cMoney 下注金额

\* @return 变化以后的金钱

\*/

public static int changeMoney(int money, boolean success, int cMoney) {

if (success) {

money += cMoney;

} else {

money -= cMoney;

}

System.out.println("剩余金额：" + money);

return money;

}

/\*\*

\* 判断游戏是否结束

\* @param money 玩家金钱

\* @return true代表结束

\*/

public static boolean isEnd(int money) {

return money <= 0;

}

}

### 注意问题

上面介绍了IO类的基本使用，熟悉了实体流和装饰流的基本使用，但是在IO类实际使用时，还是会遇到一系列的问题，下面介绍一些可能会经常遇到的问题。

#### 类的选择

对于初次接触IO技术的初学者来说，IO类体系博大精深，类的数量比较庞大，在实际使用时经常会无所适从，不知道该使用那些类进行编程，下面介绍一下关于IO类选择的一些技巧。

选择类的第一步是选择合适的实体流。

选择实体流时第一步是按照连接的数据源种类进行选择，例如读写文件应该使用文件流，如FileInputStream/FileOutputStream、FileReader/FileWriter，读写字节数组应该使用字节数组流等，如ByteArrayInputStream/ByteArrayOutputStream。

选择实体流时第二步是选择合适方向的流。例如进行读操作时应该使用输入流，进行写操作时应该使用输出流。

选择实体流时第三步是选择字节流或字符流。除了读写二进制文件，或字节流中没有对应的流时，一般都优先选择字符流。

经过以上步骤以后，就可以选择到合适的实体流了。下面说一下装饰流的选择问题。

在选择IO类时，实体流是必需的，装饰流是可选的。另外在选择流时实体流只能选择一个，而装饰流可以选择多个。

选择装饰流时第一步是选择符合要求功能的流。例如需要缓冲流的话选择BufferedReader/BufferedWriter等，有些时候也可能只是为了使用某个装饰流内部提供的方法。

选择装饰流时第二步是选择合适方向的流，这个和实体流选择中的第二步一致。

当选择了多个装饰流以后，可以使用流之间的多层嵌套实现要求的功能，流的嵌套之间没有顺序。

#### 非依次读取流数据

由于IO类设计的特点，在实际读取时，只能依次读取流中的数据，而且在通常情况下，已经读取过的数据无法再进行读取。如果需要重复读取流中某段数据时，一般的做法是将从流中读取的数据使用数组存储起来，然后根据需要读取数组中的内容即可，但是有些时候，还是有一些特殊的情况的，IO类对于这些都进行了支持。

1、间断性的读取流中的数据

对于某些特殊格式的文件，例如字体文件等，在实际读取数据时不需要顺序进行读取，而只需要根据内容的位置进行读取。这样可以使用流中的skip方法实现。例如：

int n = fis.skip(100);

该行代码的作用是，以流fis当前位置为基础，当前位置可以是流中的任何位置，向后跳过100个单位(字节流单位为字节，字符流单位是字符)，如果再使用read方法继续读取，就是读取跳跃以后新位置的内容，也就相当于跳过了100个单位的内容。

而实际在使用时，实际真正跳过的单位数量作为skip方法的返回值返回。

2、重复读取流中某段数据

当必须重复读取流中同一段数据时，如果对应的流支持mark(标记)的话，则可以重复读取同一段数据。

下面以重复读取控制台输入流System.in为例子，来介绍mark的使用，示例代码如下：

import java.io.\*;

/\*\*

\* mark使用示例

\*/

public class MarkUseDemo {

public static void main(String[] args) {

byte[] b = new byte[1024];

try{

//读取数据

int data = System.in.read();

//输出第一个字节的数据

System.out.println("第一个字节：" + data);

//判断该流是否支持mark

if(System.in.markSupported()){

//记忆当前位置，可以从当前位置

//向后最多读取100个字节

System.in.mark(100);

//读取数据

int n = System.in.read(b);

//输出读取到的内容

System.out.print("第一次读取到的内容：");

for(int i = 0;i < n;i++){

System.out.print(b[i] + " ");

}

System.out.println();

//回到标记位置

System.in.reset();

//重复读取标记位置以后的内容

n = System.in.read(b);

//输出读取到的内容

System.out.print("第二次读取到的内容：");

for(int i = 0;i < n;i++){

System.out.print(b[i] + " ");

}

System.out.println();

}

}catch(Exception e){

e.printStackTrace();

}

}

}

在该示例中，首先调用System.in流中的read方法，读取流中的第一个字节，并把读取到的数据赋值给data，然后将读取到的第一个字节的数据输出出来。

然后调用System.in流中的markSupported判断该流是否支持mark功能，如果支持的话则markSupported方法将返回true。

如果流System.in支持mark，则标记当前位置，并允许从当前位置开始最多读取后续100个字节的数据，其实IO类内部的只读取这些数据，而不真正从流中将这些数据删除。

后续继续读取流中的数据，如果读取的数据超过100个字节，则mark标记失效，并把读取到的有效数据输出到控制台。

如果需要从标记位置重复读取已经读取过的数据，则只需要调用流对象中的reset方法重置流的位置，使流可以回到mark的位置，如果继续读取的话，则从该位置开始可以向后读取。这样就可以从mark的位置开始，再次读取后续的数据了。

例如在控制台输入123456789，则该程序的执行结果是：

第一个字节：49

第一次读取到的内容：50 51 52 53 54 55 56 57 13 10

第二次读取到的内容：50 51 52 53 54 55 56 57 13 10

其中输入的第一个字节是1，读取时该字符的编码是49，而后续的内容就是流的结构，其中流末尾的13和10是在输入时，添加的回车和换行字符。

#### 中文问题

由于JDK设计时，对于国际化支持比较好，所以JDK在实际实现时支持很多的字符集，这样在进行特定字符集的处理时就需要特别小心了。

其实在进行中文处理时，只需要注意一个原则就可以了，这个原则就是将中文字符转换为byte数组时使用的字符集，需要和把byte数组转换为中文字符串时的字符集保持一致，这样就不会出现中文问题了。

当然，如果不想手动实现字符串和byte数组的转换，可以使用DataInputStream和DataOutputStream中的readUTF和writeUTF实现读写字符串。

## 总结

关于IO类的使用，还需要在实际开发过程中多进行使用，从而更深入的体会IO类设计的初衷，并掌握IO类的使用。

另外，IO类是Java中进行网络编程的基础，所以熟悉IO类的使用也是学习网络编程必须的一个基础。