Day02

JAVA IO

文件流

文件输入流

java.io.FileInputStream使用文件输入流向从文件中读取数据

构造器

```
FileInputStream(String path)
基于给定的路径对应的文件创建文件输入流

FileInputStream(File file)
基于给定的File对象所表示的文件创建文件输入流
```

例

```
package io;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

/**
 * 文件输入流
 * 用于从文件中读取字节的流
 */
public class FISDemo {
    public static void main(String[] args) throws IOException {
        /*
            FileInputStream(String path)
            FileInputStream(File file)
         */
        FileInputStream fis = new FileInputStream("fos.dat");
        /*
            int read()
            读取1个字节，以int形式返回该字节内容。int值只有"低八位"有数据，高24位
            全部补0.
            有一个特殊情况:如果返回的int值为整数-1，则表示EOF。
            EOF:end of file 文件末尾

            fos.dat文件数据
            00000001 00000010

            第一次调用:
            int d = fis.read();

            00000001 00000010
            ^^^^^^^^
```

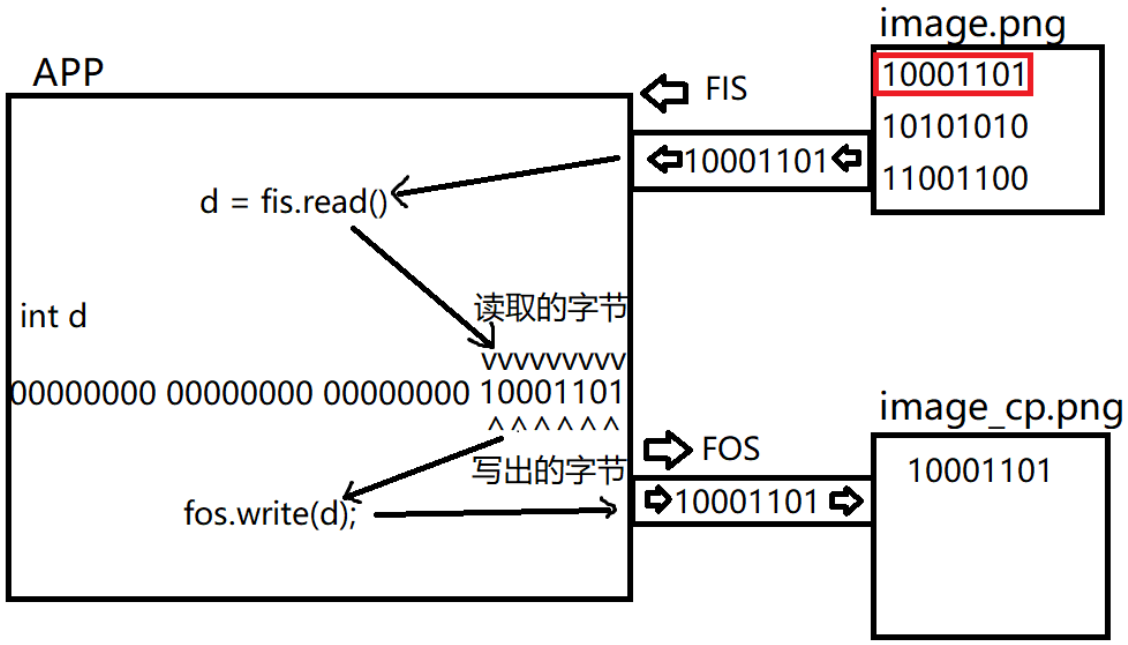
```
            读取的字节

            返回值d的二进制样子:
            00000000 00000000 00000000 00000001
            |-----自动补充的24个0------| 读取的字节


            第二次调用:
            d = fis.read();

            00000001 00000010
                     ^^^^^^^^
                     读取的字节

            返回值d的二进制样子:
            00000000 00000000 00000000 00000010
            |-----自动补充的24个0------| 读取的字节


            第三次调用:
            d = fis.read();

            00000001 00000010
                             ^^^^^^^^
                             文件末尾了

            返回值d的二进制样子:
            11111111 11111111 11111111 11111111
            |----------32位2进制都是1-----------|
         */
        int d = fis.read();
        System.out.println(d);//1
        d = fis.read();
        System.out.println(d);//2
        d = fis.read();//文件只有2个字节，因此第三次读取已经是文件末尾EOF
        System.out.println(d);//-1

        fis.close();
    }
}
```

文件复制

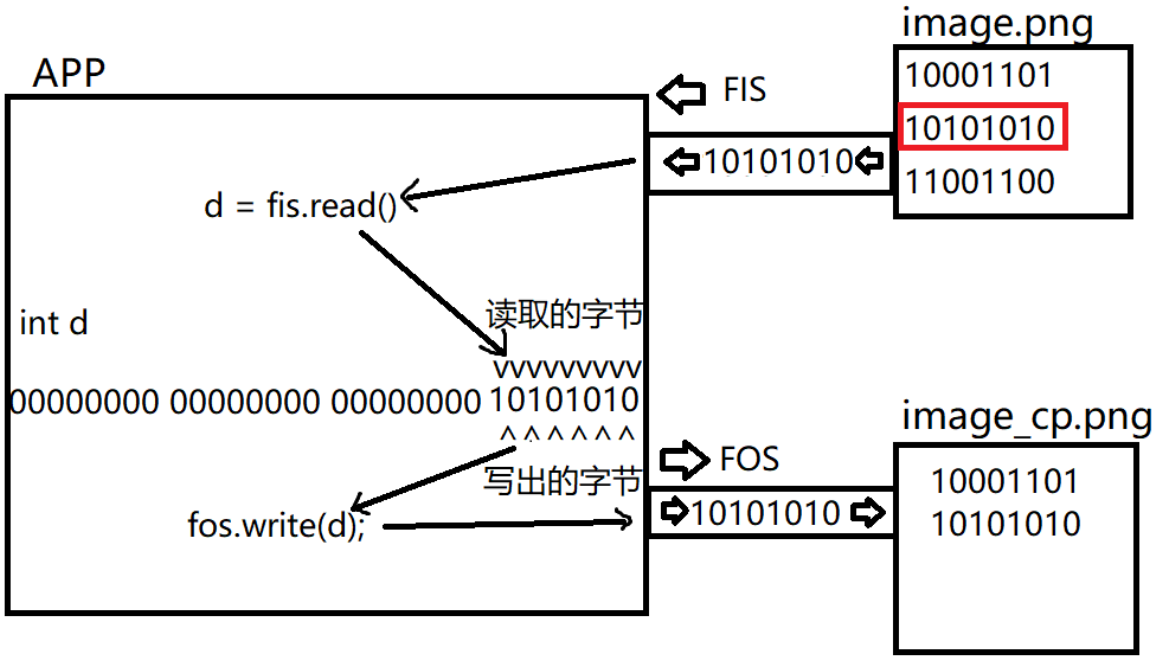
复制文件的原理就是使用文件输入流从原文件中陆续读取出每一个字节，然后再使用文件输出流将字节陆续写入到另一个文件中完成的。

示例

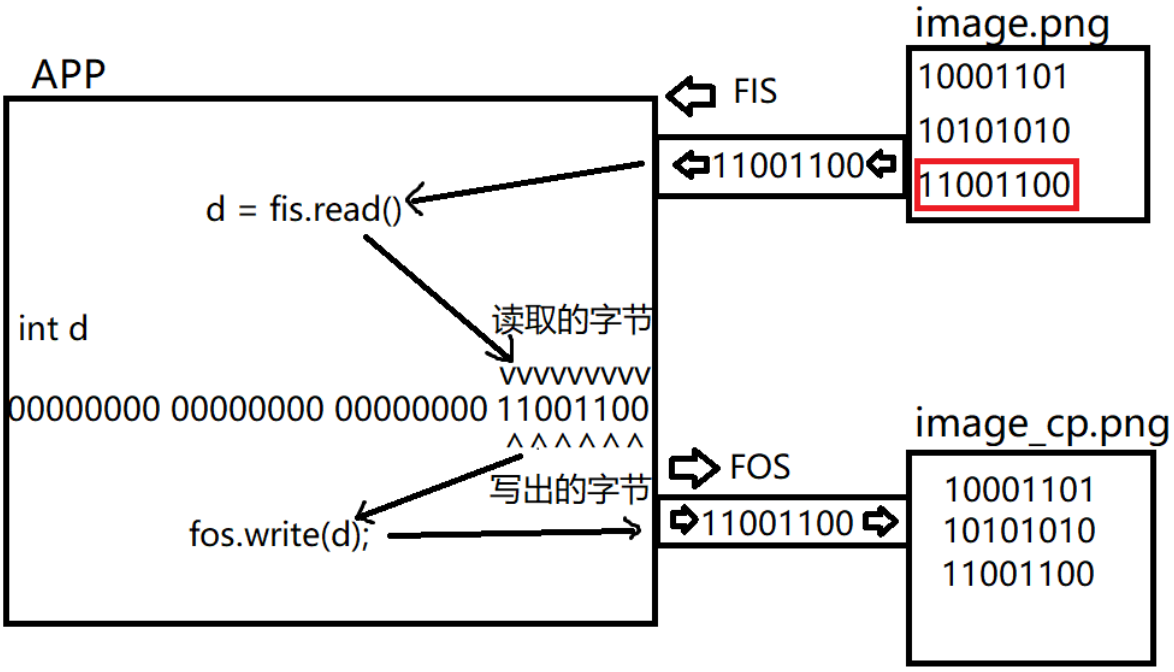
第一次读取



第二次读取



第三次读取



循环进行上述操作，直到某次fis.read()方法返回值为-1，表示读取到了文件末尾，那么就不再写出即可。

例

```
package io;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 * 文件的复制
 */
public class CopyDemo {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("image.png");
        FileOutputStream fos = new FileOutputStream("image_cp.png");

        int d;
//        while(true){
//            d = fis.read();
//            if(d==-1){
//                break;
//            }
//            fos.write(d);
//        }

        while((d = fis.read())!= -1){
            fos.write(d);
        }

        System.out.println("复制完毕!");
```

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

效率问题

上述案例在复制文件时的读写效率是很低的。因为硬盘的特性，决定着硬盘的读写效率差，而单字节读写就是在频繁调用硬盘的读写，从而产生了"木桶效应"。

为了解决这个问题，我们可以采取使用块读写的方式来复制文件，减少硬盘的实际读写的次数，从而提高效率。

块读写

- 块读:一次性读取一组字节的方式

  InputStream中定义了块读的方法

  ```
  int read(byte[] data)
  一次性读取给定字节数组总长度的字节量并存入到该数组中。
  返回值为实际读取到的字节数。如果返回值为-1表示本次没有读取到任何字节已经是流的末尾了
  ```

- 块写:一次性写出一组字节的方式

  OutputStream中定义了块写的方法

  ```
  void write(byte[] data)
  一次性将给定数组中所有字节写出
  void write(byte[] data,int offset,int len)
  一次性将data数组中从下标offset处开始的连续len个字节一次性写出
  ```

例

改为块读写形式后，复制效率得到了明显的提升

```
package io;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 * 提高每次读写的数据量减少读写次数，可以提高读写效率
 *
 * 块读取:一次性读取一组字节的方式
 * 块写:一次性写出一组字节
 */
public class CopyDemo2 {
```

```
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("image.png");
        FileOutputStream fos = new FileOutputStream("image_cp2.png");
        /*
            在InputStream中定义了块读取的方法
            int read(byte[] data)
            一次性读取给定字节数组总长度的字节量并存入到该数组中。
            返回值为实际读取到的字节数。如果返回值为-1表示本次没有读取到任何字节已经是流的末尾
了



            文件内容(6字节):
            00110011 11001100 10101010 01010101 11110000 00001111


            byte[] data = new byte[4];//4个字节的数组
            data:{00000000,00000000,00000000,00000000} 2进制表示

            第一次调用:int d = fis.read(data);
            一次性从文件中读取4(data数组的长度为4)个字节
            00110011 11001100 10101010 01010101 11110000 00001111
            ^^^^^^^^ ^^^^^^^^ ^^^^^^^^ ^^^^^^^^
            |----------读取的数据---------------|

            data:{00110011,11001100,10101010,01010101}
                  数组里的4个字节为本次读取到的全部数据
            d:4   返回值d为整数4，表示本次实际读取到了4个字节


            第二次调用:d = fis.read(data);
            一次性从文件中读取4(data数组的长度为4)个字节
            00110011 11001100 10101010 01010101 11110000 00001111 文件末尾了
                                                ^^^^^^^^ ^^^^^^^^ ^^^^^^^^
^^^^^^^^
                                               |---读取的数据-----|

            data:{11110000,00001111,10101010,01010101}
                 |-本次实际读取字节-| |---旧数据-------|
            d:2   返回值d为整数2，表示本次实际读取到了2个字节



             第二次调用:d = fis.read(data);
            一次性从文件中读取4(data数组的长度为4)个字节
            00110011 11001100 10101010 01010101 11110000 00001111 文件末尾
                                                                  ^^^^^^^^
^^^^^^^^ ^^^^^^^^ ^^^^^^^^
                                                                  已经是文件末尾

            data:{11110000,00001111,10101010,01010101}
                 |-------------旧数据-----------------|
            d:-1  返回值d为整数-1，表示已经是末尾了，本次没有读取任何数据
```

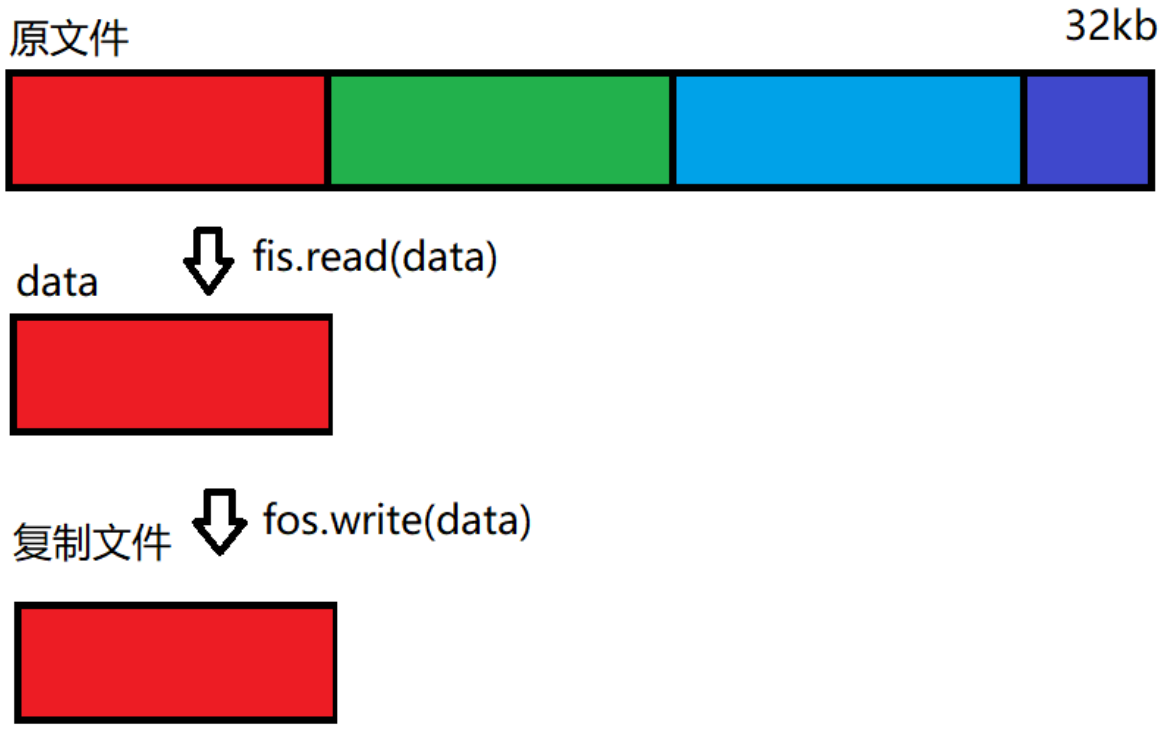
```
            OutputStream中定义了块写方法
            void write(byte[] data)
            一次性将给定数组中所有字节写出
         */
        /*
            00000000  1byte  1字节
            1024byte  1kb
            1024kb    1mb
            1024mb    1gb
            1024gb    1tb
            1024tb    1pb
         */
        byte[] data = new byte[1024*10];//10kb
        int d;//记录每次实际读取的数据量
        long start = System.currentTimeMillis();
        while( (d = fis.read(data)) !=-1){
            fos.write(data);
        }
        long end = System.currentTimeMillis();
        System.out.println("复制完毕!耗时:"+(end-start)+"ms");
        fis.close();
        fos.close();
    }
}
```

问题

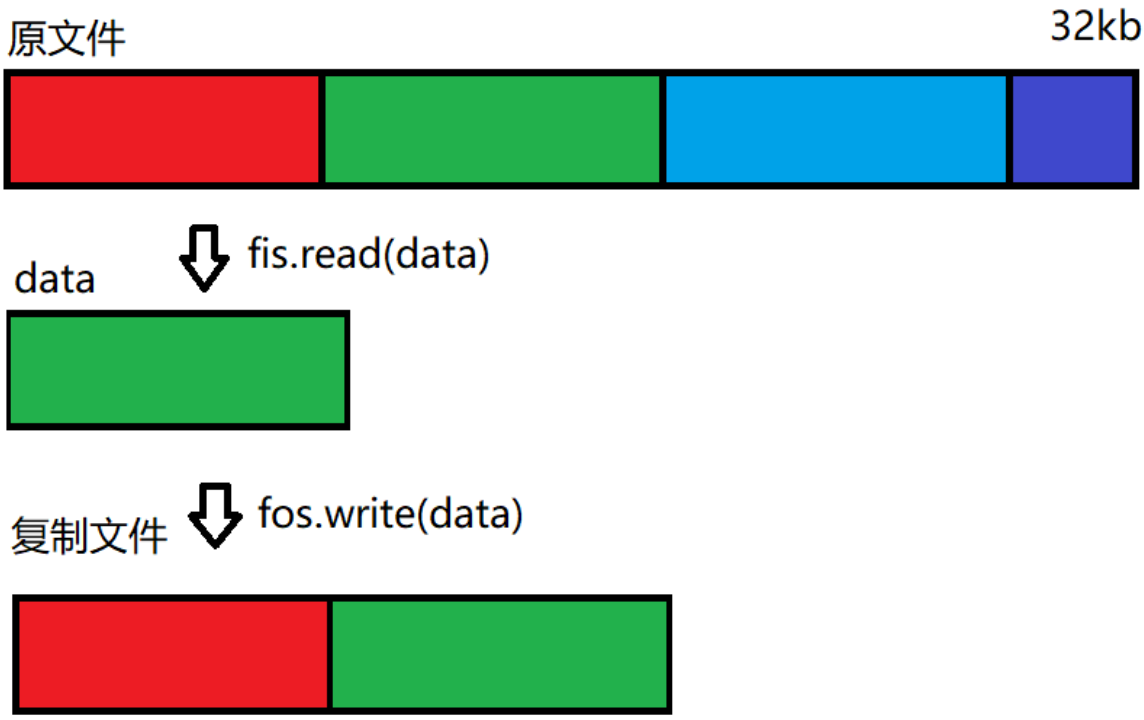
速度问题解决了，但是复制后的文件比原文件大一些。这是文件不一定是10240的倍数，这会导致最后一次读取时是读不够10240的字节数的，那么data数组中就不是所有数据都是新数据了。此时如果在写出时将data数组所有内容写出就会出现文件最后多出很多旧的数据。

示例

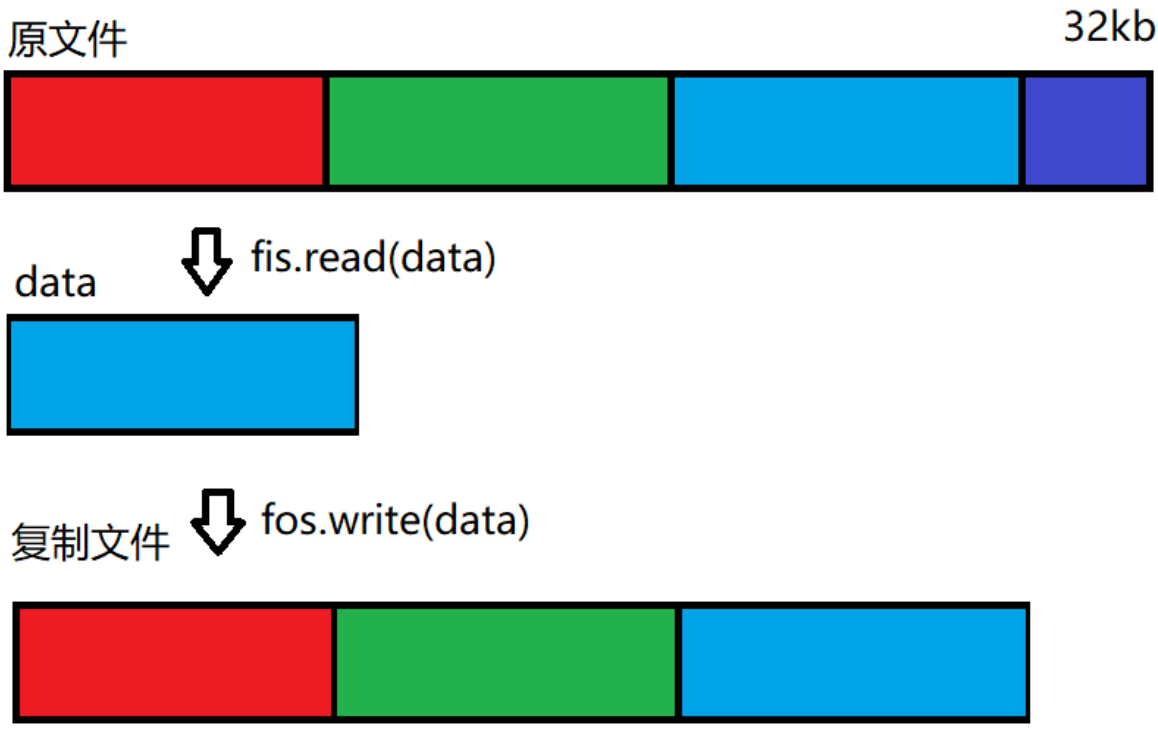
第一次操作



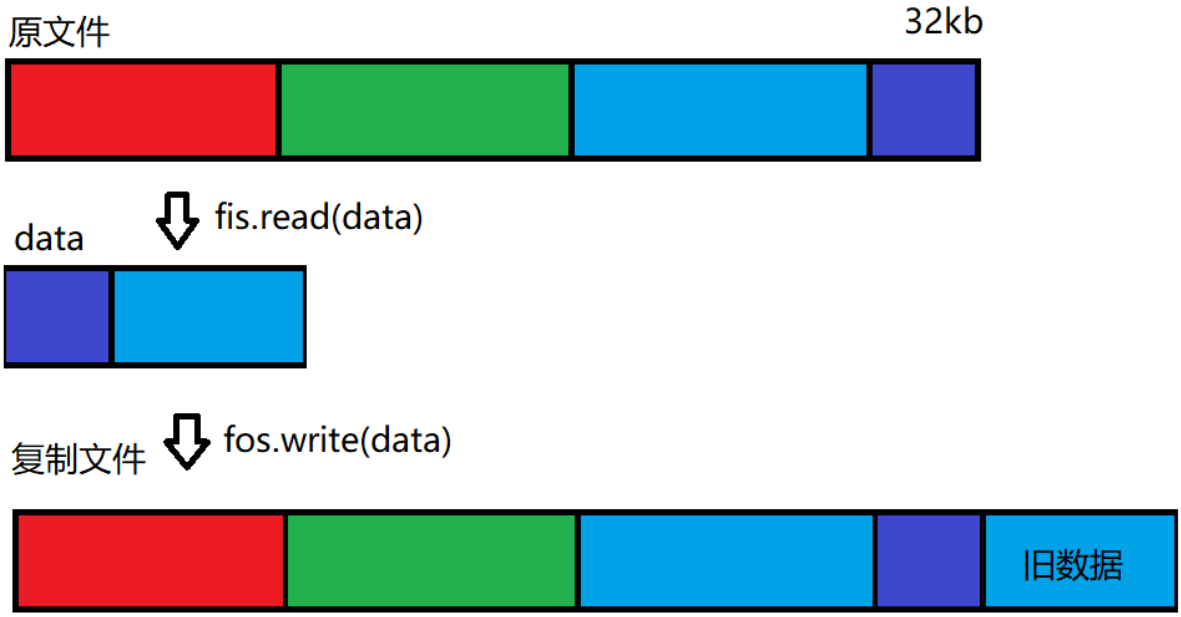
第二次操作



第三次读取操作



第四次操作



解决办法

使用OutputStream的另一个块写操作

```
void write(byte[] data,int offset,int len)
将给定数组data从offset处开始的连续len个字节一次性写出
```

例

```
package io;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
```

```java
/**
 * 提高每次读写的数据量减少读写次数，可以提高读写效率
 *
 * 块读取:一次性读取一组字节的方式
 * 块写:一次性写出一组字节
 */
public class CopyDemo2 {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("image.png");
        FileOutputStream fos = new FileOutputStream("image_cp2.png");
        byte[] data = new byte[1024*10];//10kb
        int d;//记录每次实际读取的数据量
        long start = System.currentTimeMillis();
        while( (d = fis.read(data)) !=-1){
            fos.write(data,0,d);
        }
        long end = System.currentTimeMillis();
        System.out.println("复制完毕!耗时:"+(end-start)+"ms");
        fis.close();
        fos.close();
    }
}
```

写出文本数据

文件中只能保存2进制信息，因此我们要想写出文本数据，需要先将字符串转换为2进制。

文字编码

String提供了将字符串转换为一组字节的方法

```
byte[] getBytes(Charset cs)
将当前字符串按照指定的字符集转换为一组字节

例如:
String line = "和我在成都的街头走一走，哦哦哦哦~";
byte[] data = line.getBytes(StandardCharsets.UTF_8);//将字符串按照UTF-8编码转换为一组字节
```

写入文本文件

```java
package io;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.charset.StandardCharsets;

/**
 * 写出文本数据
 */
```

```
public class WriteStringDemo {
    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream("fos.txt");

        String line = "和我在成都的街头走一走，哦哦哦哦~";
        byte[] data = line.getBytes(StandardCharsets.UTF_8);
        fos.write(data);

        fos.write("直到所有的灯都熄灭了也不停留.".getBytes(StandardCharsets.UTF_8));
        System.out.println("写出完毕!");

        fos.close();
    }
}
```

追加模式

文件输出流有两种模式

- 覆盖模式:当文件流创建时发现指定的文件已经存在了，那么会将该文件内容清空。
- 追加模式:当文件流创建时发现指定的文件已经存在了，那么文件数据全部保留通过该流新写出的数据会被陆续的追加到文件中。

文件输出流的两种模式对应的构造器

- 覆盖模式

```
FileOutputStream(String path)
FileOutputStream(File file)
```

- 追加模式

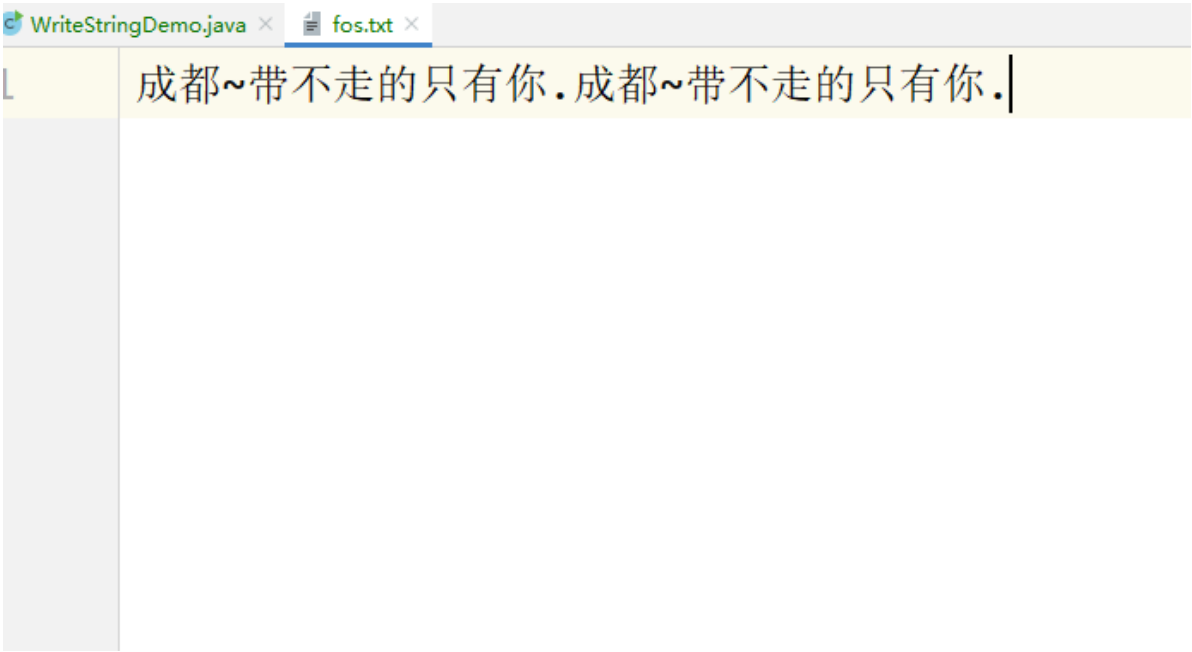
```
当append参数为true时，则开启追加模式
FileOutputStream(String path,boolean append)
FileOutputStream(File file,boolean append)
```

例

```
package io;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.charset.StandardCharsets;

/**
 * 写出文本数据
 */
public class WriteStringDemo {
    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream("fos.txt",true);
```

```
        fos.write("成都~带不走的只有你.".getBytes(StandardCharsets.UTF_8));

        System.out.println("写出完毕!");

        fos.close();
    }
}
```

执行两次上述程序会看到文件中的内容(每次执行程序写出的内容都会被保留):



读取文本数据

先将文件中的字节读取出来，然后再将这些字节按照对应的字符集转换为字符串即可

文本解码

String的构造器提供了对字节解码为字符串的操作

```
String(byte[] data,Charset cn)
将data数组中的所有字节按照指定的字符集转换为字符串
```

例

```
package io;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.charset.StandardCharsets;

/**
 * 读取文本数据
 */
public class ReadStringDemo {
```

```
    public static void main(String[] args) throws IOException {
        /*
            1:先从fos.txt中读取所有的字节
            2:再将这些字节转换为字符串
         */
        //1
        File file = new File("fos.txt");
        long len = file.length();//文件名长度

        FileInputStream fis = new FileInputStream(file);
        byte[] data = new byte[(int)len];//创建一个与文件长度等长的字节数组
        fis.read(data);//一口气将文件所有字节读入到data数组中(块读)

        //2将data数组中所有字节按照UTF-8编码还原为字符串
        String line = new String(data, StandardCharsets.UTF_8);
        System.out.println(line);

        fis.close();
    }
}
```

高级流

java将IO分为了两类

- 节点流:又称为"低级流"
  - 特点:直接链接程序与另一端的"管道"，是真实读写数据的流
  - IO一定是建立在节点流的基础上进行的。
  - 文件流就是典型的节点流(低级流)
- 处理流:又称为"高级流"
  - 特点:不能独立存在，必须链接在其他流上
  - 目的:当数据经过当前高级流时可以对数据进行某种加工操作，来简化我们的同等操作
  - 实际开发中我们经常"串联"一组高级流最终到某个低级流上，使读写数据以流水线式的加工处理完成。这一操作也被称为使"**流的链接**"。流链接也是JAVA IO的精髓所在。

缓冲流

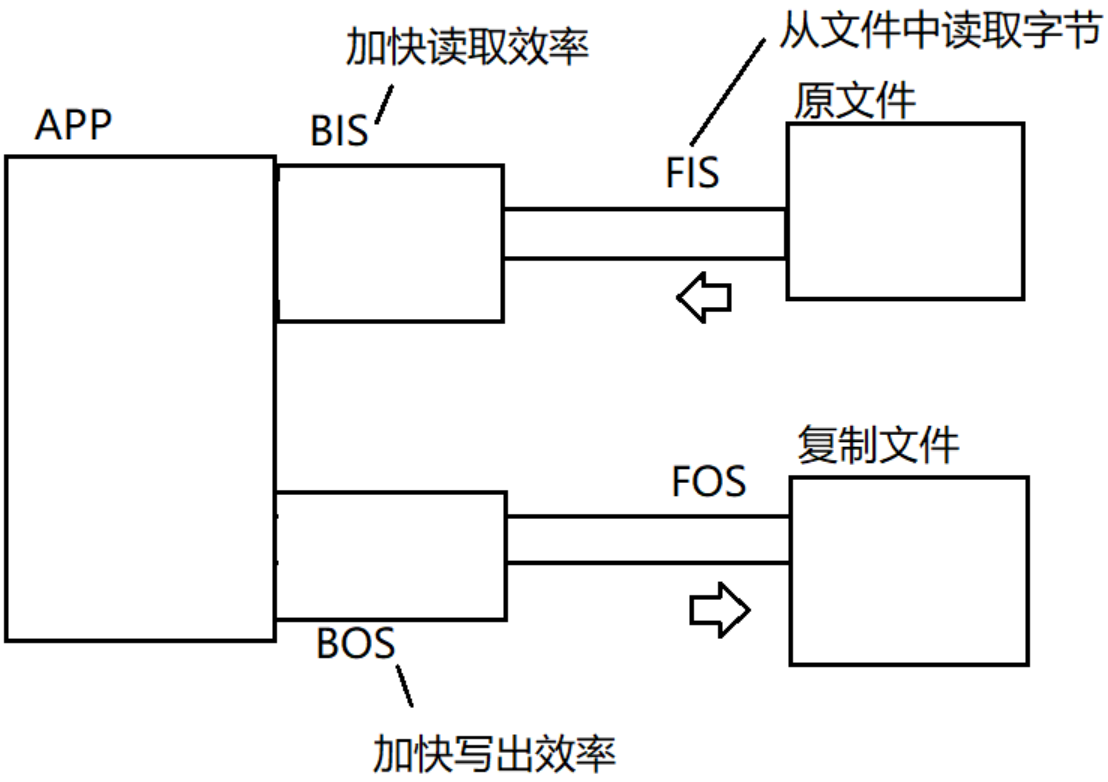
java.io.**BufferedInputStream**和**BufferedOutputStream**

功能

在流链接中的作用:**加快读写效率**

通常缓冲是最终链接在低级流上的流

例




```
package io;

import java.io.*;

/**
 * 使用缓冲流完成文件复制操作
 */
public class CopyDemo3 {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("image.png");
        BufferedInputStream bis = new BufferedInputStream(fis);

        FileOutputStream fos = new FileOutputStream("image_cp3.png");
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        int d;
        long start = System.currentTimeMillis();
        while((d=bis.read())!= -1){
            bos.write(d);
        }
        long end = System.currentTimeMillis();
```

```
        System.out.println("复制完毕!耗时:"+(end-start)+"ms");
        bis.close();
        bos.close();

    }
}
```

原理

内部定义了一个属性byte buf[]。它等同于我们之前练习复制案例时的块写操作。

并且默认创建时，该buf数组的长度为8192(8kb)长度。

缓冲流在读写数据时**总是以块读写数据**(默认是8kb)来保证读写效率的

缓冲流提供了多种构造器，可以自行指定缓冲区大小。

```
class BufferedOutputStream extends FilterOutputStream {
    The internal buffer where data is stored.
    protected byte buf[];

    The number of valid bytes in the buffer. This value is always in the range 0 through buf.length;
    elements buf[0] through buf[count-1] contain valid byte data.
    protected int count;

    Creates a new buffered output stream to write data to the specified underlying output stream.
    Params: out – the underlying output stream.
    public BufferedOutputStream( @NotNull OutputStream out) {
        this(out,  size: 8192);
    }
```

写缓冲问题

由于缓冲输出流会将写出的数据装满内部缓冲区(默认8kb的字节数组)后才会进行一次真实的写出操作。当我们的数据不足时，如果想要及时写出数据，可以调用缓冲流的flush()方法，强制将缓冲区中已经缓存的数据写出一次。

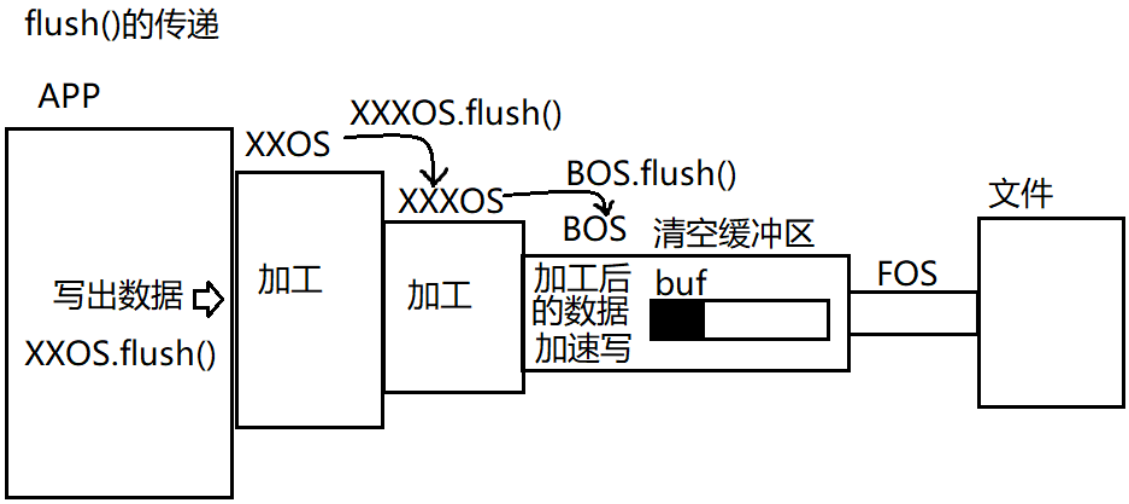
```
package io;

import java.io.BufferedOutputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.charset.StandardCharsets;

/**
 * 缓冲输出流的写缓冲问题
 */
public class BosFlushDemo {
    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream("bos.txt");
        BufferedOutputStream bos = new BufferedOutputStream(fos);
```

```
        String line = "super idol的笑容都没你的甜~";
        byte[] data = line.getBytes(StandardCharsets.UTF_8);
        bos.write(data);
        /*
            void flush()
            强制将缓冲流的缓冲取(内部维护的字节数组)中已经缓存的字节一次性写出
         */
//         bos.flush();
        System.out.println("写出完毕!");
        bos.close();//缓冲输出流的close()方法内部会自动调用一次flush()方法确保数据写出
    }
}
```

flush的传递

flush()方法是被定义在java.io.Flushable中。而字节输出流的超类java.io.OutputStream实现了

该接口，这意味着所有的字节输出流都有flush方法。而除了缓冲流之外的高级流的flush方法作用就是调用它链接的流的flush方法将该动作传递下去。最终传递给缓冲流来清空缓冲区。



对象流

java.io.**ObjectInputStream**和**ObjectOutputStream**

作用

- 对象输出流:将我们的java对象进行序列化
- 对象输入流:将java对象进行反序列化

序列化

将一个对象转换为一组**可被传输或保存**的字节。这组字节中除了包含对象本身的数据外，还会包含结构信息。

序列化的意义

实际开发中，我们通常会将对象

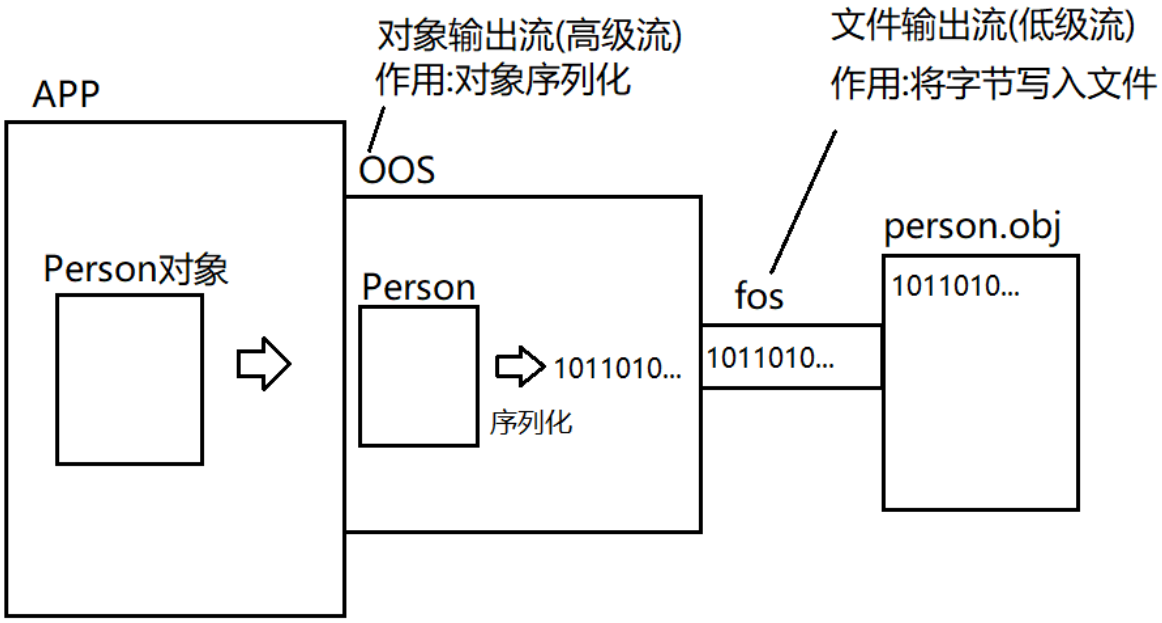
- 写入磁盘，进行长久保存
- 在网络间两台计算机中的java间进行传输

无论是保存在磁盘中还是传输，都需要将对象转换为字节后才可以进行。

对象输出流的序列化操作

```
void writeObject(Object obj)
将给定的对象转换为一组可保存或传输的字节然后通过其链接的流将字节写出
```

例:




```
package io;

import java.io.*;

/**
 * 对象流
 * 使用对象输出流完成对象序列化操作并最终保存到文件person.obj中
 */
public class OOSDemo {
    public static void main(String[] args) throws IOException {
        String name = "王克晶";
        int age = 18;
        String gender = "女";
        String[] otherInfo = {"黑","嗓门大","java技术好","大家的启蒙老师","来自廊坊佳木
斯"};
        Person p = new Person(name,age,gender,otherInfo);
        FileOutputStream fos = new FileOutputStream("person.obj");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        /*
            对象输出流提供的序列化对象方法:
            void writeObject(Object obj)
            将给定的对象转换为一组可保存或传输的字节然后通过其链接的流将字节写出

            序列化对象时要求该对象对应的类必须实现接口:java.io.Serializable
            如果写出的对象对应的类没有实现该接口，那么writeObject会抛出下面异常
            java.io.NotSerializableException
```

```
         */
        oos.writeObject(p);
        System.out.println("对象写出完毕");
        oos.close();
    }
}
```

序列化要求

对象输出流要求写出的对象必须实现接口:java.io.Serializable

上述案例中我们看到，如果写出的对象Person没有实现java.io.Serializable时会抛出异常:

java.io.NotSerializableException

总结

JAVA IO必会概念:

- java io可以让我们用标准的读写操作来完成对不同设备的读写数据工作.
- java将IO按照方向划分为输入与输出,参照点是我们写的程序.
- **输入**:用来读取数据的,是从外界到程序的方向,用于获取数据.
- **输出**:用来写出数据的,是从程序到外界的方向,用于发送数据.

java将IO比喻为"流",即:stream. 就像生活中的"电流","水流"一样,它是以同一个方向顺序移动的过程.只不过这里流动的是字节(2进制数据).所以在IO中有输入流和输出流之分,我们理解他们是连接程序与另一端的"管道",用于获取或发送数据到另一端.

因此流的读写是顺序读写的，只能顺序向后写或向后读，不能回退。

Java定义了两个超类(抽象类):

- **java.io.InputStream**:所有字节输入流的超类,其中定义了读取数据的方法.因此将来不管读取的是什么设备(连接该设备的流)都有这些读取的方法,因此我们可以用相同的方法读取不同设备中的数据

  常用方法:

  `int read()`：读取一个字节，返回的`int`值低8位为读取的数据。如果返回值为整数`-1`则表示读取到了流的末尾

  `int read(byte[] data)`：块读取，最多读取`data`数组总长度的数据并从数组第一个位置开始存入到数组中，返回值表示实际读取到的字节量，如果返回值为`-1`表示本次没有读取到任何数据，是流的末尾。

- **java.io.OutputStream**:所有字节输出流的超类,其中定义了写出数据的方法.

  常用方法:

  void write(int d)：写出一个字节，写出的是给定的int值对应2进制的低八位。

  void write(byte[] data)：块写，将给定字节数组中所有字节一次性写出。

void write(byte[]data,int off,int len)：块写，将给定字节数组从下标off处开始的连续len个字节一次性写出。

java将流分为两类:节点流与处理流:

- **节点流**:也称为**低级流**.

  节点流的另一端是明确的,是实际读写数据的流,读写一定是建立在节点流基础上进行的.

- **处理流**:也称为**高级流**.

  处理流不能独立存在,必须连接在其他流上,目的是当数据流经当前流时对数据进行加工处理来简化我们对数据的该操作.

实际应用中,我们可以通过串联一组高级流到某个低级流上以流水线式的加工处理对某设备的数据进行读写,这个过程也成为流的连接,这也是IO的精髓所在.

文件流

文件流是一对低级流，**用于读写文件的流。**

java.io.FileOutputStream文件输出流，继承自java.io.OutputStream

常用构造器

覆盖模式对应的构造器

> 覆盖模式是指若指定的文件存在，文件流在创建时会先将该文件原内容清除。

- FileOutputStream(String pathname)：创建文件输出流用于向指定路径表示的文件做写操作
- FileOutputStream(File file)：创建文件输出流用于向File表示的文件做写操作。

  注:如果写出的文件不存在文件流自动创建这个文件，但是如果该文件所在的目录不存在会抛出异常:java.io.FileNotFoundException

追加写模式对应的构造器

> 追加模式是指若指定的文件存在，文件流会将写出的数据陆续追加到文件中。

- FileOutputStream(String pathname,boolean append)：如果第二个参数为true则为追加模式，false则为覆盖模式
- FileOutputStream(File file,boolean append)：同上

常用方法:

> `void write(int d)`：向文件中写入一个字节，写入的是`int`值2进制的低八位。
>
> `void write(byte[] data)`：向文件中块写数据。将数组`data`中所有字节一次性写入文件。
>
> `void write(byte[] data,int off,int len)`：向文件中快写数据。将数组`data`中从下标`off`开始的连续`len`个字节一次性写入文件。

java.io.FileInputStream文件输入流，继承自java.io.InputStream

常用构造器

FileInputStream(String pathname) 创建读取指定路径下对应的文件的文件输入流，如果指定的文件不存在则会抛出异常java.io.FileNotFoundException

FileInputStream(File file) 创建读取File表示的文件的文件输入流，如果File表示的文件不存在则会抛出异常java.io.IOException。

常用方法

`int read()`：从文件中读取一个字节，返回的`int`值低八位有效，如果返回的`int`值为整数`-1`则表示读取到了文件末尾。

`int read(byte[] data)`：块读数据，从文件中一次性读取给定的`data`数组总长度的字节量并从数组第一个元素位置开始存入数组中。返回值为实际读取到的字节数。如果返回值为整数`-1`则表示读取到了文件末尾。

缓冲流

缓冲流是一对高级流，在流链接中链接它的**目的是加快读写效率。**缓冲流内部**默认缓冲区为8kb**，缓冲流**总是块读写数据来提高读写效率。**

java.io.BufferedOutputStream缓冲字节输出流，继承自java.io.OutputStream

常用构造器

- BufferedOutputStream(OutputStream out)：创建一个默认8kb大小缓冲区的缓冲字节输出流，并连接到参数指定的字节输出流上。
- BufferedOutputStream(OutputStream out,int size)：创建一个size指定大小(单位是字节)缓冲区的缓冲字节输出流，并连接到参数指定的字节输出流上。

常用方法

`flush()`：强制将缓冲区中已经缓存的数据一次性写出

缓冲流的写出方法功能与`OutputStream`上一致，需要知道的时`write`方法调用后并非实际写出，而是先将数据存入缓冲区(内部的字节数组中)，当缓冲区满了时会自动写出一次。

java.io.BufferedInputStream缓冲字节输出流，继承自java.io.InputStream

常用构造器

- BufferedInputStream(InputStream in)：创建一个默认8kb大小缓冲区的缓冲字节输入流，并连接到参数指定的字节输入流上。
- BufferedInputStream(InputStream in,int size)：创建一个size指定大小(单位是字节)缓冲区的缓冲字节输入流，并连接到参数指定的字节输入流上。

常用方法

缓冲流的读取方法功能与`InputStream`上一致，需要知道的时`read`方法调用后缓冲流会一次性读取缓冲区大小的字节数据并存入缓冲区，然后再根据我们调用`read`方法读取的字节数进行返回，直到缓冲区所有数据都已经通过`read`方法返回后会再次读取一组数据进缓冲区。即:块读取操作

扩展

JRE的结构