Java IO basics summary

Guide Java Java IO, Java Basics About 4720 words About 16 minutes

Introduction to IO Streams

IO stands for `Input/Output` input and output. Input is the process of inputting data into computer memory, while output is the process of outputting data to external storage (such as a database, file, or remote host). The data transfer process is similar to the flow of water, hence the name IO stream. In Java, IO streams are categorized as input streams and output streams, which are further categorized as byte streams and character streams depending on how the data is processed.

More than 40 Java IO stream classes are derived from the following four abstract base classes.

- `InputStream` / `Reader` : The base class of all input streams, the former is a byte input stream, and the latter is a character input stream.
- `OutputStream` / `Writer` : The base class of all output streams, the former is a byte output stream, and the latter is a character output stream.

Byte Stream

InputStream (byte input stream)

`InputStream` Used to read data (byte information) from the source (usually a file) ir memory. `java.io.InputStream` The abstract class is the parent class of all byte input streams.

`InputStream` Common methods:

- `read()` : Returns the next byte of data in the input stream. The returned value is between 0 and 255. If no bytes are read, the code returns `-1` , indicating end of file.
- `read(byte b[ ])` : Reads some bytes from the input stream and stores them `b` in array . If array `b` is of length zero, no bytes are read. If no bytes are available to read, return `-1` . If bytes are available to read, the number of bytes read is at most equal to `b.length` the number of bytes read, and return . This method is equivalent to `read(b, 0, b.length)` .
- `read(byte b[], int off, int len)` : `read(byte b[ ])` Added `off` the parameter (offset) and `len` parameter (maximum number of bytes to read) based on the method.
- `skip(long n)` : Ignore n bytes in the input stream and return the number of bytes actually ignored.
- `available()` : Returns the number of bytes that can be read from the input stream.
- `close()` : Close the input stream to release related system resources.

Starting from Java 9, `InputStream` several new useful methods have been added:

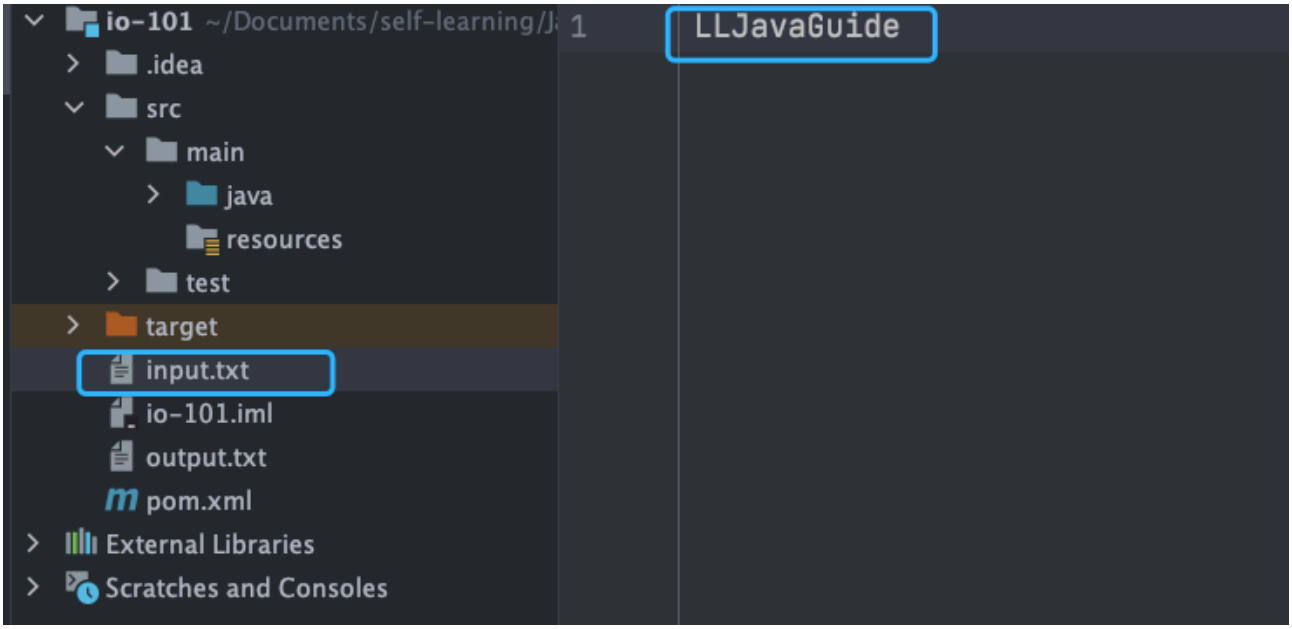
- `readAllBytes()` : Read all bytes from the input stream and return a byte array.
- `readNBytes(byte[] b, int off, int len)` : Block until `len` bytes are read.
- `transferTo(OutputStream out)` : Passes all bytes from one input stream to an output stream.

`FileInputStream` It is a commonly used byte input stream object that can directly specify the file path, read single-byte data directly, or read it into a byte array.

`FileInputStream` Code example:

```java
try (InputStream fis = new FileInputStream("input.txt")) {
    System.out.println("Number of remaining bytes:"
            + fis.available());
    int content;
    long skip = fis.skip(2);
    System.out.println("The actual number of bytes skipped:" +
skip);
    System.out.print("The content read from file:");
    while ((content = fis.read()) != -1) {
        System.out.print((char) content);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

`input.txt` File contents:



Output:

```plain
Number of remaining bytes:11
The actual number of bytes skipped:2
The content read from file:JavaGuide
```

However, we generally do not use it alone `FileInputStream`, but usually `BufferedInputStream` use it in conjunction with (byte buffer input stream, which will be discussed later).

The following code is quite common in our projects. We `readAllBytes()` read all bytes of the input stream and assign them directly to an `String` object.

```java
//         BufferedInputStream
BufferedInputStream bufferedInputStream = new
BufferedInputStream(new FileInputStream("input.txt"));
//                     String
String result = new String(bufferedInputStream.readAllBytes());
System.out.println(result);
```

`DataInputStream` Used to read data of a specified type. It cannot be used alone and must be combined with other streams, such `FileInputStream` as

```java
FileInputStream fileInputStream = new FileInputStream("input.txt");
//     fileInputStream
DataInputStream dataInputStream = new
DataInputStream(fileInputStream);
//
dataInputStream.readBoolean();
dataInputStream.readInt();
dataInputStream.readUTF();
```

`ObjectInputStream` Used to read Java objects from an input stream (deserialization) and `ObjectOutputStream` to write objects to an output stream (serialization).

```java
ObjectInputStream input = new ObjectInputStream(new
FileInputStream("object.data"));
MyClass object = (MyClass) input.readObject();
input.close();
```

In addition, the class used for serialization and deserialization must implement `Serializable` the interface. If there are attributes in the object that you do not want to be serialized, use `transient` the modifier.

OutputStream (byte output stream)

`OutputStream` Used to write data (byte information) to a destination (usually a file). `java.io.OutputStream` The abstract class is the parent class of all byte output streams.

`OutputStream` Common methods:

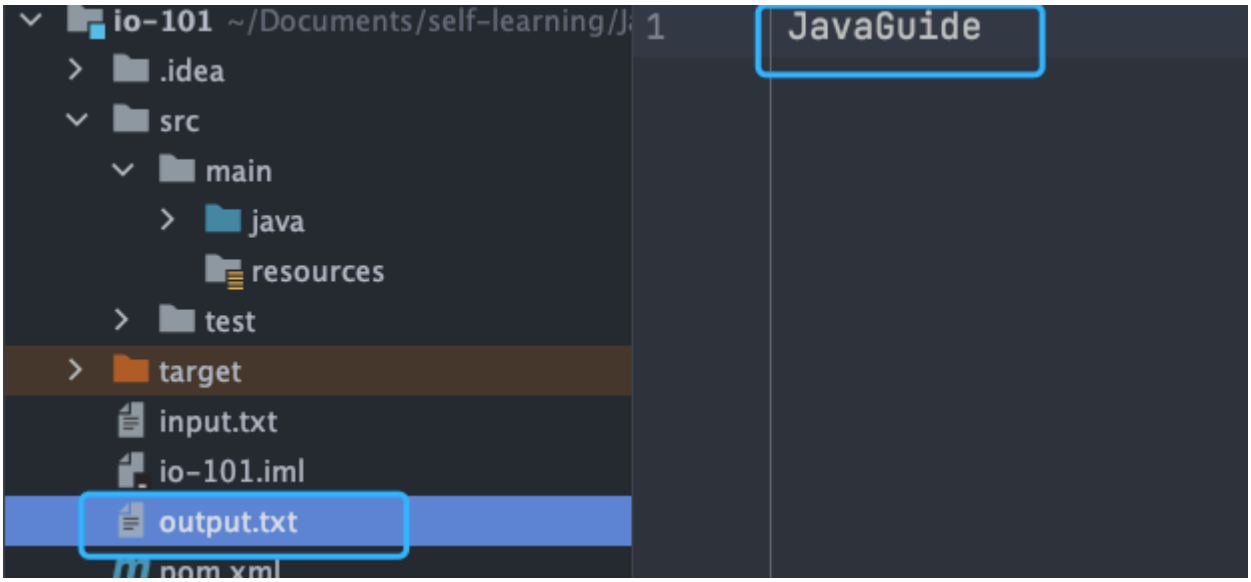
- `write(int b)` : Writes specific bytes to the output stream.
- `write(byte b[ ])` : `b` Writes the array to the output stream, equivalent to `write(b, 0, b.length)` .
- `write(byte[] b, int off, int len)` : Based on the method , the parameter (offset) and parameter (maximum number of bytes to read) `write(byte b[ ])` are added `. off len`
- `flush()` : Flushes this output stream and forces any buffered output bytes to be written out.
- `close()` : Close the output stream to release related system resources.

`FileOutputStream` It is the most commonly used byte output stream object. It can directly specify the file path, directly output single-byte data, or output a specified byte array.

`FileOutputStream` Code example:

```java
try (FileOutputStream output = new FileOutputStream("output.txt")) {
    byte[] array = "JavaGuide".getBytes();
    output.write(array);
} catch (IOException e) {
    e.printStackTrace();
}
```

Run results:



Similarly `FileInputStream` , `FileOutputStream` it is usually `BufferedOutputStream` used in conjunction with (byte buffer output stream, which will be discussed later).

```java
FileOutputStream fileOutputStream = new FileOutputStream("output.txt");
BufferedOutputStream bos = new BufferedOutputStream(fileOutputStream)
```

`DataOutputStream` Used to write data of a specified type. It cannot be used alone and must be combined with other streams, such `FileOutputStream` as

```java
//
FileOutputStream fileOutputStream = new
FileOutputStream("out.txt");
DataOutputStream dataOutputStream = new
DataOutputStream(fileOutputStream);
//
dataOutputStream.writeBoolean(true);
dataOutputStream.writeByte(1);
```

`ObjectInputStream` Used to read Java objects from the input stream (`ObjectInputStream` , deserialization) and `ObjectOutputStream` write objects to the output stream (`ObjectOutputStream` , serialization).

```java
ObjectOutputStream output = new ObjectOutputStream(new
FileOutputStream("file.txt")
Person person = new Person("Guide ", "JavaGuide   ");
output.writeObject(person);
```

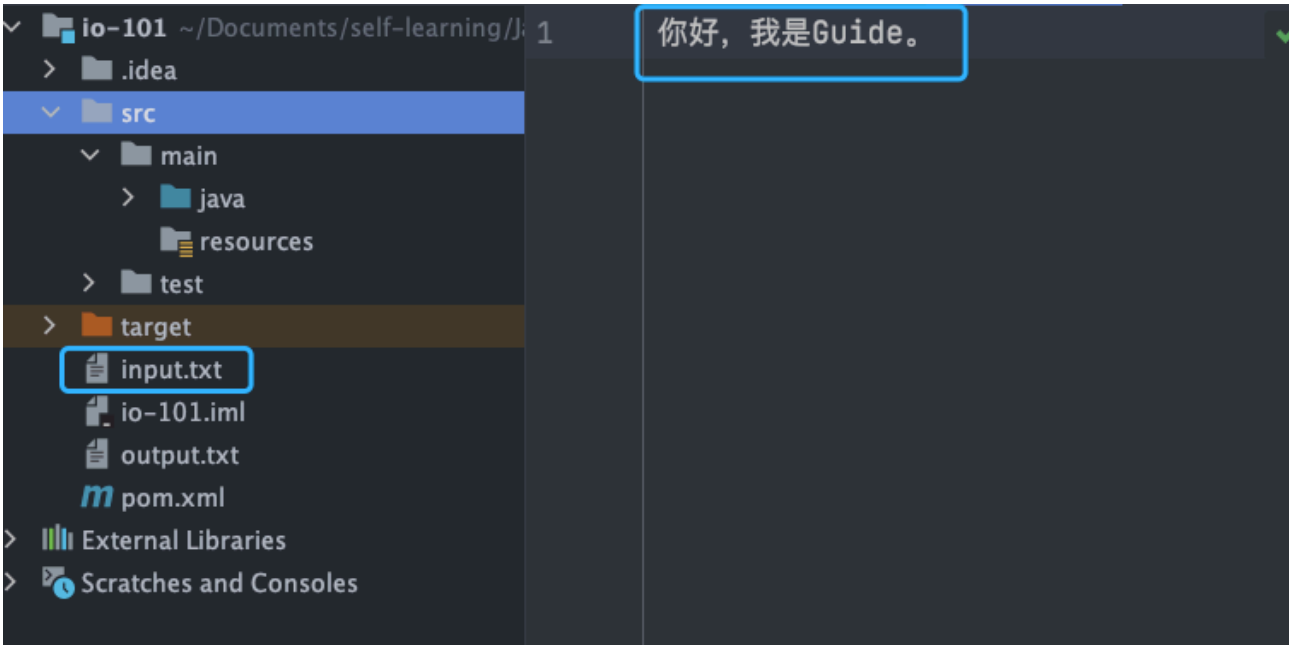
Character Stream

Whether reading or writing files or sending or receiving over the network, the smallest storage unit of information is a byte. **So why are I/O stream operations divided into byte stream operations and character stream operations?**

I think there are two main reasons:

- The character stream is obtained by converting bytes by the Java virtual machine, which is a relatively time-consuming process.
- If we don't know the encoding type, it is easy to have garbled characters.

The garbled code problem can be easily reproduced. We only need to change the file content `FileInputStream` in the code example mentioned above `input.txt` to Chinese. The original code does not need to be changed.



Output:

```java
Number of remaining bytes:9
The actual number of bytes skipped:2
The content read from file:§å®¶å¥½
```

It can be clearly seen that the read content has become garbled.

Therefore, I/O streams simply provide an interface for directly manipulating characters, making it convenient for us to perform stream operations on characters. If you are working with media files such as audio files and pictures, it is better to use byte streams, and if you are working with characters, it is better to use character streams.

The character stream uses `Unicode` encoding by default, and we can customize the encoding through the construction method.

Unicode itself is just a character set. It assigns each character a unique numeric number and doesn't specify a specific storage method. UTF-8, UTF-16, and UTF-32 are all Unicode encodings, each using different numbers of bytes to represent a Unicode character. For example, in UTF-8, English takes up 1 byte, while Chinese takes up 3 bytes.

Reader (character input stream)

`Reader` Used to read data (character information) from the source (usually a file) int memory. `java.io.Reader` The abstract class is the parent class of all character input streams.

`Reader` for reading text, `InputStream` and for reading raw bytes.

`Reader` Common methods:

- `read()` : Read a character from the input stream.
- `read(char[] cbuf)` : Reads some characters from the input stream and stores them into a character array `cbuf` , equivalent to `read(cbuf, 0, cbuf.length)` .
- `read(char[] cbuf, int off, int len)` : `read(char[] cbuf)` Added `off` the parameter (offset) and `len` parameter (maximum number of characters to read) based on the method.
- `skip(long n)` : Ignore n characters in the input stream and return the number of characters actually ignored.
- `close()` : Close the input stream and release related system resources.

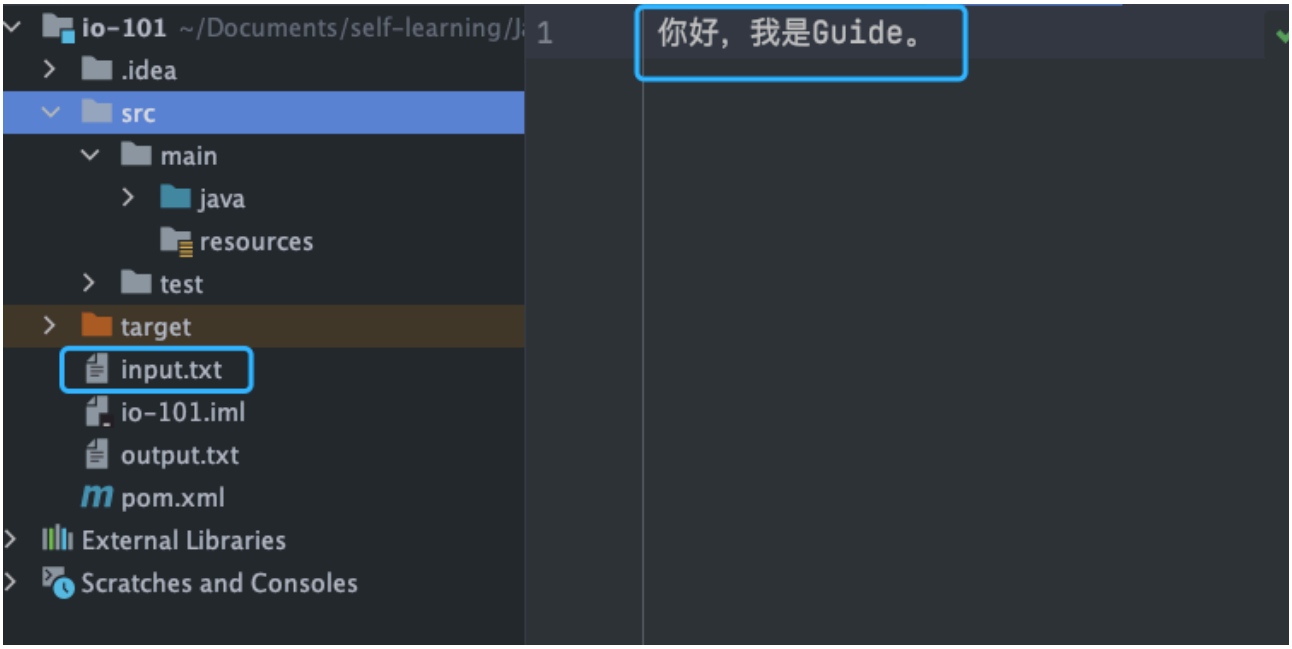
`InputStreamReader` It is a bridge for converting byte stream into character stream. Its subclass `FileReader` is an encapsulation based on this basis and can directly operate character files.

```java
//
public class InputStreamReader extends Reader {
}
//
public class FileReader extends InputStreamReader {
}
```

`FileReader` Code example:

```java
try (FileReader fileReader = new FileReader("input.txt");) {
    int content;
    long skip = fileReader.skip(3);
    System.out.println("The actual number of bytes skipped:" +
skip);
    System.out.print("The content read from file:");
    while ((content = fileReader.read()) != -1) {
        System.out.print((char) content);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

`input.txt` File contents:



Output:

```plain
The actual number of bytes skipped:3
The content read from file:    Guide
```

Writer (character output stream)

`Writer` Used to write data (character information) to a destination (usually a file). `java.io.Writer` The abstract class is the parent class of all character output streams.

`Writer` Common methods:

- `write(int c)` : Write a single character.
- `write(char[] cbuf)` : Writes a character array `cbuf` , equivalent to `write(cbuf, 0, cbuf.length)` .
- `write(char[] cbuf, int off, int len)` : `write(char[] cbuf)` Added `off` the parameter (offset) and `len` parameter (maximum number of characters to read) based on the method.
- `write(String str)` : Write a string, equivalent to `write(str, 0, str.length())` .
- `write(String str, int off, int len)` : `write(String str)` Added `off` the parameter (offset) and `len` parameter (maximum number of characters to read) based on the method.
- `append(CharSequence csq)` : Appends the specified character sequence to the specified `Writer` object and returns the `Writer` object.
- `append(char c)` : Appends the specified characters to the specified `Writer` object and returns the `Writer` object.

- `flush()` : Flushes this output stream and forces any buffered output characters to be written.
- `close()` : Close the output stream to release related system resources.

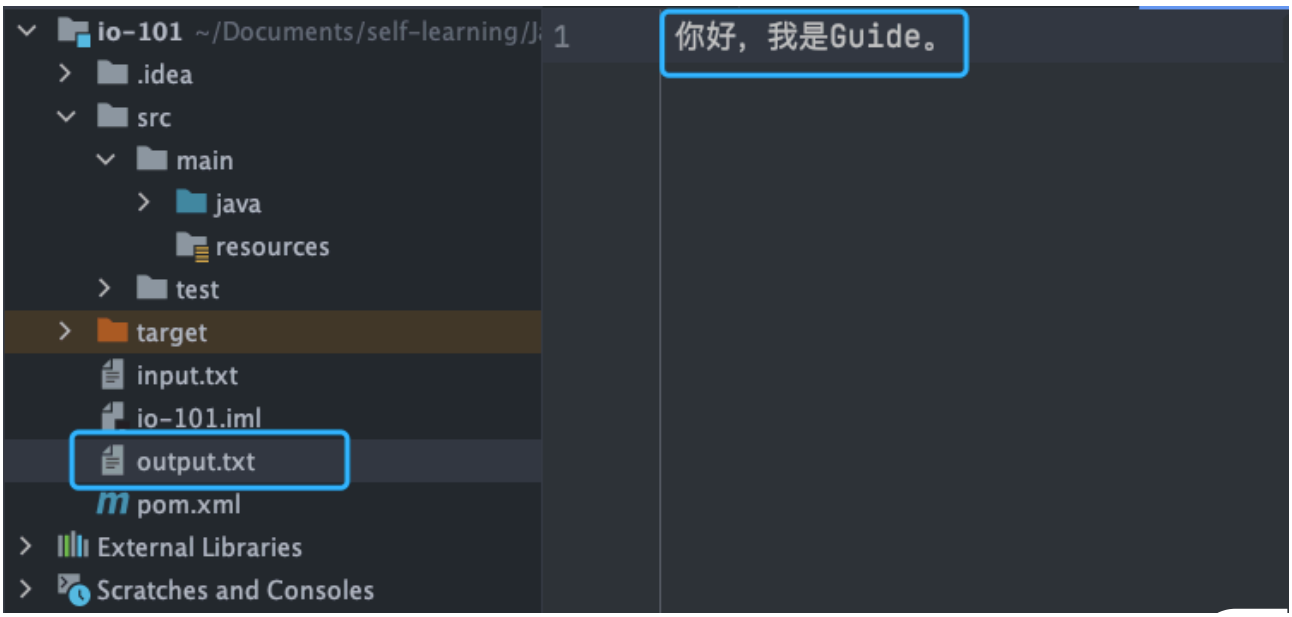
`OutputStreamWriter` It is a bridge for converting character stream into byte stream. Its subclass `FileWriter` is an encapsulation based on this basis, which can write characters directly to files.

```java
//
public class OutputStreamWriter extends Writer {
}
//
public class FileWriter extends OutputStreamWriter {
}
```

`FileWriter` Code example:

```java
try (Writer output = new FileWriter("output.txt")) {
    output.write("        Guide  ");
} catch (IOException e) {
    e.printStackTrace();
}
```

Output:



Byte buffer stream

IO operations are very performance-intensive. Buffered streams load data into a buffer and read/write multiple bytes at a time, thereby avoiding frequent IO operations and improving stream transmission efficiency.

The byte buffer stream uses the decorator pattern here to enhance the functionality of `InputStream` and `OutputStream` subclass objects.

For example, we can `BufferedInputStream` enhance the functionality of by (byte buffer input stream) `FileInputStream` .

```java
//         BufferedInputStream
BufferedInputStream bufferedInputStream = new
BufferedInputStream(new FileInputStream("input.txt"));
```

The performance difference between byte streams and byte buffer streams primarily occurs when calling the `write(int b)` and `read()` methods, which read only one byte at a time. Because byte buffer streams have an internal buffer (byte array), they first store the read bytes in the cache, significantly reducing I/O times and improving reading efficiency.

I used the `write(int b)` and `read()` methods to copy a PDF file using byte stream and byte buffer stream respectively. `524.9 mb` The time taken is as follows:

```plain
            PDF         :15428
               PDF          :2555062
```

The time difference between the two is very large, and the time consumed by the buffered stream is 1/165 of that of the byte stream.

The test code is as follows:

```java
@Test
void copy_pdf_to_another_pdf_buffer_stream() {
    //
    long start = System.currentTimeMillis();
    try (BufferedInputStream bis = new BufferedInputStream(new
FileInputStream("                    .pdf"));
         BufferedOutputStream bos = new BufferedOutputStream(new
```

```
FileOutputStream("                    -    .pdf"))) {
        int content;
        while ((content = bis.read()) != -1) {
            bos.write(content);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    //
    long end = System.currentTimeMillis();
    System.out.println("            PDF        :" + (end - start) +
"    ");
}

@Test
void copy_pdf_to_another_pdf_stream() {
    //
    long start = System.currentTimeMillis();
    try (FileInputStream fis = new FileInputStream("
    .pdf");
         FileOutputStream fos = new FileOutputStream("
       -    .pdf")) {
        int content;
        while ((content = fis.read()) != -1) {
            fos.write(content);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    //
    long end = System.currentTimeMillis();
    System.out.println("            PDF        :" + (end - start) +
"    ");
}
```

If you call the `read(byte b[])` and `write(byte b[], int off, int len)` methods that write a byte array, as long as the byte array is of the right size, the performance difference between the two is actually small and can be basically ignored.

This time we use `read(byte b[])` the and `write(byte b[], int off, int len)` methods to copy a 524.9 MB PDF file using byte stream and byte buffer stream respectively. The time taken is as follows:

```plain
            PDF        :695
                PDF         :989
```

The time difference between the two is not very large, and the performance of the buffered stream is slightly better.

The test code is as follows:

```java
@Test
void copy_pdf_to_another_pdf_with_byte_array_buffer_stream() {
    //
    long start = System.currentTimeMillis();
    try (BufferedInputStream bis = new BufferedInputStream(new
FileInputStream("                    .pdf"));
         BufferedOutputStream bos = new BufferedOutputStream(new
FileOutputStream("                    -    .pdf"))) {
        int len;
        byte[] bytes = new byte[4 * 1024];
        while ((len = bis.read(bytes)) != -1) {
            bos.write(bytes, 0, len);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    //
    long end = System.currentTimeMillis();
    System.out.println("            PDF         :" + (end - start) +
"     ");
}

@Test
void copy_pdf_to_another_pdf_with_byte_array_stream() {
    //
    long start = System.currentTimeMillis();
    try (FileInputStream fis = new FileInputStream("
   .pdf");
         FileOutputStream fos = new FileOutputStream("
      -    .pdf")) {
        int len;
        byte[] bytes = new byte[4 * 1024];
        while ((len = fis.read(bytes)) != -1) {
            fos.write(bytes, 0, len);
```

```
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    //
    long end = System.currentTimeMillis();
    System.out.println("            PDF        :" + (end - start) +
"    ");
}
```

BufferedInputStream (byte buffered input stream)

`BufferedInputStream` When reading data (byte information) from a source (usually a file) into memory, the system does not read the data byte by byte. Instead, the read bytes are first stored in a buffer and then read individually from the internal buffer. This significantly reduces the number of IO operations and improves reading efficiency.

`BufferedInputStream` A buffer is maintained internally, which is actually a byte array. `BufferedInputStream` This conclusion can be drawn by reading the source code.

java

```
public
class BufferedInputStream extends FilterInputStream {
    //
    protected volatile byte buf[];
    //
    private static int DEFAULT_BUFFER_SIZE = 8192;
    //
    public BufferedInputStream(InputStream in) {
        this(in, DEFAULT_BUFFER_SIZE);
    }
    //
    public BufferedInputStream(InputStream in, int size) {
        super(in);
        if (size <= 0) {
            throw new IllegalArgumentException("Buffer size <= 0");
        }
        buf = new byte[size];
    }
}
```

The default buffer size is **8192** bytes. Of course, you can also `BufferedInputStream(InputStream in, int size)` specify the buffer size through this construction method.

BufferedOutputStream (byte buffered output stream)

`BufferedOutputStream` When writing data (byte information) to the destination (usually a file), it will not write byte by byte, but will first store the bytes to be written in the buffer area, and then write the bytes separately from the internal buffer. This greatly reduces the number of IO times and improves efficiency.

```java
try (BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("output.txt"))) {
    byte[] array = "JavaGuide".getBytes();
    bos.write(array);
} catch (IOException e) {
    e.printStackTrace();
}
```

Similarly `BufferedInputStream` , `BufferedOutputStream` a buffer is maintained internally, and the size of this buffer is also **8192** bytes.

Character buffer stream

`BufferedReader` (Character-buffered input stream) and `BufferedWriter` (Character-buffered output stream) are similar to `BufferedInputStream` (Byte-buffered input stream) and `BufferedOutputStream` (Byte-buffered input stream), both of which maintain a byte array as a buffer. However, the former is mainly used to operate on character information.

Print Stream

Do you often use the following code?

```java
System.out.print("Hello ");
System.out.println("Hello ");
```

`System.out` It is actually used to obtain an `PrintStream` object, `print` and the method actually calls the method `PrintStream` of the object `write` .

`PrintStream` Belongs to byte print stream, corresponding to `PrintWriter` (character print stream). `PrintStream` It is `OutputStream` a subclass of , which `PrintWriter` is `Writer` a subclass of .

```java
public class PrintStream extends FilterOutputStream
    implements Appendable, Closeable {
}
public class PrintWriter extends Writer {
}
```

Random Access Streams

The random access stream introduced here refers to the support for jumping to any position in the file for reading and writing `RandomAccessFile` .

`RandomAccessFile` The construction method is as follows, we can specify `mode` (read and write mode).

```java
// openAndDelete          false
public RandomAccessFile(File file, String mode)
    throws FileNotFoundException {
    this(file, mode, false);
}
//
private RandomAccessFile(File file, String mode, boolean
openAndDelete)  throws FileNotFoundException{
  //
}
```

There are four main read and write modes:

- `r` : Read-only mode.
- `rw` : Read-write mode
- `rws` : Synchronously `rw` updates `rws` changes to the "file contents" or "metadata" to the external storage device.
- `rwd` : Synchronously `rw` updates `rwd` changes to the "file contents" to the external storage device.

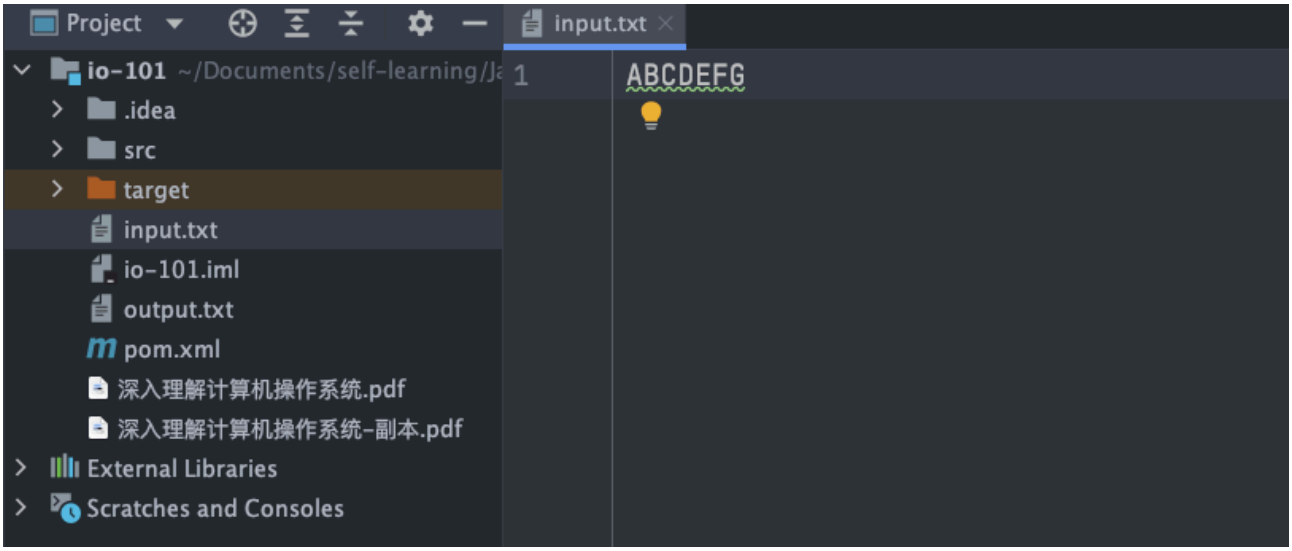
File content refers to the data actually stored in the file, while metadata is used to describe file attributes such as file size, creation and modification time.

`RandomAccessFile` There is a file pointer in that indicates the location of the next byte to be written or read. We can set the offset of the file pointer (bytes from the beginning of the file) using the method `RandomAccessFile` of . To get the current position of the file pointer, use the method. `seek(long pos)` `pos` `getFilePointer()`

`RandomAccessFile` Code example:

```java
RandomAccessFile randomAccessFile = new RandomAccessFile(new File("input.txt"), "rw");
System.out.println("               " +
randomAccessFile.getFilePointer() + ",             " + (char) randomAccessFile.read() + "                " + randomAccessFile.getFilePointer());
//              6
randomAccessFile.seek(6);
System.out.println("               " +
randomAccessFile.getFilePointer() + ",             " + (char) randomAccessFile.read() + "                " + randomAccessFile.getFilePointer());
//         7
randomAccessFile.write(new byte[]{'H', 'I', 'J', 'K'});
//              0
randomAccessFile.seek(0);
System.out.println("               " +
randomAccessFile.getFilePointer() + ",             " + (char) randomAccessFile.read() + "                " + randomAccessFile.getFilePointer());
```

`input.txt` File contents:



Output:

```plain
0,  A  1
6,  G  7
0,  A  1
```

`input.txt` The file content becomes `ABCDEFGHIJK` .

`RandomAccessFile` `write` When writing an object, if there is already data in the corresponding location, it will be overwritten .

```java
RandomAccessFile randomAccessFile = new RandomAccessFile(new File("input.txt"), "rw");
randomAccessFile.write(new byte[]{'H', 'I', 'J', 'K'});
```

`input.txt` Suppose the file content becomes before running the above program `ABCD` , and becomes after running it `HIJK` .

`RandomAccessFile` A common application is resuming large file **uploads** . What is resumable upload? Simply put, if a file upload is paused or fails (for example, due to a network issue), instead of resuming the upload, only the unsuccessful file segments need to be uploaded. Segmented upload (splitting a file into multiple segments) is the foundation of resumable uploads.

`RandomAccessFile` It can help us merge file fragments. The sample code is as follows:

```java
public boolean merge(String fileName) throws IOException {
    byte[] buffer = new byte[1024 * 10];
    int len = -1;
    try (RandomAccessFile oSavedFile = new RandomAccessFile(fileName, "rw")) {
      for (int i = 0; i < DOWNLOAD_THREAD_NUM; i++) {
        try (BufferedInputStream bis = new BufferedInputStream(
          new FileInputStream(fileName + FILE_TEMP_SUFFIX + i))) {
          while ((len = bis.read(buffer)) != -1) { // 读到文件末尾则返回-1
            oSavedFile.write(buffer, 0, len);
          }
        }
      }
      LogUtils.info("文件合并完毕 {}", fileName);
    } catch (Exception e) {
      e.printStackTrace();
      return false;
    }
    return true;
}
```

I covered the large file upload problem in detail in the Java Interview Guide .

`RandomAccessFile` The implementation relies on `FileDescriptor` (file descriptors) and `FileChannel` (memory mapped files).

Recently Updated2025/2/21 13:59
Contributors: guide , MrDecadent , Raxcl , xurunhao , Zhongtao Miao , 15168387900 , Jidcoo , Guide , Mr.Hope , paigeman , MonsterFanSec