Java NIO Buffer

By Jakob Jenkov

Table of Contents

Java NIO Buffers are used when interacting with NIO Channels. As you know, data is read from channels ir and written from buffers into channels.

A buffer is essentially a block of memory into which you can write data, which you can then later read again memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to wo memory block.

Basic Buffer Usage

Using a `Buffer` to read and write data typically follows this little 4-step process:

1. Write data into the Buffer
2. Call `buffer.flip()`
3. Read data out of the Buffer
4. Call `buffer.clear()` or `buffer.compact()`

When you write data into a buffer, the buffer keeps track of how much data you have written. Once you neec data, you need to switch the buffer from writing mode into reading mode using the `flip()` method call. In r mode the buffer lets you read all the data written into the buffer.

Once you have read all the data, you need to clear the buffer, to make it ready for writing again. You can do ways: By calling `clear()` or by calling `compact()`. The `clear()` method clears the whole buffer. The `c` method only clears the data which you have already read. Any unread data is moved to the beginning of the data will now be written into the buffer after the unread data.

Here is a simple `Buffer` usage example, with the write, flip, read and clear operations maked in bold:

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();
```

efficient developer.


```
//create buffer with capacity of 48 bytes
ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf); //read into buffer.
while (bytesRead != -1) {

  buf.flip();  //make buffer ready for read

  while(buf.hasRemaining()){
      System.out.print((char) buf.get()); // read 1 byte at a time
  }

  buf.clear(); //make buffer ready for writing
  bytesRead = inChannel.read(buf);
}
aFile.close();
```

Buffer Capacity, Position and Limit

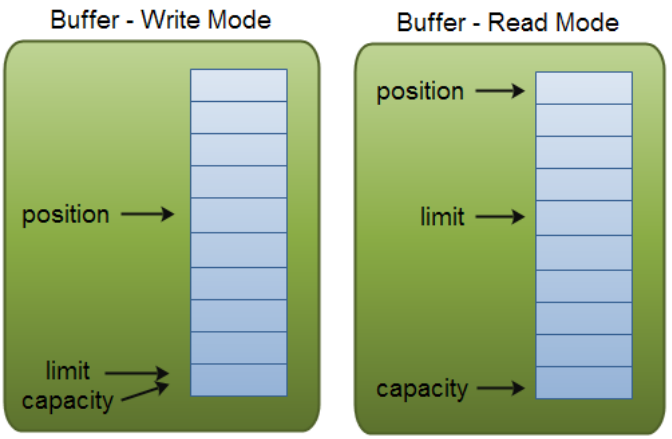
A buffer is essentially a block of memory into which you can write data, which you can then later read again memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to wo memory block.

A `Buffer` has three properties you need to be familiar with, in order to understand how a `Buffer` works. T

- capacity
- position
- limit

The meaning of `position` and `limit` depends on whether the `Buffer` is in read or write mode. Capacit means the same, no matter the buffer mode.

Here is an illustration of capacity, position and limit in write and read modes. The explanation follows in the after the illustration.



Buffer capacity, position and limit in write and read mode.

Capacity

Being a memory block, a `Buffer` has a certain fixed size, also called its "capacity". You can only write `cap` bytes, longs, chars etc. into the Buffer. Once the Buffer is full, you need to empty it (read the data, or clear it) can write more data into it.

Position

When you write data into the `Buffer`, you do so at a certain position. Initially the position is 0. When a byte has been written into the `Buffer` the position is advanced to point to the next cell in the buffer to insert data Position can maximally become `capacity - 1`.

When you read data from a `Buffer` you also do so from a given position. When you flip a `Buffer` from wri reading mode, the position is reset back to 0. As you read data from the `Buffer` you do so from `position` `position` is advanced to next position to read.

Limit

In write mode the limit of a `Buffer` is the limit of how much data you can write into the buffer. In write mode equal to the capacity of the `Buffer`.

When flipping the `Buffer` into read mode, limit means the limit of how much data you can read from the da
Therefore, when flipping a `Buffer` into read mode, limit is set to write position of the write mode. In other w
can read as many bytes as were written (limit is set to the number of bytes written, which is marked by positi

Buffer Types

Java NIO comes with the following **Buffer** types:

- ByteBuffer
- MappedByteBuffer
- CharBuffer
- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

As you can see, these `Buffer` types represent different data types. In other words, they let you work with th
the buffer as char, short, int, long, float or double instead.

The `MappedByteBuffer` is a bit special, and will be covered in its own text.

Allocating a Buffer

To obtain a `Buffer` object you must first allocate it. Every `Buffer` class has an `allocate()` method tha
Here is an example showing the allocation of a `ByteBuffer`, with a capacity of 48 bytes:

```
ByteBuffer buf = ByteBuffer.allocate(48);
```

Here is an example allocating a `CharBuffer` with space for 1024 characters:

```
CharBuffer buf = CharBuffer.allocate(1024);
```

Writing Data to a Buffer

You can write data into a `Buffer` in two ways:

1. Write data from a `Channel` into a `Buffer`
2. Write data into the `Buffer` yourself, via the buffer's `put()` methods.

Here is an example showing how a `Channel` can write data into a `Buffer`:

```
int bytesRead = inChannel.read(buf); //read into buffer.
```

Here is an example that writes data into a `Buffer` via the `put()` method:

```
buf.put(127);
```

There are many other versions of the `put()` method, allowing you to write data into the `Buffer` in many di
For instance, writing at specific positions, or writing an array of bytes into the buffer. See the JavaDoc for the
buffer implementation for more details.

flip()

The `flip()` method switches a `Buffer` from writing mode to reading mode. Calling `flip()` sets the `pos`
back to 0, and sets the `limit` to where position just was.

In other words, `position` now marks the reading position, and `limit` marks how many bytes, chars etc. w
into the buffer - the limit of how many bytes, chars etc. that can be read.

Reading Data from a Buffer

There are two ways you can read data from a `Buffer`.

1. Read data from the buffer into a channel.
2. Read data from the buffer yourself, using one of the get() methods.

Here is an example of how you can read data from a buffer into a channel:

```
//read from buffer into channel.
int bytesWritten = inChannel.write(buf);
```

Here is an example that reads data from a `Buffer` using the get() method:

```
byte aByte = buf.get();
```

There are many other versions of the `get()` method, allowing you to read data from the `Buffer` in many d ways. For instance, reading at specific positions, or reading an array of bytes from the buffer. See the JavaD concrete buffer implementation for more details.

rewind()

The `Buffer.rewind()` sets the `position` back to 0, so you can reread all the data in the buffer. The `li` remains untouched, thus still marking how many elements (bytes, chars etc.) that can be read from the `Buf`

clear() and compact()

Once you are done reading data out of the `Buffer` you have to make the `Buffer` ready for writing again. so either by calling `clear()` or by calling `compact()`.

If you call `clear()` the `position` is set back to 0 and the `limit` to `capacity`. In other words, the `Buff` cleared. The data in the `Buffer` is not cleared. Only the markers telling where you can write data into the `B`

If there is any unread data in the `Buffer` when you call `clear()` that data will be "forgotten", meaning you have any markers telling what data has been read, and what has not been read.

If there is still unread data in the `Buffer`, and you want to read it later, but you need to do some writing first `compact()` instead of `clear()`.

`compact()` copies all unread data to the beginning of the `Buffer`. Then it sets `position` to right after th unread element. The `limit` property is still set to `capacity`, just like `clear()` does. Now the `Buffer` is writing, but you will not overwrite the unread data.

mark() and reset()

You can mark a given position in a `Buffer` by calling the `Buffer.mark()` method. You can then later re position back to the marked position by calling the `Buffer.reset()` method. Here is an example:

```
buffer.mark();

//call buffer.get() a couple of times, e.g. during parsing.

buffer.reset();  //set position back to mark.
```

equals() and compareTo()

It is possible to compare two buffers using `equals()` and `compareTo()`.

equals()

Two buffers are equal if:

1. They are of the same type (byte, char, int etc.)
2. They have the same amount of remaining bytes, chars etc. in the buffer.
3. All remaining bytes, chars etc. are equal.

As you can see, equals only compares part of the `Buffer`, not every single element inside it. In fact, it just the remaining elements in the `Buffer`.

compareTo()

The `compareTo()` method compares the remaining elements (bytes, chars etc.) of the two buffers, for use sorting routines. A buffer is considered "smaller" than another buffer if:

1. The first element which is equal to the corresponding element in the other buffer, is smaller than tha other buffer.
2. All elements are equal, but the first buffer runs out of elements before the second buffer does (it has

elements).

Next: Java NIO Scatter / Gather

Connect with me:

Newsletter - Get all my free tips!

First Name * Last Name * Email *

Yes, give me tips!