Java Magazine

Coding

Modern file input/output with Java: Going fast with NIO and NIO.2

Ben Evans | January 7, 2022

Reach for these low-level Java APIs when you need to move a lot of file data or socket data quickly.

This article is about achieving high performance in terms of file input/output. High performance means not only that the I/O operation is executed quickly but that it also consumes (or ties up) minimal resources in the JVM and elsewhere.

This is the third article in a series. The first, "Modern file input/output with Java Path API and Files helper methods," introduced Java's `Path` API as well as an older API.

The second article, "Modern file input/output with Java: Let's get practical," showed how the `Path` API (also known as NIO.2) handles file system–specific extensions, including how to access features such as file attributes and symbolic links.

This article delves into more-advanced topics centered around performance. This has been a long-standing area of interest in the Java platform, as the earliest versions of Java provided only limited I/O capabilities.

Introducing NIO and buffers

The best place to begin is with the first API added to address the performance of Java file input/output. New Input/Output (NIO) was added to Java 1.4 as JSR 51. Lack of nonblocking communications and other I/O features had been a major criticism of early versions of Java, and the arrival of NIO brought a broad and useful feature set, including

- An abstraction layer for I/O operations that did not require traversing the Java heap
- The ability to encode and decode character sets
- An interface that could map files to data held in memory
- The capability to perform nonblocking I/O
- A new regular expression library

These capabilities were particularly important as Java grew into an attractive language to use for server-side development. More recent versions have continued this trend and have added other performance-oriented APIs, such as asynchronous I/O. Those are discussed shortly.

First, though, consider the NIO `Buffer` and `Channel` classes. They each provide a class that acts as a container for a linear sequence of elements of a specific (primitive) type. For simplicity, the following examples use `ByteBuffer` (a subclass of `Buffer`).

Note that NIO provides a *low-level abstraction* for high-performance I/O, but those APIs are not always as easy for developers to use as the more highly abstracted APIs covered in the previous articles. You should use the low-level APIs only if you have a specific performance need.

Byte buffers can be backed by either an on-heap Java `byte[]` array or an area of memory that is outside the Java heap (but is still within the C heap of the JVM process). Of these two approaches, the second is much more common.

- In the first case, the on-heap array buffer essentially provides a more object-oriented view of an underlying `byte[]`.
- The second case is called the *direct buffers* approach, and this approach bypasses the Java heap as much as possible. This can have performance benefits, for example, when you are doing a lot of copying to and from other off-heap data. In practice, direct buffers are far more commonly used than the on-heap array buffers.

`ByteBuffer` provides three static methods for creating a buffer.

- `allocateDirect()` for executing native I/O operations
- `allocate()` for creating a new heap-allocated buffer
- `wrap()` for setting up a buffer backed by a byte array that already exists

You can see how this works in the following code:

```
ByteBuffer b =
   ByteBuffer.allocateDirect(128 * 1024 * 1024);
ByteBuffer b2 =
   ByteBuffer.allocate(1024 * 1024);
```

```
byte[] data = {1, 2, 3, 4, 5};
ByteBuffer b3 = ByteBuffer.wrap(data);
```

Byte buffers are all about low-level access to the bytes, which means that you have to deal with the details manually. This includes the need to handle the *endianness* of the data format and the fact that Java numeric primitives (including bytes) are signed.

The API for the low-level details of the buffer is quite simple.

```
b.order(ByteOrder.LITTLE_ENDIAN);

int capacity = b.capacity();
int position = b.position();
int limit = b.limit();
int remaining = b.remaining();
boolean more = b.hasRemaining();
```

As you can see, when you use the `ByteBuffer` API, you must deal with very low-level aspects, such as the capacity of the buffer and your current position within it.

You can read and write data to the buffer in one of two ways: either one value at a time or as a bulk operation. It is by using the bulk operations that you can expect to realize performance gains from these low-level functions.

Here's the API for handling single values.

```
b.put((byte)101);
b.putChar('a');
b.putInt(0xcafebabe);

double d = b.getDouble();
```

The single-value API also contains an overload that is used for absolute positioning within the buffer.

```
b.put(0, (byte)9);
```

The bulk API works with either a `byte[]` or a `ByteBuffer` and operates on a potentially large number of values as a single operation, like this.

```
b.put(data);
b.put(b2);

b.get(data, 0, data.length);
```

Introducing channels

Buffers are an in-memory abstraction of whether you're working with on-heap or off-heap operations. In the case of off-heap operations, you should be moving data from the buffer to somewhere else that is also off-heap without having to transit the data through the Java heap.

For example, you may want to read and write data from the buffer to or from a file or socket directly. This is achieved by having a second object, a `Channel`, from the package `java.nio.channels`. A channel object represents an entity that can support read or write operations. Files and sockets are the usual examples, but you can imagine other, custom implementations used for special purposes.

Channel objects are created in the *open* state and can subsequently be closed. However, once closed, they cannot be reopened. Channels are conceived of as a one-way flow to or from a buffer. This means that they are either readable or writable—but not both.

The key to understanding channels is that

- Reading from a channel puts bytes into a buffer
- Writing to a channel takes bytes from a buffer

Accordingly, the key methods on channels are

- `read()` reads from the channel into the buffer (for readable channels).
- `write()` writes to the channel from the buffer (for writable channels).

These methods are frequently combined with the `compact()` method on a buffer object. The `compact()` method discards all data before the current position in the buffer and copies all later data to the beginning of the buffer. The method also skips the position of the cursor to after the bytes that have been copied. This allows the compact operation to immediately be followed by, say, another `read()`.

This is probably easiest to understand by seeing it in action. For example, suppose you have a large file that you want to checksum in 16-MB chunks.

```
FileInputStream fis = getSomeStream();
boolean fileOK = true;

try (FileChannel fchan = fis.getChannel()) {
   var buffy =
     ByteBuffer.allocateDirect(16 * 1024 * 1024);

   while(fchan.read(buffy) !=
      -1 || buffy.position() > 0 || fileOK) {
      fileOK = computeChecksum(buffy);
      buffy.compact();
      }
   }
   catch (IOException e) {
   System.out.println("Exception in I/O");
}
```

The code above will use native I/O as much as possible and will avoid a lot of copying of bytes on and off the Java heap. If the `computeChecksum()` method you're using has been well-implemented, this could be a very performant implementation.

Memory-mapped buffers

Mapped byte buffers

Mapped byte buffers are a type of direct byte buffer that contain a memory-mapped file (or a region of a file). These buffers are created from a `FileChannel` object but here's an important note: The `File` object corresponding to the `MappedByteBuffer` must not be used after the memory-mapped operations or an exception will be thrown. To mitigate this, you can use `try` to scope the objects tightly, as follows:

```
try (var raf = new RandomAccessFile(new
     File("input.txt"), "rw");
     var fc = raf.getChannel()) {
  MappedByteBuffer mbf =
    fc.map(FileChannel.MapMode.READ_WRITE, 0,
    fc.size());
  byte[] b = new byte[(int)fc.size()];
  mbf.get(b, 0, b.length);

  // Zero the in-memory copy
  for (int i = 0; i < fc.size(); i = i + 1) {
    b[i] = 0;
  }

  // Reposition to the start of the file
  mbf.position(0);

  // Zero the file
  mbf.put(b);
}
```

A mapped buffer extends `ByteBuffer` with additional operations that are specific to memory-mapped file regions, but it is otherwise a direct byte buffer. The point of a mapped file is that the contents can be changed *externally*. This means that the data in a mapped byte buffer can change at any time, such as if the content of the corresponding region of the mapped file is changed by another program.

Note that the precise semantics of file mapping are operating-system dependent, so the JDK is unable to make any guarantees about when any changes to the on-disk copy of the file will show up in the in-memory mapped copy.

SeekableByteChannel

Concurrent access to file I/O is handled by the interface `java.nio.channels.SeekableByteChannel`. The main implementation of this interface (`FileChannel`) can hold the current position of where you are reading from a file, as well as the position in the file that you are writing to. This means you can have multiple threads reading from and/or writing to the same channel at different positions, making for faster file I/O.

Creating seekable channels is very easy.

```
var readChannel =
   Files.newByteChannel(Path.of("temp.txt"),
```

```
    StandardOpenOption.READ);

var writeChannel =
    Files.newByteChannel(Path.of("temp.txt"),
    StandardOpenOption.WRITE);
```

The channels can now make use of the methods on `SeekableByteChannel`, including

- `position()`, which returns this channel's position
- `position(long newPosition)`, which sets this channel's position
- `read(ByteBuffer dst)`, which reads a sequence of bytes into the given buffer
- `size()`, which returns the current size of the entity this channel is connected to
- `write(ByteBuffer src)`, which writes bytes into the channel from the buffer

You could now pass `readChannel` to a reading thread and `writeChannel` to a writing thread and these channels could be used independently—despite ultimately operating on the same file.

Despite everything you can achieve with buffers, there are still limitations on what can be done for large I/O operations. For example, allocation of buffers is limited to 2 GB, because the constructor parameter is an int. This means that manipulation of files larger than this must be done piecemeal.

That's not a hypothetical. Imagine tasks, such as transferring 10 GB of data between file systems, that perform badly when done synchronously on a single thread.

Before the arrival of NIO.2 with Java 7, handling these types of operations would typically be done by writing custom multithreaded code and managing a separate thread pool for performing a background copy. This type of low-level coding can be error-prone and, frankly, it's the type of code that should be delegated to a framework capability.

Asynchronous operations with NIO.2

NIO.2 added asynchronous capabilities for both socket-based and file-based I/O, allowing you to take full advantage of your hardware's capabilities. This is an important feature for any language that wishes to remain relevant in the server-side and systems programming spaces.

So, what is meant by the term *asynchronous I/O*? Asynchronous I/O is simply a type of input/output processing that allows other activity to take place *before* the reading and writing has finished.

The NIO.2 API provides many new asynchronous channels that you can work with.

- `AsynchronousFileChannel` for file-based I/O
- `AsynchronousSocketChannel` for socket-based I/O; this supports timeouts
- `AsynchronousServerSocketChannel` for asynchronous sockets accepting connections
- `AsynchronousDatagramChannel` for "fire and forget" I/O; it does not check for a valid connection

There are two main styles that you can adopt when using the NIO.2 asynchronous I/O APIs: the *Future*

style and the *Callback* style.

Future style. The Future style uses a `Future` object from `java.util.concurrent`. The `Future` object represents the result of your asynchronous operation and is either still pending or will be fully materialized once the operation has been completed.

Typically you would use the `get()` method (with or without a timeout) to retrieve the results when the asynchronous I/O activity has been completed. The following example reads 100 bytes from a file and then gets the result (which will be the number of bytes actually read).

```
var file = Path.of("/usr/ben/foobar.txt");

try (var channel =
  AsynchronousFileChannel.open(file)) {
    var buffer = ByteBuffer.allocate(100);
    Future<Integer> result = channel.read(buffer, 0);

    BusinessProcess.doSomethingElse();

    var bytesRead = result.get();
    System.out.println("Bytes read [" + bytesRead + "]");
  } catch (IOException | ExecutionException |
    InterruptedException e) {
    e.printStackTrace();
}
```

`Future` objects are relatively simple to think about, because they have a `get()` method that will return the result of the operation—and will block if the operation has not completed yet. These objects also have the nonblocking `isDone()` method that tells you if the operation has completed yet. This means that you can avoid blocking indefinitely by simply regularly checking `isDone()` and then only call `get()` after the operation has completed.

Callback style. In case the Future style seems counterintuitive to you, there is an alternative technique, called Callback style, which uses the `CompletionHandler` interface. Some developers are more comfortable using the Callback style because it is similar to event-handling code.

The `java.nio.channels.CompletionHandler<V, A>` interface (where `V` is the result type and `A` is the attached object you are getting the result from) has two methods that must be given implementations. This means, of course, that you can't use a lambda expression to represent one. Instead, anonymous inner classes are typically used.

These methods, `completed(V, A)` and `failed(V, A)`, must be given an implementation that describes how the program should behave when the asynchronous I/O operation has been completed successfully or has failed for some reason. One (and only one) of these two methods will be invoked when the asynchronous I/O activity has been completed.

The following is the same task as before, reading 100 bytes from a file, but this time using the `CompletionHandler<Integer, ByteBuffer>` interface:

```
var file = Path.of("/usr/ben/foobar.txt");
```

```
try (var channel = AsynchronousFileChannel.open(file)) {
    var buffer = ByteBuffer.allocate(100);
    var handler = new CompletionHandler<Integer,
        ByteBuffer>() {
        public void completed(Integer result,
            ByteBuffer attachment) {
            System.out.println("Bytes read [" + result + "]");
        }

        public void failed(Throwable exception,
            ByteBuffer attachment) {
            exception.printStackTrace();
        }
   };

   channel.read(buffer, 0, buffer, handler);
} catch (IOException e) {
    e.printStackTrace();
}
```

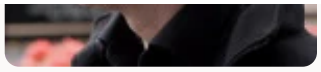
The examples above were all file-based, but it is possible to do very similar tasks using the socket APIs as well.

Conclusion

This article showed some of the high-performance capabilities of the Java NIO and NIO.2 APIs. These APIs should not be the first choice for general I/O operations due to their lower-level abstractions, which makes them more complicated to use. However, they are a powerful tool that can be used to produce fast results when I/O performance is critical to your application.

Dig deeper

- Modern file input/output with Java Path API and Files helper methods
- Modern file input/output with Java: Let's get practical
- Working and unit testing with temporary files in Java
- Quiz yourself: Use the Files class to copy or move files and directories



Ben Evans

Ben Evans (@kittylyst) is a Java Champion and Senior Principal Software Engineer at Red Hat. He has written five books on programming, including *Optimizing Java* (O'Reilly) and *The Well-Grounded Java Developer* (Manning). Previously he was Lead Architect for Instrumentation at New Relic, a founder of jClarity (acquired by Microsoft) and a member of the Java SE/EE Executive Committee.

‹ Previous Post

Next Post ›

Resources for

About
Careers
Developers
Investors
Partners
Startups

Why Oracle

Analyst Reports
Best CRM
Cloud Economics
Corporate Responsibility
Diversity and Inclusion
Security Practices

Learn

What is Customer Service?
What is ERP?
What is Marketing Automation?
What is Procurement?
What is Talent Management?
What is VM?

What's New

Try Oracle Cloud Free Tier
Oracle Sustainabillity
Oracle COVID-19 Response
Oracle and SailGP
Oracle and Premier League
Oracle and Red Bull Racing Honda

Contact Us

US Sales 1.800.633.0738
How can we help?
Subscribe to Oracle Content
Try Oracle Cloud Free Tier
Events
News

© 2022 Oracle | Site Map Privacy / Do Not Sell My Info Cookie Preferences Ad Choices Careers