Java ByteBuffer performance issue

While processing multiple gigabyte files I noticed something odd: it seems that reading from a file using a filechannel into a re-used ByteBuffer object allocated with allocateDirect is much slower than reading from a MappedByteBuffer, in fact it is even slower than reading into byte-arrays using regular read calls!

I was expecting it to be (almost) as fast as reading from mappedbytebuffers as my ByteBuffer is allocated with allocateDirect, hence the read should end-up directly in my bytebuffer without any intermediate copies.

My question now is: what is it that I'm doing wrong? Or is bytebuffer+filechannel really slowe r than regular io/mmap?

I the example code below I also added some code that converts what is read into long values, as that is what my real code constantly does. I would expect that the ByteBuffer getLong() method is much faster than my own byte shuffeler.

Test-results: mmap: 3.828 bytebuffer: 55.097 regular i/o: 38.175

```
        // mmap()
        MappedByteBuffer mbb = fileChannel.map(FileChannel.MapMode.READ_WRITE,
0, size);
        byte [] buffer1 = new byte[24];
        start = System.currentTimeMillis();
        for(int index=0; index<n; index++) {
                mbb.position(index * 24);
                mbb.get(buffer1, 0, 24);
                long dummy1 = byteArrayToLong(buffer1, 0);
                long dummy2 = byteArrayToLong(buffer1, 8);
                long dummy3 = byteArrayToLong(buffer1, 16);
        }
        System.out.println("mmap: " + (System.currentTimeMillis() - start) /
1000.0);

        // bytebuffer
        ByteBuffer buffer2 = ByteBuffer.allocateDirect(24);
        start = System.currentTimeMillis();
        for(int index=0; index<n; index++) {
            buffer2.rewind();
            fileChannel.read(buffer2, index * 24);
            buffer2.rewind();   // need to rewind it to be able to use it
            long dummy1 = buffer2.getLong();
            long dummy2 = buffer2.getLong();
            long dummy3 = buffer2.getLong();
        }
        System.out.println("bytebuffer: " + (System.currentTimeMillis() -
start) / 1000.0);

        // regular i/o
        byte [] buffer3 = new byte[24];
        start = System.currentTimeMillis();
        for(int index=0; index<n; index++) {
```

As loading large sections and then processing is them is not an option (I'll be reading data all over the place) I think I should stick to a MappedByteBuffer. Thank you all for your suggestions.

java performance nio bytebuffer

edited Jan 27 '12 at 2:32
Jonas
24.4k 57 170 270

asked Oct 12 '11 at 10:35
Folkert van Heusden
620 1 11 23

add a comment

4 Answers

I believe you are just doing micro-optimization, which **might just *not* matter** (www.codinghorror.com).

Below is a version with a larger buffer and redundant `seek` / `setPosition` calls removed.

- When I enable "native byte ordering" (which is actually unsafe if the machine uses a different 'endian' convention):

```
mmap: 1.358
bytebuffer: 0.922
regular i/o: 1.387
```

- When I comment out the order statement and use the default big-endian ordering:

```
mmap: 1.336
bytebuffer: 1.62
regular i/o: 1.467
```

- Your original code:

```
mmap: 3.262
bytebuffer: 106.676
regular i/o: 90.903
```

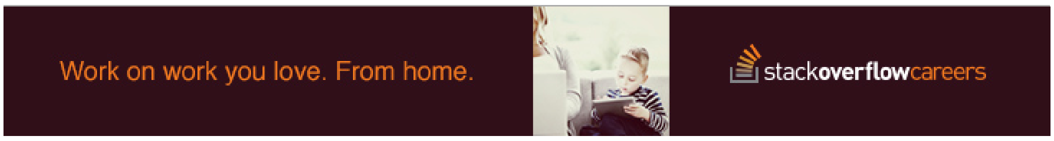
Here's the code:

```
    if (testFile.exists() && testFile.length() >= DATASIZE) {
      System.out.println("File exists");
    } else {
      testFile.delete();
      System.out.println("Preparing file");
      byte [] buffer = new byte[BUFFSIZE];
      pos = 0;
      nDone = 0;
      while (pos < DATASIZE) {
        fileHandle.write(buffer);
        pos += buffer.length;
      }

      System.out.println("File prepared");
    }
    fileChannel = fileHandle.getChannel();

    // mmap()
    MappedByteBuffer mbb = fileChannel.map(FileChannel.MapMode.READ_WRITE,
0, DATASIZE);
    byte [] buffer1 = new byte[BUFFSIZE];
    mbb.position(0);
    start = System.currentTimeMillis();
    pos = 0;
    while (pos < DATASIZE) {
      mbb.get(buffer1, 0, BUFFSIZE);
      // This assumes BUFFSIZE is a multiple of 8.
      for (int i = 0; i < BUFFSIZE; i += 8) {
        long dummy = byteArrayToLong(buffer1, i);
      }
      pos += BUFFSIZE;
    }
    System.out.println("mmap: " + (System.currentTimeMillis() - start) /
```

answered Oct 12 '11 at 12:56
JBert
1,630 6 20

That would indeed by faster. Did not expect it to be that much faster so thanks! – Folkert van Heusden Oct 12 '11 at 14:09

add a comment



When you have a loop which iterates more than 10,000 times it can trigger the whole method to be compiled to native code. However, your later loops have not been run and cannot be optimised to the same degree. To avoid this issue, place each loop in a different method and run again.

Additionally, you may want to set the Order for the ByteBuffer to be order(ByteOrder.nativeOrder()) to avoid all the bytes swapping around when you do a `getLong` and read more than 24 bytes at a time. (As reading very small portions generates much more system calls) Try reading 32*1024 bytes at a time.

I wound also try `getLong` on the MappedByteBuffer with native byte order. This is likely to be the fastest.

edited Oct 12 '11 at 13:21
Anthony Accioly
11.5k 2 26 56

answered Oct 12 '11 at 10:47
Peter Lawrey
251k 24 242 471

Moving the code into seperate methods did not make any difference. Using also getLong in the mappedbytebuffer indeed made it even faster. But still I wonder why the second test ("read a bytebuffer from a filechannel") is so slow.\ – Folkert van Heusden Oct 12 '11 at 12:07

1 You are performing one system call for every 24 bytes. In the first example, you are performing only one or two system calls total. – Peter Lawrey Oct 12 '11 at 13:27

add a comment

A `MappedByteBuffer` will always be the fastest, because the operating system associates the OSlevel disk buffer with your process memory space. Reading into an allocated direct buffer, by comparison, first loads the block into the OS buffer, then copies the contents of the OS buffer into the allocated in-process buffer.

Your test code also does lots of very small (24 byte) reads. If your actual application does the same, then you'll get an even bigger performance boost from mapping the file, because each of the reads is a separate kernel call. You should see several times the performance by mapping.

As for the direct buffer being slower than the java.io reads: you don't give any numbers, but I'd expect a slight degredation because the `getLong()` calls need to cross the JNI boundary.

answered Oct 12 '11 at 11:57
kdgregory
23.2k 4 47 73

3 From what I read (in a book about NIO from o'reilly), a read to a properly allocated bytebuffer should also be direct without any copies. Unfortunately mapping the input-file to memory won't work in the real app as this can be terabytes in size. The numbers were at the bottom of my mail: mmap: 3.828 seconds bytebuffer: 55.097 seconds regular i/o: 38.175 seconds. – Folkert van Heusden Oct 12 '11 at 12:11

@Folkert - either the author of that book was wrong, or you are misinterpreting what he/she said. Disk controllers deal with large block sizes, and the OS needs a place to buffer that data and carve out the piece that you need. – kdgregory Oct 12 '11 at 12:23

1 But the real problem is that each of your reads -- in either NIO or IO -- is a separate system call, while the mapped file is a direct memory access (with a possible page fault). If your real application has a large proportion of localized reads, you will probably benefit from a buffer cache (which can be memory-mapped or on-heap). If you're jumping all over a terabyte-scale file, then the disk IO will become the limiting factor and even memory-mapping won't help. – kdgregory Oct 12 '11 at 12:26

So direct buffer and mapped memory does the same thing(avoid memory copy), beside direct buffer may cause a lot of system call ? right ? – MrROY Jan 4 '14 at 4:44

@MrROY - I can't understand what you're asking. But no, a direct ByteBuffer shouldn't cause a lot of system calls. RandomAccessFile might. – kdgregory Jan 11 '14 at 12:00

show **1** more comment

Reading into the direct byte buffer is faster, but getting the data out of it into the JVM is slower. Direct byte buffer is intended for cases where you're just copying the data without actually looking at it in the Java code. Then it doesn't have to cross the native->JVM boundary at all, so it's quicker than using e.g. a byte[] array or a normal ByteBuffer, where the data would have to cross that boundary twice in the copy process.

answered Oct 12 '11 at 23:54
EJP
129k 11 81 145

add a comment

Not the answer you're looking for? Browse other questions tagged java performance nio bytebuffer or ask your own question.