Java NIO写大文件比较 原

优雅先生 发布于 2014/03/25 13:07 字数 1667 阅读 18353 收藏 37 点赞 7 评论 8

测试说明

写2G文件，分批次写入，每批次写入128MB；

分别在Win7系统（3G内存，双核，32位，T系列处理器）和MacOS系统（8G内存，四核，64位，i7系列处理器）下运行测试。理论上跟硬盘类型和配置也有关系，这里不再贴出了。

测试代码

```
import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.lang.reflect.Method;
import java.nio.ByteBuffer;
import java.nio.MappedByteBuffer;
import java.nio.channels.Channels;
import  nels.FileChannel;
import  nels.FileChannel.MapMode;
import  nels.ReadableByteChannel;
import  .AccessController;
import  .PrivilegedAction;

import  h;

/**
 * NIO写
 * @autl
 *
 */
public  gFileComparison {

        k be written per time
        ic final int DATA_CHUNK = 128 * 1024 * 1024;

        // total data size is 2G
        private static final long LEN = 2L * 1024 * 1024 * 1024L;


        public static void writeWithFileChannel() throws IOException {
                File file = new File("e:/test/fc.dat");
                if (file.exists()) {
                        file.delete();
                }

                RandomAccessFile raf = new RandomAccessFile(file, "rw");
                FileChannel fileChannel = raf.getChannel();

                byte[] data = null;
                long len = LEN;
                ByteBuffer buf = ByteBuffer.allocate(DATA_CHUNK);
                int dataChunk = DATA_CHUNK / (1024 * 1024);
                while (len >= DATA_CHUNK) {
                        System.out.println("write a data chunk: " + dataChunk + "MB");

                        buf.clear(); // clear for re-write
                        data = new byte[DATA_CHUNK];
                        for (int i = 0; i < DATA_CHUNK; i++) {
                                buf.put(data[i]);
                        }
```

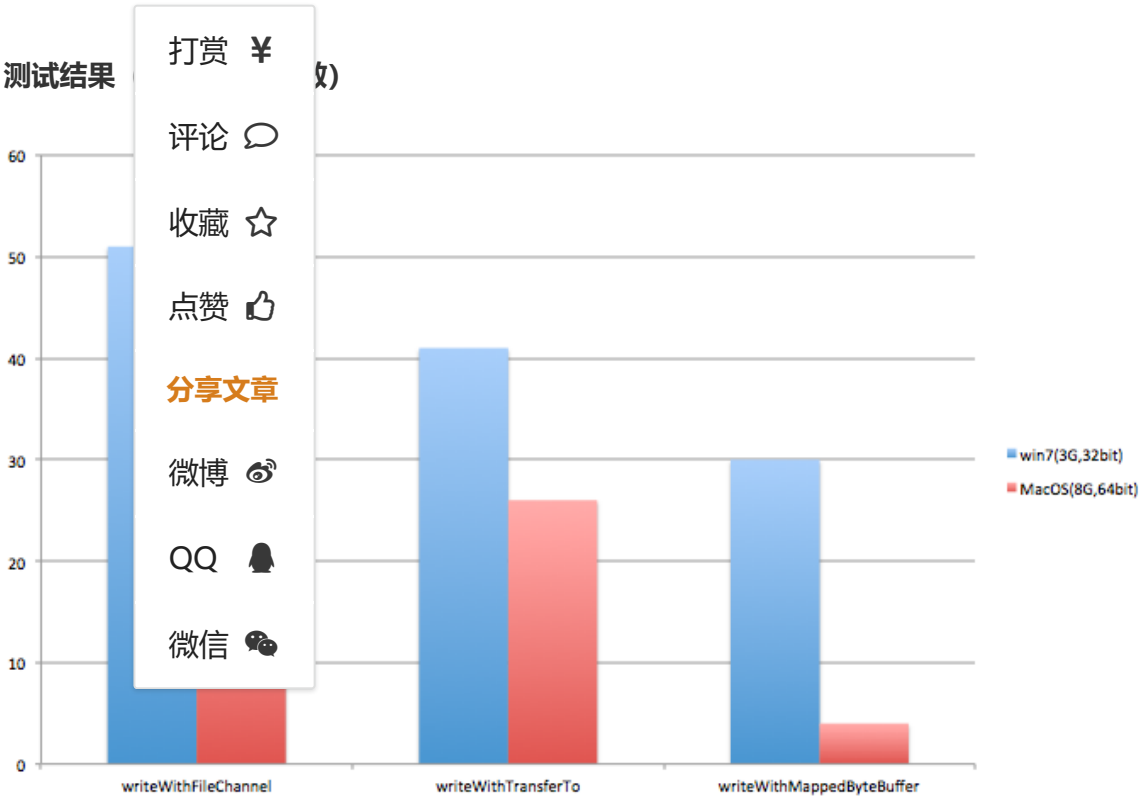
```
                buf.flip(); // switches a Buffer from writing mode to reading mode
                fileChannel.write(buf);
                fileChannel.force(true);

                len -= DATA_CHUNK;
        }

        if (len > 0) {
                System.out.println("write rest data chunk: " + len + "B");
                buf = ByteBuffer.allocateDirect((int) len);
                data = new byte[(int) len];
                for (int i = 0; i < len; i++) {
                        buf.put(data[i]);
                }

                buf.flip(); // switches a Buffer from writing mode to reading mode, positic
                fileChannel.write(buf);
                fileChannel.force(true);
                data = null;


Channel.close();
close();



 * write big file with MappedByteBuffer
 * @throws IOException
 */
public static void writeWithMappedByteBuffer() throws IOException {
        File file = new File("e:/test/mb.dat");
        if (file.exists()) {
                file.delete();
        }

        RandomAccessFile raf = new RandomAccessFile(file, "rw");
        FileChannel fileChannel = raf.getChannel();
        int pos = 0;
        MappedByteBuffer mbb = null;
        byte[] data = null;
        long len = LEN;
        int dataChunk = DATA_CHUNK / (1024 * 1024);
        while (len >= DATA_CHUNK) {
                System.out.println("write a data chunk: " + dataChunk + "MB");

                mbb = fileChannel.map(MapMode.READ_WRITE, pos, DATA_CHUNK);
                data = new byte[DATA_CHUNK];
                mbb.put(data);

                data = null;

                len -= DATA_CHUNK;
                pos += DATA_CHUNK;
```

```
if (len > 0) {
        System.out.println("write rest data chunk: " + len + "B");

        mbb = fileChannel.map(MapMode.READ_WRITE, pos, len);
        data = new byte[(int) len];
        mbb.put(data);
}

data = null;
p(mbb);   // release MappedByteBuffer
Channel.close();


c void writeWithTransferTo() throws IOException {
 file = new File("e:/test/transfer.dat");
file.exists()) {
    file.delete();


omAccessFile raf = new RandomAccessFile(file, "rw");
Channel toFileChannel = raf.getChannel();

 len = LEN;
[] data = null;
ArrayInputStream bais = null;
ableByteChannel fromByteChannel = null;
long position = 0;
int dataChunk = DATA_CHUNK / (1024 * 1024);
while (len >= DATA_CHUNK) {
        System.out.println("write a data chunk: " + dataChunk + "MB");

        data = new byte[DATA_CHUNK];
        bais = new ByteArrayInputStream(data);
        fromByteChannel = Channels.newChannel(bais);

        long count = DATA_CHUNK;
        toFileChannel.transferFrom(fromByteChannel, position, count);

        data = null;
        position += DATA_CHUNK;
        len -= DATA_CHUNK;
}

if (len > 0) {
        System.out.println("write rest data chunk: " + len + "B");

        data = new byte[(int) len];
        bais = new ByteArrayInputStream(data);
        fromByteChannel = Channels.newChannel(bais);

        long count = len;
        toFileChannel.transferFrom(fromByteChannel, position, count);
}
```

```
        toFileChannel.close();
        fromByteChannel.close();
}

/**
 * 在MappedByteBuffer释放后再对它进行读操作的话就会引发jvm crash，在并发情况下很容易发生
 * 正在释放时另一个线程正开始读取，于是crash就发生了。所以为了系统稳定性释放前一般需要检
 * 查是否还有线程在读或写
 * @param mappedByteBuffer
 */
public static void unmap(final MappedByteBuffer mappedByteBuffer) {
        try
        {
            if (mappedByteBuffer == null) {
                    return;
            }

            mappedByteBuffer.force();
            AccessController.doPrivileged(new PrivilegedAction<Object>() {
                    @Override
                    @SuppressWarnings("restriction")
                    public Object run() {
                            try {
                                    Method getCleanerMethod = mappedByteBuffer.getClass
                                                    .getMethod("cleaner", new Class[0])
                                    getCleanerMethod.setAccessible(true);
                                    sun.misc.Cleaner cleaner =
                                                    (sun.misc.Cleaner) getCleanerMethod
                                                            .invoke(mappedByteBuffer, r
                                    cleaner.clean();

                            } catch (Exception e) {
                                    e.printStackTrace();
                            }
                            System.out.println("clean MappedByteBuffer completed");
                            return null;
                    }
            });

        } catch (Exception e) {
                e.printStackTrace();
        }
}

public static void main(String[] args) throws IOException {
        StopWatch sw = new StopWatch();

        sw.startWithTaskName("write with file channel's write(ByteBuffer)");
        writeWithFileChannel();
        sw.stopAndPrint();

        sw.startWithTaskName("write with file channel's transferTo");
        writeWithTransferTo();
        sw.stopAndPrint();
```

```
                writeWithMappedByteBuffer();
                sw.stopAndPrint();
        }

}
```

测试结果



1. 显然writeWithMappedByteBuffer方式性能最好，且在硬件配置较高情况下优势越加明显
2. 在硬件配置较低情况下，writeWithTransferTo比writeWithFileChannel性能稍好
3. 在硬件配置较高情况下，writeWithTransferTo和writeWithFileChannel的性能基本持平
4. 此外，注意writeWithMappedByteBuffer方式除了占用JVM堆内存外，还要占用额外的native内存（Direct Byte Buffer内存）

内存映射文件使用经验

1. MappedByteBuffer需要占用"双倍"的内存（对象JVM堆内存和Direct Byte Buffer内存），可以通过-XX:MaxDirectMemorySize参数设置后者最大大小
2. 不要频繁调用MappedByteBuffer的force()方法，因为这个方法会强制OS刷新内存中的数据到磁盘，从而只能获得些微的性能提升（相比IO方式），可以用后面的代码实例进行定时、定量刷新

数据。为了降低这种风险，避免用MappedByteBuffer写超大文件，可以把大文件分割成几个小文件，但不能太小（否则将失去性能优势）

4. ByteBuffer的rewind()方法将position属性设回为0，因此可以重新读取buffer中的数据；limit属性保持不变，因此可读取的字节数不变
5. ByteBuffer的flip()方法将一个Buffer由写模式切换到读模式
6. ByteBu 和compact()可以在我们读完ByteBuffer中的数据后重新切回写模式。不同的是clear() 设置为0，limit设为capacity，换句话说Buffer被清空了，但Buffer内的数据并没有被清 fer中还有未被读取的数据，那调用clear()之后，这些数据会被"遗忘"，再写入就会覆盖 据。而调用compcat()之后，这些未被读取的数据仍然可以保留，因为它将所有还未被读 到Buffer的左端，然后设置position为紧随未读数据之后，limit被设置为capaci 不会被覆盖

定时、定量 文件到磁盘

```
import 
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;

public class MappedFile {

        // 文件名
        private String fileName;

        // 文件所在目录路径
        private String fileDirPath;

        // 文件对象
        private File file;

        private MappedByteBuffer mappedByteBuffer;
        private FileChannel fileChannel;
        private boolean boundSuccess = false;

        // 文件最大只能为50MB
        private final static long MAX_FILE_SIZE = 1024 * 1024 * 50;

        // 最大的脏数据量512KB,系统必须触发一次强制刷
        private long MAX_FLUSH_DATA_SIZE = 1024 * 512;

        // 最大的刷间隔,系统必须触发一次强制刷
```

```
// 文件写入位置
private long writePosition = 0;

// 最后一次刷数据的时候
private long lastFlushTime;

// 上一次刷的文件位置
private long lastFlushFilePosition = 0;

dFile(String fileName, String fileDirPath) {
r();
.fileName = fileName;
.fileDirPath = fileDirPath;
.file = new File(fileDirPath + "/" + fileName);
!file.exists()) {
    try {
            file.createNewFile();
    } catch (IOException e) {
            e.printStackTrace();
    }
```



```
 * 内存映射文件绑定
 * @return
 */
public synchronized boolean boundChannelToByteBuffer() {
        try {
                RandomAccessFile raf = new RandomAccessFile(file, "rw");
                this.fileChannel = raf.getChannel();
        } catch (Exception e) {
                e.printStackTrace();
                this.boundSuccess = false;
                return false;
        }

        try {
                this.mappedByteBuffer = this.fileChannel
                                .map(FileChannel.MapMode.READ_WRITE, 0, MAX_FILE_SIZE);
        } catch (IOException e) {
                e.printStackTrace();
                this.boundSuccess = false;
                return false;
        }

        this.boundSuccess = true;
        return true;
}

/**
```

```
     * @return
     */
    public synchronized boolean writeData(byte[] data) {

            return false;
    }

    /**
     * 在文件末尾追加数据
     * @param data
     *
     * @exception
     */
    public synchronized boolean appendData(byte[] data) throws Exception {
        if (!boundSuccess) {
            boundChannelToByteBuffer();
        }

        long writePosition = writePosition + data.length;
        if (writePosition >= MAX_FILE_SIZE) {   // 如果写入data会超出文件大小限制，不写入
            flush();
            writePosition = writePosition - data.length;
            System.out.println("File="
                                                    + file.toURI().toString()
                                                    + " is written full.");
            System.out.println("already write data length:"
                                                    + writePosition
                                                    + ", max file size=" + MAX_FILE_SIZ
            return false;
        }

        this.mappedByteBuffer.put(data);

        // 检查是否需要把内存缓冲刷到磁盘
        if ( (writePosition - lastFlushFilePosition > this.MAX_FLUSH_DATA_SIZE)
                ||
                (System.currentTimeMillis() - lastFlushTime > this.MAX_FLUSH_TIME_GAP
                 && writePosition > lastFlushFilePosition) ) {
            flush();   // 刷到磁盘
        }

        return true;
    }

    public synchronized void flush() {
        this.mappedByteBuffer.force();
        this.lastFlushTime = System.currentTimeMillis();
        this.lastFlushFilePosition = writePosition;
    }

    public long getLastFlushTime() {
        return lastFlushTime;
    }
```

```
        return fileName;
    }

    public String getFileDirPath() {
        return fileDirPath;
    }

    public boolean isBundSuccess() {
        return boundSuccess;
    }

    getFile() {
    rn file;

    c long getMaxFileSize() {
    rn MAX_FILE_SIZE;

    getWritePosition() {
    rn writePosition;

    getLastFlushFilePosition() {
    rn lastFlushFilePosition;

    public long getMAX_FLUSH_DATA_SIZE() {
        return MAX_FLUSH_DATA_SIZE;
    }

    public long getMAX_FLUSH_TIME_GAP() {
        return MAX_FLUSH_TIME_GAP;
    }

}
```