Java Nio Large File Transfer Tutorial

examples.javacodegeeks.com/core-java/nio/java-nio-large-file-transfer-tutorial/

This article is a tutorial on transferring a large file using Java Nio. It will take shape via two examples demonstrating a simple local file transfer from one location on hard disk to another and then via sockets from one remote location to another remote location.

Table Of Contents

1. Introduction

This tutorial will make use of the FileChannel abstraction for both remote and local copy. Augmenting the remote copy process will be a simple set of abstractions (ServerSocketChannel & SocketChannel) that facilitate the transfer of bytes over the wire. Finally we wrap things up with an asynchronous implementation of large file transfer. The tutorial will be driven by unit tests that can run from command line using maven or from within your IDE.

2. Technologies used

The example code in this article was built and run using:

- Java 1.8.101 (1.8.x will do fine)
- Maven 3.3.9 (3.3.x will do fine)
- Spring source tool suite 4.6.3 (Any Java IDE would work)
- Ubuntu 16.04 (Windows, Mac or Linux will do fine)

2. FileChannel

A FileChannel is a type of Channel used for writing, reading, mapping and manipulating a File. In addition to the familiar Channel (read, write and close) operations, this Channel has a few specific operations:

- Has the concept of an absolute position in the File which does not affect the Channels current position.
- Parts or regions of a File can be mapped directly into memory and work from memory, very useful when dealing with large files.
- Writes can be forced to the underlying storage device, ensuring write persistence.

- Bytes can be transferred from one ReadableByteChannel / WritableByteChannel instance to another ReadableByteChannel / WritableByteChannel, which FileChannel implements. This yields tremendous IO performance advantages that some Operating systems are optimized for.
- A part or region of a File may be locked by a process to guard against access by other processes.

FileChannels are thread safe. Only one IO operation that involves the FileChannels position can be in flight at any given point in time, blocking others. The view or snapshot of a File via a FileChannel is consistent with other views of the same File within the same process. However, the same cannot be said for other processes. A file channel can be created in the following ways:

- … `FileChannel.open(...)`
- … `FileInputStream(...).getChannel()`
- … `FileOutputStream(...).getChannel()`
- … `RandomAccessFile(...).getChannel()`

Using one of the stream interfaces to obtain a FileChannel will yield a Channel that allows either read, write or append privileges and this is directly attributed to the type of Stream (FileInputStream or FileOutputStream) that was used to get the Channel. Append mode is a configuration artifact of a FileOutputStream constructor.

4. Background

The sample program for this example will demonstrate the following:

1. Local transfer of a file (same machine)
2. Remote transfer of a file (potentially remote different processes, although in the unit tests we spin up different threads for client and server)
3. Remote transfer of a file asynchronously

Particularly with large files the advantages of asynchronous non blocking handling of file transfer cannot be stressed enough. Large files tying up connection handling threads soon starve a server of resources to handle additional requests possibly for more large file transfers.

5. Program

The code sample can be split into local and remote domains and within remote we further specialize an asynchronous implementation of file transfer, at least on the receipt side which is arguably the more interesting part.

5.1. Local copy

FileCopy

```
01  final class FileCopy

02

03              FileCop()
        private {
```

```
04          throw new IllegalStateException(Constants.INSTANTIATION_NOT_ALLOWED);
05      }
06
07                                      String           String
        public static void copy(final src,        final target)       throws
    IOException
    {
08            (StringUtils.isEmpty(src) || StringUtils.isEmpty(target))
            if {
09                                                "src and target
                throw new IllegalArgumentException(required"                );
10          }
11
12                String fileName =
            final getFileName(src);
13
14          try
    (FileChannel from = (FileChannel.open(Paths.get(src),
    StandardOpenOption.READ));
15                  FileChannel to = (FileChannel.open(Paths.get(target
                    +                                                  "/"
    + fileName), StandardOpenOption.CREATE_NEW, StandardOpenOption.WRITE)))
    {
16              transfer(from, to, 0l,
                from.size());
17          }
18      }
19
```

```
20                    String                    String src)
        private static getFileName(       final {
21          assert StringUtils.isNotEmpty(src);
22
23                File file
            final =           new File(src);
24              (file.isFile())
            if {
25                return file.getName();
26          }else {
27                                          "src is not a valid
                throw new RuntimeException(file"                    );
28          }
29      }
30
31                                        FileChannel           FileChannel
        private static void transfer(final from,            final to,            long
                              IOException
    position,long size)throws {
32                !Objects.isNull(from)  &&
            assert !Objects.isNull(to);
33
34                (position < size)
            while {
35
    position += from.transferTo(position, Constants.TRANSFER_MAX_SIZE,
    to);
36          }
```

```
37      }
38  }
```

- line 14: we `open` the `from` Channel with the `StandardOpenOption.READ` meaning that this Channel will only be read from. The path is provided.
- line 15: the `to` Channel is opened with the intention to write and create, the path is provided.
- line 31-37: the two Channels are provided (from & to) along with the `position` (initially where to start reading from) and the `size` indicating the amount of bytes to transfer in total. A loop is started where attempts are made to transfer up to `Constants.TRANSFER_MAX_SIZE` in bytes from the `from` Channel to the `to` Channel. After each iteration the amount of bytes transferred is added to the `position` which then advances the cursor for the next transfer attempt.

5.2. Remote copy

FileReader

```
01              FileReader
    final class {
02
03                  FileChannel
        private final channel;
04      private final FileSender sender;
05
06                      FileSender              String              IOException
        FileReader(final sender,            final path)       throws {
07            (Objects.isNull(sender) || StringUtils.isEmpty(path))
            if {
08                                              "sender and path
              throw new IllegalArgumentException(required"                  );
09          }
10
11              .sender =
            thissender;
```

```
12          this
    .channel = FileChannel.open(Paths.get(path),
    StandardOpenOption.READ);
13      }
14
15                          IOException
        void read()throws {
16          try {
17              transfer();
18          }finally {
19              close();
20          }
21      }
22
23                           IOException
        void close()throws {
24          this.sender.close();
25          this.channel.close();
26      }
27
28                                    IOException
        private void transfer()throws {
29                                     .channel,
            this.sender.transfer(this0l,          this.channel.size());
```

```
30      }
31  }
```

- line 12: the FileChannel is opened with the intent to read `StandardOpenOption.READ`, the `path` is provided to the File.
- line 15-21: we ensure we transfer the contents of the FileChannel entirely and the close the Channel.
- line 23-26: we close the `sender` resources and then close the FileChannel
- line 29: we call `transfer(...)` on the `sender` to transfer all the bytes from the FileChannel

FileSender

```
01              FileSender
    final class {
02
03      private final InetSocketAddress hostAddress;
04      private SocketChannel client;
05
06                                      IOException
        FileSender(final int port)throws {
07              .hostAddress
            this=               new InetSocketAddress(port);
08              .client =
            thisSocketChannel.open(          this.hostAddress);
09      }
10
11                          FileChannel
        void transfer(final channel,             long position,long size)throws
    IOException
    {
12          assert !Objects.isNull(channel);
```

```
13
14                (position < size)
            while {
15
    position += channel.transferTo(position,
    Constants.TRANSFER_MAX_SIZE,                                    this
    .client);
16          }
17      }
18
19      SocketChannel getChannel()
        {
20          return this.client;
21      }
22
23                          IOException
        void close()throws {
24          this.client.close();
25      }
26  }
```

line 11-17: we provide the FileChannel, `position` and `size` of the bytes to transfer from the given `channel`. A loop is started where attempts are made to transfer up to `Constants.TRANSFER_MAX_SIZE` in bytes from the provided Channel to the SocketChannel `client`. After each iteration the amount of bytes transferred is added to the `position` which then advances the cursor for the next transfer attempt.

FileReceiver

```
01              FileReceiver
    final class {
```

```
02
03      private final int port;
04                    FileWriter
        private final fileWriter;
05      private final long size;
06
07                                        FileWriter                    size)
        FileReceiver(final int port,final fileWriter,          final long {
08              .port =
            thisport;
09              .fileWriter =
            thisfileWriter;
10              .size =
            thissize;
11      }
12
13                            IOException
        void receive()throws {
14          SocketChannel channel
            =                       null;
15
16          try (final
    ServerSocketChannel serverSocketChannel = ServerSocketChannel.open())
    {
17              init(serverSocketChannel);
18
```

```
19              channel =
                serverSocketChannel.accept();
20
21              doTransfer(channel);
22          }finally {
23                (!Objects.isNull(channel))
                if {
24                  channel.close();
25              }
26
27              this.fileWriter.close();
28          }
29      }
30
31                                      SocketChannel                IOException
        private void doTransfer(final channel)                throws {
32          assert !Objects.isNull(channel);
33
34          this.fileWriter.transfer(channel,this.size);
35      }
36
```

```
37                              ServerSocketChannel
    private void init(final serverSocketChannel)                    throws
IOException
{
38          assert !Objects.isNull(serverSocketChannel);
39
40          serverSocketChannel.bind(new InetSocketAddress(this.port));
41      }
42  }
```

The `FileReceiver` is a mini server that listens for incoming connections on the `localhost` and upon connection, accepts it and initiates a transfer of bytes from the accepted Channel via the `FileWriter` abstraction to the encapsulated FileChannel within the `FileWriter`. The `FileReceiver` is only responsible for receiving the bytes via socket and then delegates transferring them to the `FileWriter`.

FileWriter

```
01              FileWriter
final class {
02
03                    FileChannel
        private final channel;
04
05                      String              IOException
        FileWriter(final path)        throws {
06             (StringUtils.isEmpty(path))
            if {
07               throw new IllegalArgumentException("path required");
08          }
09
```

```
10          this
    .channel = FileChannel.open(Paths.get(path), StandardOpenOption.WRITE,
    StandardOpenOption.CREATE_NEW);
11      }
12
13                          SocketChannel                                   IOException
        void transfer(final channel,              final long bytes)throws {
14          assert !Objects.isNull(channel);
15
16              position =
            long 0l;
17                (position < bytes)
            while {
18              position
                +=          this
    .channel.transferFrom(channel, position,
    Constants.TRANSFER_MAX_SIZE);
19          }
20      }
21
22                      ByteBuffer                              IOException
        int write(final buffer,           long position)throws {
23          assert !Objects.isNull(buffer);
24
25              bytesWritten
            int =             0;
26              (buffer.hasRemaining())
            while{
```

```
27              bytesWritten
                +=              this
    .channel.write(buffer, position +
    bytesWritten);
28          }
29
30          return bytesWritten;
31      }
32
33                          IOException
        void close()throws {
34          this.channel.close();
35      }
36  }
```

The `FileWriter` is simply charged with transferring the bytes from a SocketChannel to it's encapsulated FileChannel. As before, the transfer process is a loop which attempts to transfer up to `Constants.TRANSFER_MAX_SIZE` bytes with each iteration.

5.2.1. Asynchronous large file transfer

The following code snippets demonstrate transferring a large file from one remote location to another via an asynchronous receiver `FileReceiverAsync`.

OnComplete

```
1  @FunctionalInterface
2                  OnComplete
   public interface {
3
4           onComplete(FileWriterProxy
       void fileWriter);
```

```
5  }
```

The `OnComplete` interface represents a callback abstraction that we pass to our `FileReceiverAsync` implementation with the purposes of executing this once a file has been successfully and thoroughly transferred. We pass a `FileWriterProxy` to the `onComplete(...)` and this can server as context when executing said method.

FileWriterProxy

```
01             FileWriterProxy
    final class {
02
03                   FileWriter
        private final fileWriter;
04      private final AtomicLong position;
05      private final long size;
06                   String
        private final fileName;
07
08                            String
    FileWriterProxy(final path,       final FileMetaData metaData)throws
    IOException
    {
09                !Objects.isNull(metaData)  &&
            assert StringUtils.isNotEmpty(path);
10
11              .fileWriter      FileWriter(path      +
            this=            new +                "/" metaData.getFileName());
12              .position
            this=          new AtomicLong(0l);
13              .size =
            thismetaData.getSize();
```

```
14              .fileName =
            thismetaData.getFileName();
15      }
16
17      String getFileName()
        {
18          return this.fileName;
19      }
20
21      FileWriter getFileWriter()
        {
22          return this.fileWriter;
23      }
24
25      AtomicLong getPosition()
        {
26          return this.position;
27      }
28
29              done()
        boolean {
30                      .position.get()
            return this==                 this.size;
31      }
```

```
32  }
```

The `FileWriterProxy` represents a proxy abstraction that wraps a `FileWriter` and encapsulates `FileMetaData`. All of this is needed when determining what to name the file, where to write the file and what the file size is so that we know when the file transfer is complete. During transfer negotiation this meta information is compiled via a custom protocol we implement before actual file transfer takes place.

FileReceiverAsync

```
001             FileReceiverAsync
    final class {
002
003                     AsynchronousServerSocketChannel
        private final server;
004                     AsynchronousChannelGroup
        private final group;
005                     String
        private final path;
006     private final OnComplete onFileComplete;
007
008                                                     String
        FileReceiverAsync(final int port,final int poolSize,final path,       final
    OnComplete onFileComplete)
    {
009         assert !Objects.isNull(path);
010
011             .path =
            thispath;
012             .onFileComplete =
            thisonFileComplete;
013
014         try {
```

```
015             this
    .group =
    AsynchronousChannelGroup.withThreadPool(Executors.newFixedThreadPool(poolSize));

016                 .server =
                thisAsynchronousServerSocketChannel.open(          this
    .group).bind(new InetSocketAddress(port));

017                 (IOException e)
            }catch {

018                                                 "unable to start              ,
                throw new IllegalStateException(FileReceiver"                     e);

019         }

020     }

021

022          start()
        void {

023         accept();

024     }

025

026                     wait)
        void stop(long {

027

028         try {

029             this.group.shutdown();

030                 .group.awaitTermination(wait,
                thisTimeUnit.MILLISECONDS);

031                 (InterruptedException e)
            }catch {
```

```
032                                     "unable to stop                 ,
                throw new RuntimeException(FileReceiver"                    e);
033         }
034     }
035
036                             AsynchronousSocketChannel
        private void read(final channel,                                final
    FileWriterProxy proxy)
    {
037                !Objects.isNull(channel) &&
            assert !Objects.isNull(proxy);
038
039                ByteBuffer buffer =
            final ByteBuffer.allocate(Constants.BUFFER_SIZE);
040         channel.read(buffer,
            proxy,                      new
    CompletionHandler<Integer, FileWriterProxy>()
    {
041
042             @Override
043                                         Integer
            public void completed(final result,         final
    FileWriterProxy attachment)
    {
044                   (result    )
                  if >=        0{
045                      (result   )
                     if >        0{
046                         writeToFile(channel, buffer,
                            attachment);
```

```
047                     }
048
049                     buffer.clear();
050                     channel.read(buffer,
                        attachment,                     this);
051                         (result    || attachment.done())
                    }else if <        0 {
052                     onComplete(attachment);
053                     close(channel,
                        attachment);
054                 }
055             }
056
057             @Override
058             public void failed(final Throwable exc,final
    FileWriterProxy attachment)
    {
059                                         "unable to read      ,
                    throw new RuntimeException(data"                exc);
060             }
061         });
062     }
063
064                                 FileWriterProxy proxy)
        private void onComplete(final {
```

```
065          assert !Objects.isNull(proxy);
066
067          this.onFileComplete.onComplete(proxy);
068      }
069
070                                AsynchronousSocketChannel channel)
         private void meta(final {
071          assert !Objects.isNull(channel);
072
073                ByteBuffer buffer =
             final ByteBuffer.allocate(Constants.BUFFER_SIZE);
074          channel.read(buffer,new StringBuffer(),new
     CompletionHandler<Integer, StringBuffer>()
     {
075
076              @Override
077                                              Integer
                 public void completed(final result,        final
     StringBuffer attachment)
     {
078                    (result    )
                     if <         0{
079                      close(channel,null);
080                  }else {
081
```

```
082                         (result   )
                        if >        0{
083                             attachment.append(new String(buffer.array()).trim());
084                         }
085
086                         if
    (attachment.toString().contains(Constants.END_MESSAGE_MARKER))
    {
087                             final
    FileMetaData metaData =
    FileMetaData.from(attachment.toString());
088                             FileWriterProxy
                                fileWriterProxy;
089
090                             try {
091                                 fileWriterProxy
                                    =                 new
                                                .path,
    FileWriterProxy(FileReceiverAsync.thismetaData);
092                                 confirm(channel,
                                    fileWriterProxy);
093                                    (IOException e)
                                }catch {
094                                 close(channel,null);
095                                 throw new RuntimeException(
    "unable to create file writer       ,
    proxy"                              e);
096                             }
097                         }else {
```

```
098                         buffer.clear();
099                         channel.read(buffer,
                            attachment,                     this);
100                     }
101                 }
102             }
103
104             @Override
105             public void failed(final Throwable exc,final
    StringBuffer attachment)
    {
106                 close(channel,null);
107                                             "unable to read meta      ,
                    throw new RuntimeException(data"                      exc);
108             }
109         });
110     }
111
112                                 AsynchronousSocketChannel
        private void confirm(final channel,                           final
    FileWriterProxy proxy)
    {
113                 !Objects.isNull(channel) &&
            assert !Objects.isNull(proxy);
114
```

```
115         final
    ByteBuffer buffer =
    ByteBuffer.wrap(Constants.CONFIRMATION.getBytes());
116                                         CompletionHandler<Integer, Void>()
            channel.write(buffer,null,new {
117
118             @Override
119                                          Integer             Void attachment)
                public void completed(final result,        final {
120                     (buffer.hasRemaining())
                    while {
121                     channel.write(buffer,null,this);
122                 }
123
124                 read(channel, proxy);
125             }
126
127             @Override
128                                                      Void attachment)
                public void failed(final Throwable exc,final {
129                 close(channel,null);
130                                           "unable to         ,
                    throw new RuntimeException(confirm"          exc);
131             }
132
```

```
133         });
134     }
135
136                 accept()
        private void {
137                                     CompletionHandler()
            this.server.accept(null,new {
138                                       AsynchronousSocketChannel
                public void completed(final channel,                    final
    Void attachment)
    {
139
140
141                 accept();
142
143
144                 meta(channel);
145             }
146
147                                                       Void attachment)
                public void failed(final Throwable exc,final {
148                                         "unable to accept new
                    throw new RuntimeException(connection"
    ,
    exc);
149             }
```

```
150         });
151     }
152
153     private void writeToFile(final AsynchronousSocketChannel channel, final
    ByteBuffer buffer, final FileWriterProxy proxy)
154         assert !Objects.isNull(buffer) && !Objects.isNull(proxy) &&
    !Objects.isNull(channel);
155
156         try {
157             buffer.flip();
158
159             final long bytesWritten = proxy.getFileWriter().write(buffer,
    proxy.getPosition().get());
160             proxy.getPosition().addAndGet(bytesWritten);
161                (IOException e)
            }catch {
162             close(channel,
                proxy);
163                                    "unable to write bytes to      ,
            throw new RuntimeException(file"                          e);
164         }
165     }
166
```

```
167                                 AsynchronousSocketChannel
        private void close(final channel,                                 final
    FileWriterProxy proxy)
    {
```

```
168         assert !Objects.isNull(channel);
```

```
169
```

```
170         try {
```

```
171
```

```
172                (!Objects.isNull(proxy))
               if {
```

```
173                 proxy.getFileWriter().close();
```

```
174             }
```

```
175             channel.close();
```

```
176                (IOException e)
            }catch {
```

```
177                                        "unable to close channel and
               throw new RuntimeException(FileWriter"
    ,
    e);
```

```
178         }
```

```
179     }
```

The `FileReceiverAsync` abstraction builds upon the idiomatic use of AsynchronousChannels demonstrated in this tutorial.

6. Running the program

The program can be run from within the IDE, using the normal JUnit Runner or from the command line using maven. Ensure that the test resources (large source files and target directories exist).
Running tests from command line

```
1  mvn
   clean    install
```

You can edit these in the `AbstractTest` and `FileCopyAsyncTest` classes. Fair warning the `FileCopyAsyncTest` can run for a while as it is designed to copy two large files asynchronously, and the test case waits on a CountDownLatch without a max wait time specified.

I ran the tests using the "spring-tool-suite-3.8.1.RELEASE-e4.6-linux-gtk-x86_64.tar.gz" file downloaded from the SpringSource website. This file is approximately 483mb large and below are my test elapsed times. (using a very old laptop).

Test elapsed time

```
1  Running com.javacodegeeks.nio.large_file_transfer.remote.FileCopyTest

2  Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.459 sec
   -                                                                    in
   com.javacodegeeks.nio.large_file_transfer.remote.FileCopyTest

3  Running com.javacodegeeks.nio.large_file_transfer.remote.FileCopyAsyncTest

4  Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 26.423 sec
   -                                                                     in
   com.javacodegeeks.nio.large_file_transfer.remote.FileCopyAsyncTest

5  Running
   com.javacodegeeks.nio.large_file_transfer.        local.FileCopyTest

6  Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.562 sec
   -                                                                    in
   com.javacodegeeks.nio.large_file_transfer.local.FileCopyTest
```

7. Summary

In this tutorial, we demonstrated how to transfer a large file from one point to another. This was showcased via a local copy and a remote transfer via sockets. We went one step further and demonstrated transferring a large file from one remote location to another via an asynchronous receiving node.

8. Download the source code

This was a Java NIO Large File Transfer tutorial

Download
You can download the full source code of this example here: **Java Nio Large File Transfer**