II. Java New I/O

Zoltán Majó / Christoph Denzler

Input/Output (I/O) Operationen sind üblicherweise mindestens eine Grössenordnung langsamer als normale Memory-Zugriffe. Es lohnt sich deshalb bei der Performanceoptimierung auch bei den I/O-Operationen anzusetzen!

Java NIO bietet nun folgende Features:

- Channels und Buffers
- Memory mapped files
- File locking
- Nonblocking I/O

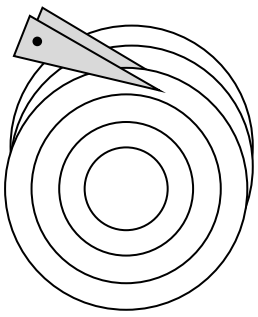
1. Warum java.nio, was war denn nicht gut an java.io?

Das Java I/O-Modell welches im Package java.io implementiert ist, arbeitet hauptsächlich mit der Abstraktion von Strömen (Streams). Dh. I/O-Operationen werden als sequentielle Folgen von z.B. Bytes betrachtet. Dieses Modell ist korrekt und einfach verständlich. Aber es ist langsam. Betrachten wir dies doch im Folgenden an den Beispielen File-I/O und Network-I/O etwas genauer:

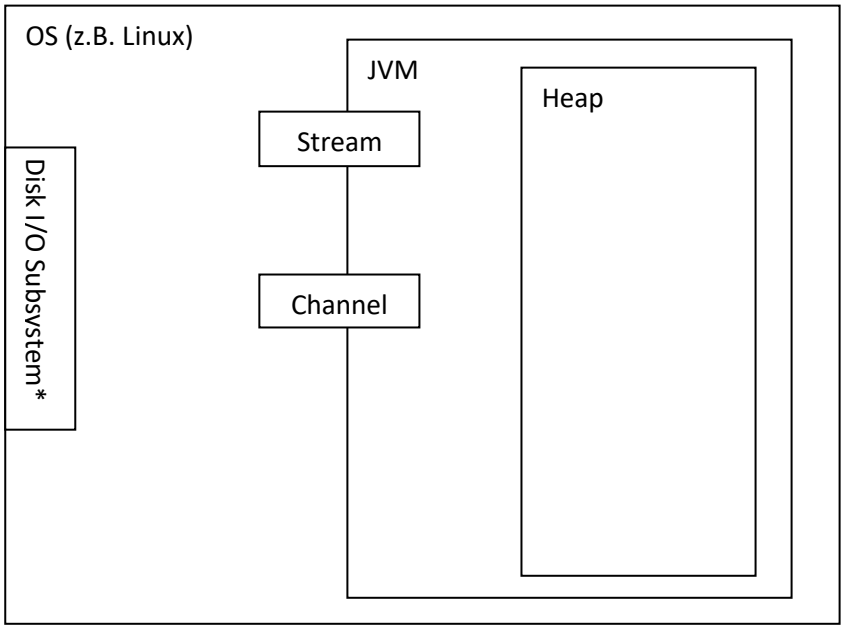
1.1 Files

Im Umgang mit Files gelten häufig folgende Aussagen:

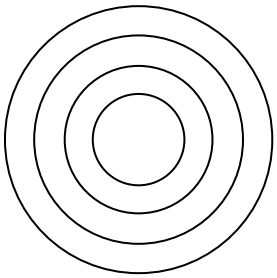
1. **Files werden nicht rein sequentiell verarbeitet, sondern in Blöcken.**
   Die Speicherdichte und –grösse heutiger Festplatten erlaubt es diesen gar nicht mehr auf einzelne Bytes zuzugreifen. Deshalb wird eine Festplatte in verschiedene logische Einheiten (Partitionen, Sektoren, Blöcke) eingeteilt. Die kleinste Einheit (der Block) bildet dabei auch die minimale Einheit in der Daten von der Disk gelesen oder auf die Disk geschrieben werden:

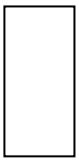


Disk



2. **Files werden nicht *nur* gelesen oder *nur* geschrieben.**
   Files werden verwendet um Speicher auszulagern. Dabei soll die Darstellung im File möglichst gleich sein wie die Belegung im Hauptspeicher. Beispiele:
   - Temporäre Dateien für temporäre Daten
   - Datenbanken (BLOBs, Indizes, …)





3. **Der Inhalt von Files interessiert häufig gar nicht.**
   Einige typische Situationen:
   - Kopieren von Files
   - Streaming von Audio und Video (serverseitig)
   - File- oder statischer Web-Server (Ausliefern von angeforderten Files)

Das Beispiel eines Filecopy-Programms soll die Unterschiede verdeutlichen helfen. Hier die Implementation mit java.io:

```
1   FileInputStream fin = new FileInputStream( src );
2   FileOutputStream fout = new FileOutputStream( dest );
3
4    int b = fin.read();
5    while (b >= 0) {
6      fout.write(b);
7      b = fin.read();
8    }
```

Diese Implementation liest sequentiell jedes einzelne Byte aus dem InputStream und schreibt es sogleich wieder in den OutputStream. Bei Erreichen des EOF, wird ein -1 gelesen.

1.2 Blöcke oder Ströme?

I/O-Operationen werden verwendet um mit der „Aussenwelt“ zu kommunizieren. Die Stream-Metapher verwendet dafür einen Strom von Bytes. Ein Strom hat eine „Flussrichtung“ kann also nur Input- oder Output-Operationen unterstützen. Dabei werden I/O-Operationen als sequentielle Lese- und Schreibzugriffe auf Bytes verstanden. Die Bytes werden einzeln und sequentiell (d.h. ein Byte nach dem anderen) bearbeitet. Ein ByteStream dient als Basisstrom der zudem beliebig dekoriert werden kann (PrintStream, BufferedStream, ObjectStream, etc…). Streams werden auch intern verwendet um Objekte zu serialisieren.

Diese Metapher ist sehr einfach, entspricht aber nicht der Art und Weise wie die meisten Betriebssystem heute I/O verarbeiten. NIO verwendet daher Blöcke. Es ist also nicht mehr das einzelne Byte die kleinste Einheit die verarbeitet wird, sondern ein ganzer Block. *Block*-I/O ist sehr viel schneller als *streamed*-I/O. Aber leider auch nicht mehr so einfach und elegant.

2. Channels and Buffers

2.1 Buffers

- Ein Buffer enthält Daten auf die lesend *und* schreibend zugegriffen werden kann.
- Buffer sind die signifikanteste Änderung von NIO gegenüber dem herkömmlichen I/O Modell. Denn auf die Stream-Objekte konnte man direkt Lese- und Schreiboperationen aufrufen.
- In NIO werden diese Operationen ausschliesslich auf den Buffers gemacht.
- Ein Buffer ist im Wesentlichen ein Byte-Array (`byte[]`).
- Für die meisten Anwendungen wird man einen `ByteBuffer` verwenden. Es gibt aber für alle primitiven Typen auch eigene Buffers: `CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer, DoubleBuffer.` Zu finden im Package `java.nio.`

2.2 Channels

- Channels sind ähnlich zu den Streams, denn alle Daten die gelesen und geschrieben werden, müssen durch einen Channel gehen.
- Im Gegensatz zu Streams sind Channels bidirektional, d.h. man kann schreibend *und* lesend auf denselben Channel zugreifen.
- Channels lesen und schreiben ihre Daten in Blöcken, also *niemals* in einzelnen Bytes.
- Man kann nicht direkt in einen Channel schreiben oder daraus lesen. Dies geschieht immer über einen zugeordneten Buffer.

2.3 Beispiel: FileCopy mit NIO

Ein Lesezugriff funktioniert nun in 3 Schritten: Statt einfach einen FileInputStream zu öffnen müssen wir zuerst einen Channel bereitstellen (1). Dann können wir diesen Channel verwenden um die Daten zu lesen. Aber die Daten stehen dann nicht im Channel direkt zur Verfügung, sondern nur in einem Buffer. Dieser muss erst erstellt werden (2). Danach können die Daten aus dem Channel in den Buffer gelesen werden (3). Diese Schritte im Code:

1. Wir erhalten einen Channel über einen FileInputStream:
```
FileInputStream fin = new FileInputStream("readme.txt");
FileChannel fcin = fin.getChannel();
```

2. Danach erzeugen wir einen Buffer:
```
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
```

3. Und lesen die Bytes aus dem Channel in den Buffer:
```
int r = fcin.read( buffer );
```
   Dabei wird als Resultat die Anzahl tatsächlich gelesener Bytes zurückgegeben oder -1 falls das Ende der Datei erreicht wurde. Beachte: wir müssen dem Channel nicht angeben, wie viele Bytes wir lesen möchten. Jeder Buffer hält Buch über seine Daten. Mehr dazu in Abschnitt 3.

Der Schreibzugriff ist analog:

1. Wir erhalten einen Channel über einen FileOutputStream:
```
FileOutputStream fout = new FileOutputStream("writeme.txt");
FileChannel fcout = fout.getChannel();
```

2. Danach erzeugen wir einen Buffer:
```
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
```

Dann schreiben wir in den Buffer ():

```
for(int i = 0; i < message.length; i++) {
  buffer.put(message[i]);
}
buffer.flip();
```

`flip` bereitet den Buffer vor, in einen Channel geschrieben zu werden

3. Zuletzt wird der Buffer in den Channel geschrieben:

   ```
   int w = fcout.write( buffer );
   ```

   Auch hier zeigt das Resultat an, wie viele Bytes geschrieben wurden (möglicherweise auch 0). Wiederum müssen wir dem Buffer nicht sagen, wie viele Bytes geschrieben werden müssen!

Nun können wir dies zum CopyFile-Programm zusammensetzen:

```
 1  FileInputStream fin = new FileInputStream( src );
 2  FileOutputStream fout = new FileOutputStream( dest );

 3  FileChannel fcin = fin.getChannel();
 4  FileChannel fcout = fout.getChannel();

 5  ByteBuffer buffer = ByteBuffer.allocate( 1024 );

 6  int r = fcin.read( buffer );
 7  while (r >= 0) {
 8    buffer.flip();

 9    fcout.write( buffer );
10    buffer.clear();
11    r = fcin.read( buffer );
12  }
```

Die tatsächlichen I/O-Operationen finden nur in den Zeilen 11 (6) und 9 statt. In Zeile 8 wird der Buffer mit flip() vorbereitet für das Schreiben auf einen Channel. Die clear-Operation in Zeile 10 ist die analoge Operation für das Lesen: der Buffer erfährt einen Reset und ist danach bereit neue Daten aufzunehmen.

3. Buffers im Detail

Es wurde schon darauf hingewiesen, dass Buffers ihre Daten selber verwalten und wissen, wie viele Bytes geschrieben oder gelesen werden können bzw. müssen. Im CopyFile Beispiel haben uns die einzelnen Bytes nicht interessiert. Will man trotzdem auf die Daten zugreifen, dann geschieht dies durch die Zugriffsmethoden get() und put(). Zusätzlich ist der Zustand eines Buffers über folgende drei Werte definiert:

- `position`
- `limit`
- `capacity`

Diese drei Variablen führen Buch über die Daten im Buffer.

Position

Ein Buffer ist nur ein aufgepeppter Byte-Array. Eine `get()` oder `put()`-Operation arbeitet also auf einem darunterliegenden `byte[]`. Es muss daher eine Position geben an der aktuell gelesen oder geschrieben wird. `position` ist also der Positionszeiger, der bei den get- und put-Operationen auch entsprechend der Anzahl Bytes die gelesen oder geschrieben wurden erhöht wird.

Limit

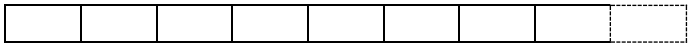
Der `limit`-Wert gibt an, bis zu welcher Position Daten in einem Buffer bereit stehen (im read-Fall). Oder es ist die Position bis wohin in den Buffer geschrieben werden kann. Es gilt immer `position` ≤ `limit`

Capacity

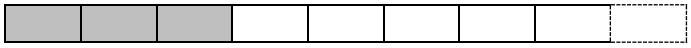
`capacity` gibt an, wie viele Bytes überhaupt in den Buffer passen. Capacity gibt also die Länge des darunterliegenden Arrays an – oder mindestens die Grösse des Arrays die wir benutzen dürfen (und kann daher auch nicht neu gesetzt werden!) Es gilt zu jeder Zeit: `limit` ≤ `capacity`

Beispielablauf:

1. Neuer Buffer mit 8 Bytes



2. Es werden 3 Bytes vom Channel in den Buffer gelesen



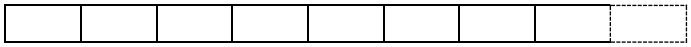
3. Es werden nochmals 2 Bytes vom Channel in den Buffer gelesen

4. Wir sind nun bereit, den Buffer in einen anderen Channel zu schreiben. Es wird ein `flip()` ausgeführt.

5. Der Output-Channel schreibt in Blöcken von 4 Bytes. Nach der ersten Schreiboperation :

6. Der zweite Block wird geschrieben. Nach der zweiten Schreiboperation:

7. Der Buffer wird nun wieder zum Lesen vorbereitet. Es wird ein `clear()` ausgeführt.



Betrachten Sie die API-Dokumentation zu den get- und put-Methoden in java.nio.ByteBuffer. Es gibt sowohl relative als auch absolute Zugriffsmethoden, also mit oder ohne Positionsangaben, wo gelesen oder

geschrieben werden soll. Zudem gibt es auch Methoden um Wertetypen zu lesen/schreiben (auch relativ und absolut).

Bisher haben wir die Standard-Eigenschaften der Buffers betrachtet. Wenn man mit Channels und Buffers arbeitet ist dieses Wissen unabdingbar. Im Folgenden sind einige weitere Features erwähnt, die die Arbeit mit Buffers noch effizienter machen können.

4. Fortgeschrittene Operationen mit Buffers

4.1 Buffer wrapping

Wir haben bisher gesehen, dass ein Buffer über die statische Methode allocate angelegt werden kann.

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
```

Zudem haben wir gesehen, dass ein Buffer immer auf einem Byte-Array arbeitet. Warum also nicht einen ByteBuffer erzeugen, indem man selbst ein `byte[]` übergibt:

```
byte[] array = new byte[1024];
ByteBuffer buffer = ByteBuffer.wrap(array);
```

Dies ist in vielen Fällen sehr effizient, weil jetzt auch das Kopieren des Arrays entfällt.

Warnung: Der Array kann jetzt natürlich auch direkt verändert werden ohne die Zugriffsoperationen des Buffers zu benützen. Die Daten des Buffers sind somit nicht mehr sauber gekapselt, die Variablen `position` und `limit` können dadurch nicht mehr stimmen. Dies ist zwar unschön aber effizient!

4.2 Memory-mapped file I/O

Noch einen Schritt weitergehen würde bedeuten, dass der Inhalt eines Files direkt in den Hauptspeicher abgebildet wird. Dies ist mit dem memory-mapped file I/O tatsächlich auch möglich!

Dabei wird immer ein Teil (blockweise) des Files in den Hauptspeicher abgebildet. Sämtliche Änderungen an einem Byte im Speicher bedeutet gleichzeitig auch eine Änderung im File. Memory-mapped file I/O kann angewendet werden um sehr effizient grosse Datenmengen zu bearbeiten. Beispiele: Bild-, Ton- oder Videobearbeitung.

Folgender Code bildet das erste Kilobyte eines Files in einen Buffer ab:

```
RandomAccessFile raf = new RandomAccessFile("somefile.txt", "rw");
FileChannel fc = raf.getChannel();

MappedByteBuffer buffer = fc.map( FileChannel.MapMode.READ_WRITE, 0, 1024);
```

MappedByteBuffer ist eine Unterklasse von ByteBuffer. Man kann also damit genau gleich arbeiten wie gewohnt. Das Betriebssystem ist verantwortlich das Mapping korrekt zu auszuführen.

Beachte: Dies ist die einzige Möglichkeit einen memory-mapped Buffer zu erstellen! Buffer wrapping ist in Kombination mit Memory-mapping nicht möglich. Warum?

4.3 File locking

Der Begriff des File locking kann irreführend sein. In Java NIO wird damit *nicht* der Zugriff auf ein File (oder einen Teil davon) für andere Benutzern oder Prozessen verhindert. Es handelt sich um einen Lock im Sinne von Concurrent Programming, also um ein Mittel zur Synchronisation von Prozessen.

Es ist möglich ein ganzes File zu sperren mit `lock()` oder nur Teile davon `lock(position, size, shared)`. Shared locks ermöglichen es anderen Prozessen auch shared locks zu erwerben. Es ist aber nicht möglich auf einem File mit shared locks einen exklusiven lock zu erwerben.

In den meisten Betriebssystemen sind file locking-Operationen atomar. Es ist somit möglich über ein file locking Prozesse zu synchronisieren, d.h. es ist möglich eine Java-Applikation mit einer anderen (z.B. C-Applikation) zu synchronisieren. File locks werden im Namen der JVM gehalten und nicht im Namen eines einzelnen Threads. Sie eignen sich daher nicht für die Synchronisation von Filezugriffen einzelner Threads in derselben JVM. Dafür sind die normalen Synchronisationsmechanismen von Java zu verwenden.

Ein File-Lock kann wie folgt erworben werden:

```
RandomAccessFile raf = new RandomAccessFile( "somefile.txt", "rw" );
FileChannel fc = raf.getChannel();

// Get lock
FileLock lock = fc.lock( start, end, false );
```

Hier wird ein Bereich von *start* bis *end* des Files `somefile.txt` gesperrt und zwar mit einem exklusiven Lock. Wird dieser Lock nicht mehr gebraucht, kann er mit folgender Anweisung wieder aufgehoben werden:

```
lock.release();
```

Portabilität

Da File locking auf das darunterliegende File- und Betriebssystem abstützt ist die Semantik eines Locks nicht immer dieselbe. Um eine möglichst hohe Portabilität einer Applikation zu erhalten die File Locks verwendet sollten folgende beide Regeln eingehalten werden:

- Nur exklusive Locks verwenden
- Alle Locks nur als Hinweis verwenden und sich nicht darauf verlassen, dass der Zugriff auch tatsächlich gesperrt ist.

4.4 Networking mit asynchronen I/O-Operationen

Networking in NIO wird genau gleich verwendet wie die anderen Operationen in NIO: mit Channels und Buffers. Die herkömmlichen I/O-Operationen von java.io blockieren aber solange, bis weitere Daten zum Lesen oder zum Schreiben zur Verfügung stehen. Dies führt dazu, dass pro Netzwerkverbindung (Socket) ein Thread verwendet wird welcher für den Datenverkehr zuständig ist. Damit ist es möglich mehrere Verbindungen gleichzeitig zu bedienen, auch wenn einige davon gerade auf Daten warten. Bei einer grossen Anzahl Verbindungen wird aber das Thread-Management und Context-Switching zum Flaschenhals. Wenn die read() und write()-Operationen nicht blockieren würden, dann könnten mehrere Verbindungen von nur einem Thread bedient werden und damit könnten auch Ressourcen geschont werden.

Asynchrone Aufrufe blockieren natürlich nicht, sie lesen oder schreiben immer, möglicherweise auch 0 Bytes! Es braucht also eine Einrichtung, die einem sagt, ob Daten bereit sind. Dies ist die Aufgabe eines Selectors.

4.4.1 Selector

Nicht alle Channels unterstützen asynchrone I/O-Operationen sondern nur solche, die eine Unterklasse von java.nio.channels.SelectableChannel sind. Ein SelectableChannel kann einen Selector über Ereignisse informieren, die auf seinen Daten geschehen sind. Ein Selector ist ein Observer eines SelectableChannels. Um Events von einem Channel zu empfangen benötigt man also zuerst einmal einen Selector:

```
Selector selector = Selector.open();
```

Später werden wir diesen Selector bei einem Channel registrieren. Zunächst brauchen wir dazu aber einen Channel:

```
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.configureBlocking( false );
ServerSocket ss = ssc.socket();
InetSocketAddress address = new InetSocketAddress( 8080 );
ss.bind( address );
```

Hier wird ein ServerSocketChannel geöffnet um Netzwerkverbindungen empfangen zu können. Dieser Channel soll asynchron – also nicht blockierend – arbeiten. Der Aufruf von `ssc.configureBlocking(false)` ist wichtig, da sonst synchrone I/O-Operationen ausgeführt werden. Zudem benötigen wir den Socket dieses Channels um ihn an die Adresse 8080 binden zu können. Jetzt steht also ein Channel bereit um Internetverbindungen aufbauen zu können. Wie aber erfahren wir, ob z.B. eine eingehende Verbindung aufgebaut worden ist und ob Daten anliegen?

Dazu muss nun der Selector beim Channel registriert werden und zwar mit dem Wunsch über bestimmte Ereignisse informiert zu werden:

```
SelectionKey key = ssc.register( selector, SelectionKey.OP_ACCEPT );
```

Damit wird der Channel aufgefordert Ereignisse der Art ACCEPT (also das akzeptieren einer Verbindung) an den Selector zu melden. Das Resultat dieser Registrierung ist ein Schlüssel, der diese Registrierung repräsentiert. Wenn der Selector nun von einer eingehenden Verbindung in Kenntnis gesetzt wird (zur Erinnerung: der Selector beobachtet den Channel im Sinne eines Observers), dann setzt er im entsprechenden SelectionKey ein ready-Bit. In diesem Falle das OP_ACCEPT-Bit.

Beachte: Der Selector notifiziert selber keine anderen Objekte. Er führt nur die ready-Bits in den Registrierungen (= SelectionKey-Objekte) durch. Ob nun eine I/O-Operation anliegt oder nicht, muss beim Selector abgefragt werden (polling). Hier ein Codebeispiel dafür:

```
1   while (true) {
2     selector.select();

3     Set<SelectionKey> selectedKeys = selector.selectedKeys();
4     Iterator<SelectionKey> it = selectedKeys.iterator();

5     while (it.hasNext()) {
6       SelectionKey key = (SelectionKey)it.next();
7       // deal with I/O event
8       it.remove();
      }
    }
```

Hier werden in Zeile 2 zuerst einmal die Events selektiert, die vorhin registriert wurden. Das Ergebnis dieser Selektion wird in Zeile 3 zwischengespeichert. selectedKeys kann man sich als eine Art Eventqueue vorstellen. Daraus bezieht man nun einen Event um den anderen (Zeile 4 – 6). In Zeile 7 wird der Event verarbeitet. Wichtig

ist, dass die Eventqueue auch geleert wird. Dies passiert in Zeile 8, wo jeder SelecktionKey der bearbeitet wurde aus den selectedKeys entfernt wird.

Wie sieht denn der kommentierte Abschnitt : `// deal with I/O event` genauer aus?

```
1  if ((key.readyOps() & SelectionKey.OP_ACCEPT) == SelectionKey.OP_ACCEPT) {
2    ServerSocketChannel ssc = (ServerSocketChannel)key.channel();
3    SocketChannel sc = ssc.accept();
4    sc.configureBlocking( false );
5    sc.register( selector, SelectionKey.OP_READ );
6  } else if ((key.readyOps() & SelectionKey.OP_READ) == SelectionKey.OP_READ) {
7    SocketChannel sc = (SocketChannel)key.channel();
8    requestBuffer.clear();
9    sc.read(requestBuffer);
```

Für jeden SelectionKey wird geprüft um welches I/O-Ereignis es sich handelt (Zeile 1 und 6). Im Falle eines ACCEPT-Events wird der ServerSocketChannel verwendet um die Verbindung zu akzeptieren und dabei auch gleich noch einen neuen SocketChannel (inkl. Socket) zu erstellen (Zeile 3). Damit auch mit diesem SocketChannel non-blocking gearbeitet werden kann, muss das entsprechende Flag richtig gesetzt werden (Zeile 4). Nun haben wir einen neuen SocketChannel und möchten natürlich informiert werden sobald Daten zur Verarbeitung anstehen. Dazu wird auf dem SocketChannel der selector registriert, so dass er READ-Ereignisse empfängt (Zeile 5). Im Falle eines solchen Lese-Ereignisses, wird der Channel aus dem SelectionKey extrahiert (Zeile 7). Nachdem (in Zeile 8) ein Buffer bereitgestellt wurde, kann dann dieser die Daten aus der Netzwerkverbindung aufnehmen (Zeile 9).

Das ganze Programm steht als Klasse `MultiPortEcho.java` zur Verfügung.