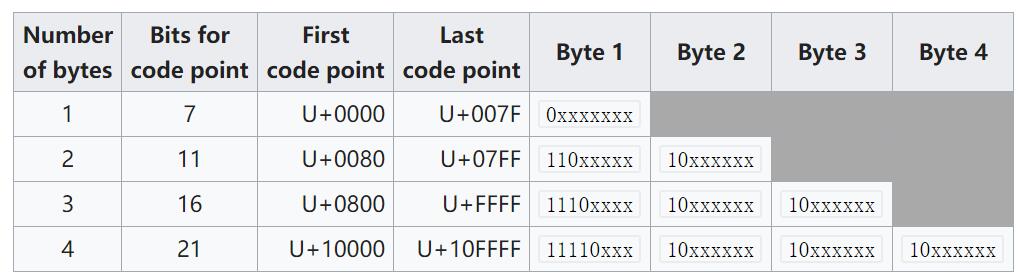
# Übungen Java NIO

## Aufgabe 1

Die Methode «mark()» ermöglicht es, die aktuelle Position eines Buffers zwischen zu speichern. Mit der Methode «reset()» kann dann auf diese Markierung zurückgesprungen und als neue Position wiederverwendet werden. Wird ohne Markierung zurückgesprungen, wird eine Exception geworfen

Anwendung könnte z.B. das Parsen einer Byte Sequenz sein, in welcher UTF-8 Zeichen erkennt werden müssen (Diese können 1 – 4 Bytes lang und je nachdem auch mal fehlerhaft sein). Je nachdem möchte man die aktuelle Position markieren, um bei einer fehlerhaften Sequenz zurückspringen zu können, um ein Element zu verwerfen/droppen und wieder regulär weiterarbeiten zu können:

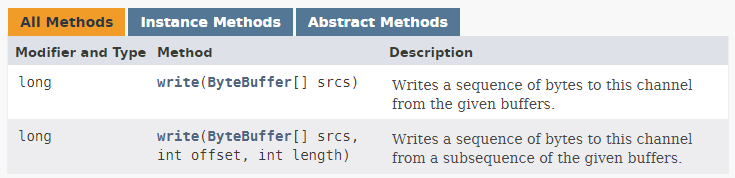


## Aufgabe 2

Bei einem «Gathering Write» wird dem Channel ein Array von Buffern zum Schreiben übergeben. Der Channel arbeitet die übergebenen Buffers dann sequentiell ab: Er liesst die effektiv verbleibenden Daten von Position bis Limit und schreibt diese auf den Channel hinaus.

Anwendung könnte z.B. das Schreiben einer TCP Response sein, welche über einen statischen Header und Body verfügt.

Interessante Methoden von GatheringByteChannel:

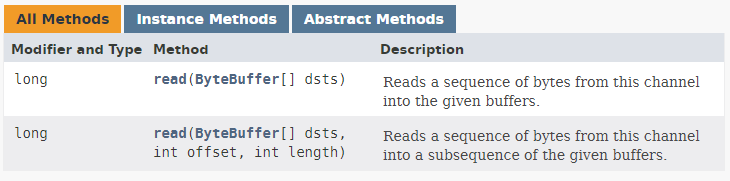


## Aufgabe 3

Bei einem «Scattering Read» wird dem Channel ein Array von Buffers zum Befüllen übergeben. Der Channel liest Daten ein und arbeitet die Buffers sequentiell ab: Er befüllt diese, bis sie voll sind.

Anwendung könnte z.B. das Lesen einer TCP Anfrage sein, welche über einen statischen Header und Body verfügt. Problematisch/nachteilig hierbei ist jedoch, dass durch das Übergeben des Arrays mit den zu verwendeten Buffern die maximale Leselänge zwangsläufig schon vorbestimmt ist (Nämlich die Länge aller Buffers, also Capacity - Position). Dynamische Bodylängen sind so nicht möglich oder aber man muss diese einzeln via Body Buffer weiterlesen

Interessante Methoden von ScatteringByteChannel:



## Aufgabe 4

In der Regel verwendet der ByteBuffer als Buffer ein auf dem JVM Heap alloziertes byte[] Array. Dies führt zu zwei Problemen:

* Werden diese Buffers/Arrays häufig erstellt und via GC wieder weggeräumt, kann sich dafür benötigte Zeit aufsummieren und nachteilig auswirken (JVM muss viel arbeiten).
* Durch die Abstrahierung der JVM Heap zum Heap des Prozesses, kann nie eine native Performance für das Allozieren/Deallozieren von Speichern erreicht werden

Die Klasse ByteBuffer besitzt deshalb die Methode «allocateDirectory(numberOfBytes)», welche den Speicher für den Buffer nicht im Heap der JVM sondern direkt im Betriebssystem alloziert. Das führt dazu:

* Die JVM verwaltet nur noch eine Reference auf den nativ allozierten Speicher durch eine fixe, reale Adresse, nicht aber mehr den Inhalt des ganzen Speichers
* Durch die gefixte reale Adresse, kann das Betriebssystem den Speicher effektiver handhaben
* Die Implementierung der JVM kann in gewissen Fällen so schneller direkt auf den nativen Speicher zugreifen, da viele Abstraktionsschichten wegfallen/nicht mehr benötigt werden

## Aufgabe 5

Die Methode «slice()» der Klasse ByteBuffer erstellt einen neuen Byte Buffer, welcher weiterhin den bisherigen internen Buffer referenziert, jedoch nur noch als Teilstück von Position bis Capacity. Änderungen am gesclicten Byte Buffer können also nur noch Auswirkungen auf diesen Teilbereich haben (Im Gegensatz zur Methode «duplicate()», welche den ganzen Buffer abdeckt).

Lese- und Schreiboperationen auf einen gesclicten Byte Buffer verändern also die Indizes und je nachdem auch den internen Buffer, nicht aber die Indizes aller anderen ByteBuffers welche auf den gleichen internen Buffer zeigen. Dies ist interessant, wenn parallel (z.B. parallele Verarbeitung des TCP Headers und Body auf einem grossen Buffer) oder unabhängig auf einem internen Buffer gelesen werden soll, doch muss man aufpassen, wer wie den internen Buffer verändert (Data Corruption).

## Aufgabe 7 (Vor Aufgabe 6)

Für das Erstellen der Testreihe wurden drei Systeme verwendet:

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Hardware | Betriebssystem | Java |
| 1 | Desktop PC mit Intel Core i7-4790 Quad Core von Anfang 2017, 16GB RAM und 3 TB HDD | Windows 10 x64 | Oracle Java 10.0.1 JDK x64 |
| 2 | Notebook (Dell M3800) mit Intel Core i7-4702HQ Quad Core von Anfang 2013, 16 GB RAM und 1 TB SSD | Linux Debian 10 x64 | Oracle Java 10.0.1 JDK x64 |
| 3 | Virtualisierte VM (ESXi) mit 2 vCPU E5630 Quad Core von 2010, 4 GB RAM und 200 GB HDD | Linux Debian 8 x64 | Oracle Java 1.7.0-171 JRE x64 |

Mit folgenden Vermutungen bin ich gestartet:

* Java NIO wird wahrscheinlich sehr stark von einer schnellen CPU profitieren (Hohe Taktrate, nicht Anzahl Cores). Die Anzahl CPU Cores wird zu vernachlässigen sein, da wir nicht mit einem Threadpool arbeiten (Sprich wir testen nicht den Throughput eines Server Sockets/Tomcat Servers)
* Je nachdem wird die Geschwindigkeit der Java NIO Schreibvorgänge falsch dargestellt, da durch das Non-Blocking nicht 100% garantiert werden kann, wann genau das Betriebssystem die Daten auf den Disk Buffer schreibt und/oder diesen auf die Disk flusht (Der Write kann delayed werden und die Methode kommt direkt zurück, was die Zeitmessung verfälscht). Auf jeden Fall wird, respektive muss, der Write auf jedem Fall vor dem nächsten flip/read passieren
* Da die Debatte IO vs. NIO sehr kontrovers ist, können dementsprechend die Testresultate auch so ausfallen (Siehe auch Aufgabe 6)

Für die Tests wurden folgende Anpassungen gemacht:

* Ursprünglich wollte ich auch die Klasse StreamCopyFile.java testen. Das einzelne Lesen und Schreiben eines Bytes ist aber performancetechnisch so katastrophal schlecht und dauert so lange, dass ein Performancetest mit grösseren Datensätzen unmöglich wird (Nach 30 Minuten habe ich den initialen Test abgebrochen)
* Die Stream Methoden werden bei einer Blocksize grösser als 8192 nicht nur einmal, sondern fünfmal getestet. Grund dafür ist, dass bei einer einzigen Ausführung Testausreisser entstehen können
* Als maximale Dateigrösse wurde nicht 512 MB, sondern 1 GB verwendet
* Als Buffergrössen wurden folgende Werte verwendet
  + 8 KB aka 8192 (HDD Block Size)
  + 0.5 MB aka 524288
  + 1 MB aka 1048576
  + 10 MB aka 10485760
  + 100 MB aka 104857600

Das Testresultat für System 1 (Desktop PC) sieht wie folgt aus (Reihenfolge: FastStreamCopy, CopyFile, FastCopyFile):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Filesize | 8 KB | 0.5 MB | 1 MB | 10 MB | 100 MB |
| 1  2  4  8  16  32  64  128  256  512  1024  2048  4096  8192  16384  32768  65536  131072  262144  524288  1048576  2097152  4194304  8388608  16777216  33554432  67108864  134217728  268435456  536870912  1073741824 | 0 0 0  1 0 0  0 0 1  0 0 0  1 0 0  0 1 0  1 0 1  0 0 0  0 0 0  1 0 0  0 0 0  0 0 0  0 1 0  0 1 0  0 0 0  0 0 0  3 0 0  3 1 0  3 0 1  3 0 0  3 4 0  6 3 1  6 3 3  12 7 7  22 15 15  34 37 32  68 68 62  156 133 127  296 262 249  784 601 579  2809 1231 1182 | 9 1 0  1 0 0  1 0 0  1 0 0  1 0 0  0 0 0  0 0 0  1 0 0  1 0 0  0 1 0  0 0 0  1 0 0  1 0 1  0 0 0  0 0 0  3 0 0  3 0 0  6 0 0  3 1 1  3 1 1  3 1 0  6 0 1  6 3 0  9 3 1  15 9 9  21 21 15  43 40 35  100 79 65  159 154 129  321 305 254  2184 882 580 | 9 0 1  0 0 0  1 0 0  1 0 0  0 1 0  1 0 0  1 1 0  0 0 0  1 0 1  1 0 0  1 0 0  1 1 0  7 0 0  0 1 0  3 0 0  3 0 1  3 0 0  0 1 0  6 0 1  3 1 0  6 1 0  6 1 0  6 1 1  15 7 6  22 9 9  40 20 15  75 40 34  153 82 70  293 154 129  887 361 255  2379 679 683 | 25 1 3  7 1 3  6 1 1  7 1 3  7 0 3  7 0 3  6 0 4  7 0 3  7 0 1  7 1 1  7 1 3  7 1 3  7 1 1  6 0 3  9 1 1  9 1 4  9 0 4  12 1 4  12 1 3  12 1 4  12 0 3  12 1 3  15 4 6  18 7 9  31 17 15  59 35 26  103 67 46  200 135 92  700 269 173  1414 545 335  3258 1157 699 | 103 10 26  73 9 26  70 9 26  73 7 26  73 6 23  73 7 27  73 7 26  73 6 26  76 9 26  73 9 26  78 6 26  75 7 26  73 9 26  78 9 28  81 7 25  74 6 28  78 10 25  84 6 30  81 7 23  84 7 23  87 9 26  96 8 26  90 10 31  100 15 32  109 23 39  262 220 220  590 516 525  1134 990 945  2531 2068 1918  6024 4255 3797  10692 8097 7657 |

Das Testresultat für System 2 (Notebook) sieht wie folgt aus (Reihenfolge: FastStreamCopy, CopyFile, FastCopyFile):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Filesize | 8 KB | 0.5 MB | 1 MB | 10 MB | 100 MB |
| 1  2  4  8  16  32  64  128  256  512  1024  2048  4096  8192  16384  32768  65536  131072  262144  524288  1048576  2097152  4194304  8388608  16777216  33554432  67108864  134217728  268435456  536870912  1073741824 | 0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 1 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 2  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  1 1 0  2 2 1  4 3 3  7 6 6  15 13 12  33 30 26  60 63 52  249 131 181  294 288 443  520 817 1261  975 1966 2459 | 0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  1 0 0  1 1 1  3 3 2  8 6 5  12 12 9  24 25 19  85 54 137  97 135 185  286 343 618  428 815 1316  1006 2093 2600 | 1 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  1 0 0  0 0 0  0 0 0  1 0 0  0 0 0  0 0 0  0 0 0  0 0 0  2 0 0  0 0 0  1 0 0  0 0 0  2 0 0  1 0 0  2 2 1  3 3 3  6 7 5  13 16 24  26 41 39  50 46 70  98 149 205  213 320 585  375 723 1321  732 2079 2339 | 3 1 3  2 0 2  2 0 2  2 0 2  2 0 2  2 0 2  2 0 2  2 0 2  2 0 2  2 1 2  2 0 2  2 0 2  2 1 2  4 3 2  1 0 3  3 0 2  1 1 2  1 0 3  3 0 2  2 1 2  2 1 3  5 2 3  9 4 5  19 9 8  30 17 13  34 33 23  86 65 44  131 138 263  255 313 495  526 688 1347  1097 2033 2666 | 28 9 34  24 7 27  20 9 25  21 6 24  20 6 13  18 6 12  18 6 12  18 6 12  18 6 11  17 5 12  18 6 11  18 6 11  18 6 11  17 5 12  17 6 11  17 5 11  18 6 12  18 5 12  18 5 12  18 6 12  18 6 12  19 8 13  22 10 14  27 13 17  35 22 22  53 38 32  95 71 54  152 137 106  299 266 316  741 637 880  1058 1544 2501 |

Das Testresultat für System 3 (VM) sieht wie folgt aus (Reihenfolge: FastStreamCopy, CopyFile, FastCopyFile):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Filesize | 8 KB | 0.5 MB | 1 MB | 10 MB | 100 MB |
| 1  2  4  8  16  32  64  128  256  512  1024  2048  4096  8192  16384  32768  65536  131072  262144  524288  1048576  2097152  4194304  8388608  16777216  33554432  67108864  134217728  268435456  536870912  1073741824 | 0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  6 0 1  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 0 0  0 1 0  0 0 0  0 0 0  0 0 0  0 0 0  0 1 0  1 2 1  1 4 2  4 8 6  5 13 13  27 16 11  14 18 33  36 29 28  41 69 74  137 270 743  947 773 1593  1608 1689 2867  3422 4120 6951  12963 11587 13091 | 1 0 0  1 0 0  2 0 0  0 0 1  2 0 0  0 0 0  0 0 0  2 0 0  0 0 0  0 0 0  1 0 0  0 0 0  0 0 0  0 0 0  0 0 2  1 0 1  0 0 0  0 0 0  2 0 0  1 1 1  2 2 2  5 3 3  10 40 62  46 61 64  126 127 125  230 250 252  885 306 327  701 735 1689  1522 1586 3346  3365 4475 6076  9219 15061 18166 | 7 29 0  0 2 0  3 0 4  4 0 7  3 0 7  0 5 0  8 0 7  3 0 2  7 0 0  5 2 0  3 0 0  5 0 3  3 0 3  0 2 0  3 0 0  3 2 0  3 0 1  0 0 0  4 0 3  7 0 3  5 4 4  9 12 6  19 15 10  14 11 90  169 116 149  246 248 294  171 158 746  598 584 1593  1579 1539 3149  3734 4241 6285  8537 10493 13159 | 130 4 8  23 6 6  22 2 6  17 2 5  22 2 8  19 2 5  22 2 5  17 2 6  23 2 6  18 2 6  22 2 6  19 2 6  22 2 9  34 1 13  29 1 10  23 1 8  34 2 7  23 2 6  18 2 4  15 2 4  26 3 5  20 1140 9  36 11 12  124 26 18  54 28 22  286 188 45  222 136 76  465 293 555  1815 449 2866  4081 5034 6462  10328 11263 13527 | 219 33 50  755 27 45  177 19 44  178 19 43  237 18 44  176 19 43  176 20 43  177 22 44  241 18 43  179 19 43  175 18 47  177 18 44  177 18 43  180 18 43  176 19 43  177 18 43  178 18 42  273 18 57  187 18 42  194 18 43  194 22 62  260 21 44  183 26 48  200 33 55  230 50 66  285 83 79  418 133 116  596 323 657  1733 714 2740  3970 2164 6586  11169 12620 14564 |

Fazit:

* NIO ist unter System 1 (Windows) mit einem modernen Prozessor (Taktrate 3.6 – 4.0 GHz) von anfangs 2017 ein mehrfaches schneller als IO 🡪 CPU Speed rules?
* Unter System 2 (Linux) mit einem 5 Jahre alten Prozessor (Taktrate 2.2 – 2.4 GHz) ist dies genau umgekehrt, sprich IO ist mehrheitlich immer schneller als NIO
  + Frage: Würde NIO unter Linux schneller als IO sein, wenn ich einen modernen Prozessor verwenden könnte? Wahrscheinlich ja (Ich hatte zu wenig Zeit, dies zu testen)
* System 3 verfügt über einen noch älteren Prozessor aus dem Jahre 2010 (Immerhin hat er eine Taktrate von 2.53 – 2.8 GHz). Bei kleineren Datengrössen ist NIO hier je nachdem mal noch schneller als IO, bei grösseren Dateien aber immer langsamer (Immerhin nicht um so einen grossen Faktor wie bei System 2).
  + Disclaimer: Da System 3 eine virtualisierte VM ist, können keine Quality of Services gewährt werden. Das Resultat ist dementsprechend nicht sehr aussagekräftig
* Am besten bewähren sich Buffergrössen im Bereich von 0.5 MB bis 1.0 MB. Zu kleine Buffergrössen generieren zu viele Contextswitches, grosse sind zu langsam
  + Frage: Warum sind grosse Buffer so langsam? Durch das «Konzentrieren» auf Lesen oder Schreiben (und dem nicht immer ewigen hin und herspringen), sollte dies doch schneller sein?
  + Mutmassung: Je nachdem ist diese Annahme richtig, doch braucht das Allozieren von 100 MB Speicher am Anfang einfach zu lange, um diesen Vorteil zu sehen 🡪 Testset mit 5 GB oder 10 GB einführen, um dies zu testen

Abschliessende Empfehlung für einen Algorithmus:

* Liegt ein moderner Prozessor mit einer hohen Taktrate vor, sollte NIO verwendet werden
  + Bei der richtigen Wahl einer guten Buffergrösse (0.5 MB – 1.0 MB) ist NIO für alle Dateigrössen immer um den Faktor 2 -3 schneller als IO. Die Dateigrösse spielt also keine Rolle (Wir lassen Minidateien mal aussen vor, da wir in Milli- und nicht Nanosekunden messen)
* Liegt ein älterer Prozessor vor oder wird virtualisiert, sollte IO verwendet werden

## Aufgabe 6

Nach dem Resultat von Aufgabe 7 muss man aufpassen, den Use Case nicht miteinander zu verwechseln: Sequentiell Dateien herumkopieren vs. Komplexe SMTP Anfragen mit viel State verarbeiten.

Folgende Punkte sind interessant:

* Folie 10: Somewhere along the line, someone got «scalable» and «fast» mixed up – and it stuck 🡪 Möchte man einen möglichst hohen Throughput haben (IO) oder möglichst viele parallele Clients mit verhältnismässig mässig Resourcen bedienen können (NIO)
* Folie16: Asynchrones NIO kann nur 75% der originalen IO Leistung erzielen kann. Blockendes (!) NIO hingegen erzielt die gleiche Leistung wie blockendes IO 🡪 Blockierendes NIO … das hat er nicht wirklich gesagt?!
* Folie 17, 25 und 26: Wahrscheinlich muss das Betreiben eines Tomcat Threadpools mit 200+ Threads vor NPTL in Kernel 2.6 eine echte Qual und Herausforderung gewesen sein
* Folie 27: Context Switching ist billig
* Folie 31: Uncontended Synchronisation ist billig, Contended Syhcronisation nicht 🡪Non Blocking State Structures verwenden. Quelle: <https://plumbr.io/blog/locked-threads/improving-lock-performance-in-java>
* Folie 56: Connection Time ist enorm wichtig: HTTP vs. SMTP
* Folie 53: How to write your own String class that doesn’t use that much memory & auto-boxing
* Folie 62: Man kann viele Threads verwalten & besitzen, wenn viele davon auch schlafen 🡪 Asynchroner Dispatcher Thread mit synchronem riesen Thread-Pool

Vieles davon lässt darauf schliessen, dass in modernen Systemen die Handhabung von vielen & schlafenden Threads enorm billig ist/geworden ist. Ein grosses Gesamtbild wurde mir aber nicht ersichtlich, warum ich auf den Thread (<https://news.ycombinator.com/item?id=1551776>) mit folgender Antwort gestossen bin:

* With modern operating systems, idle threads are practically free. 🡪 Wie erwähnt
* Managing non-contending threads is extremely inexpensive now. 🡪 Seit NPTL
* Multi-core systems. 🡪 Man baut heute keine CPU Cores mehr mit mehr als 4 GHz, sprich das parallele Rechnen gewinnt gegenüber einer hohen Taktrate. Witziger Artikel: <http://www.pcgameshardware.de/Retrospektive-Thema-214694/News/Irrtuemer-Intels-10-Gigahertz-und-Nvidias-CineFX-PCGH-Retro-18-November-667403/>
* Selectors and state-restore used by asynchronous NIO libraries in high-load environments is more expensive than waking up and putting to sleep threads. 🡪 Ergibt auch Sinn und führt dazu, dass sich NIO unter High Load selber killen kann, da es sich selber retten will, aber nur noch stärker belastet

## Verwendete Ressourcen

* <https://docs.oracle.com/javase/10/docs/api/java/nio/ByteBuffer.html>
* <http://tutorials.jenkov.com/java-nio/scatter-gather.html>
* <https://examples.javacodegeeks.com/core-java/nio/java-nio-channels-example/>
* <https://www.javamex.com/tutorials/io/nio_buffer_direct.shtml>