java.NIO

Claudiu Soroiu

În cadrul acestui articol vom prezenta o serie de facilități care au fost introduse în librăria de clase java începând cu versiunea 1.4. Utilitatea acestor facilități este destul de semnificativă deoarece cu ajutorul lor se pot îmbunătăți considerabil performanțele aplicațiilor care folosesc transferuri de date din fișiere sau din rețeaua de calculatoare (locală sau Internet).

Voi începe acest articol prin a reaminti faptul că până acum, *Java* punea la dispoziția programatorilor două ierarhii importante de clase pentru a accesa resurse externe de date (în principal fișiere și date de pe rețea).

Aceste ierarhii sunt ierarhia *InputStream/OutputStream* și ierahia *Reader/Writer*.

Diferența dintre cele două ierarhii este aceea că prima poate fi folosită pentru a transfera orice fel de date indiferent de formatul lor, iar ce-a dea doua este limitată la date tipăribile (text) deoarece se bazează pe un anumit set de caractere care poate fi setat.

Toate operațiile de citire, respectiv scriere, care se pot efectua cu ajutorul claselor care fac parte din una dintre ierarhiile amintite mai sus blochează (*îngheață*) aplicația (mai precis firul de execuție curent) până când se reușește sau nu să se citească/scrie cantitatea de date necesară.

Mai ales pentru aplicațiile orientate pe conexiuni între calculatoare (*client/server*) blocarea firului de execuție curent duce la o degradare excesivă a performanțelor aplicației pe măsură ce mai mulți clienți sunt conectați la același server din mai multe motive pe care nu are rost să le enumerăm aici.

Cel mai important argument este acela că, pe măsură ce numărul de clienți crește, se mărește și numărul firelor de execuție și operațiile de creare și *switch* (interschimbare) a contextelor firelor de execuție sunt foarte costisitoare din punctul de vedere al timpului consumat.

De obicei, un client are atașat pe server propriul lui fir de execuție care se ocupă de transferul datelor înspre/dinspre client.

Până aici toate "bune" și "frumoase", dar haideți să vedem ce aduce în plus *java.nio*. *java.nio* este un pachet de clase inclus în versiunile de *Java* 1.4 și mai noi.

Cred că ați intuit deja, că una dintre facilitățile pe care le aduce *java.nio* este aceea că accesul la resurse externe de date se poate face într-un mod în care să nu se blocheze firul de execuție curent, dar despre această facilitate vom aminti spre sfârșitul articolului, după ce vom prezenta celelalte elemente pe care se bazează aceasta.

Blocarea fișierelor

Deși blocarea fișierelor nu este folosită foarte des, această posibilitate a fost introdusă în noile versiuni ale limbajului *Java*.

Înainte de versiunea 1.4 nu exista nici o metodă, prin care să se poată verifica sau bloca fișierele în cadrul aplicațiilor *Java*.

Blocarea fișierelor se poate face prin intermediul clasei *FileChannel*.

O instanță a clasei *FileChannel* poate fi obținută cu ajutorul claselor *FileInputStream*, *FileOutputStream* sau *RandomAccessFile*.

Un fișier (zonă dintr-un fișier) poate fi blocat pentru acces partajat sau exclusiv.

În general, în cazul în care un proces blochează o resursă în mod partajat, atunci acea resursă mai poate fi blocată în mod partajat de alt proces. În cazul în care un proces blochează o resursă în mod exclusiv, atunci acea resursă nu mai poate fi blocată decât după ce a fost eliberată de primul proces.

Blocarea fișierelor are efect doar în cazul a două sau mai multe procese diferite. În cadrul aceleiași aplicații *Java*, dacă un fișier este blocat în mod exclusiv de un fir de execuție, atunci fișierul mai poate fi blocat în orice mod de către alt fir de execuție.

Această facilitate a fost introdusă pentru a permite integrarea aplicațiilor *Java* cu alte aplicații.

Zone tampon

Una dintre cele mai importante facilități introduse de *java.nio*, constituind și elementul care stă la baza acestui pachet, este introducerea claselor *Buffer* (zonă tampon).

Un *Buffer* reprezintă un tablou unidimensional, cu elemente de tip primitiv, încapsulat într-un obiect.

În *Java buffer*-ele au fost înzestrate cu câteva proprietăți importante: *marcaj*, *poziție curentă*, *limită*, *ca-*

Figura 1: Buffer nou

pacitate și *modificabilitate*. Primele patru proprietăți sunt de tip întreg și verifică relația: 0 ≤ *marcaj* ≤ *poziție curentă* ≤ *limită* ≤ *capacitate*, iar ultima proprietate este de tip logic și ne indică faptul ca putem pune date într-un anumit *buffer*.

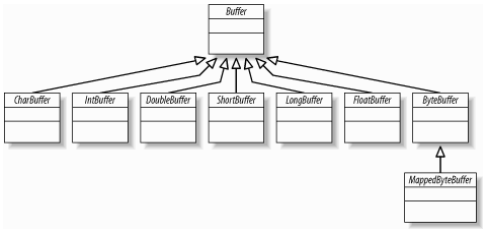

Figura 2: Ierarhia de Buffer-e

Există mai multe tipuri de astfel de zone tampon în *Java*, și anume: *CharBuffer*, *DoubleBuffer*, *FloatBuffer*, *IntBuffer*, *LongBuffer*, *ShortBuffer*, *ByteBuffer* și *MappedByteBuffer*.

În momentul în care este creat un obiect de tipul *Buffer* poziția curentă a acestuia este 0, limita este egală cu capacitatea (dimensiunea) zonei tampon, iar marcajul este nedefinit.

Pentru a pune date într-o astfel de structură se folosesc metodele *put* care sunt specifice fiecărui tip de *Buffer*.

În momentul în care se execută metoda *put*, se depun datele în zona tampon și se incrementează poziția, dacă aceasta nu depășește limita sau se raportează excepția *BufferOverflowException* în caz contrar.

Pentru a citi dintr-o astfel de structură se folosesc metodele *get* specifice fiecărui tip de *Buffer*.

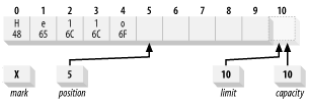

Figura 3: Buffer după un apel put

În momentul în care se execută metoda *get*, se returnează datele cerute și se incrementează poziția. Dacă dimensiunea datelor cerute este mai mare decât elementele rămase necitite din zona tampon, atunci se raportează excepția *BufferUnderflowException*.

Pentru a verifica dacă mai sunt date disponibile în zona tampon, pentru citire, sau mai este spațiu suficient pentru scriere se poate folosi metoda *remaining* care returnează dimensiunea disponibilă, și anume diferența dintre limită și poziția curentă.

În practică, mai întâi se umple o zonă tampon (folosjnd apeluri ale metodelor *put*) și apoi se golește (folosind apeluri ale metodelor *get*).

După ce sunt depuse date în zona tampon, pentru a începe operația de citire, mai trebuie făcută o operație suplimentară, deoarece, după cum se poate observa, metodele *get* incrementează și ele poziția în *buffer*.

Această operație poartă denumirea de *flip*. În urma acestei operații, poziția devine 0, marcajul nedefinit, iar limita va primi valoarea anterioară a poziției.

Acest fapt ne permite ca printr-o succesiune de citiri repetate să accesăm toate datele cuprinse între poziția 0 și noua limită.

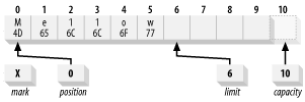

Figura 4: Buffer după flip

O altă operație importantă este operația *reset* în urma căreia se revine pe poziția marcată (dacă există una), adică poziția va primi valoarea marcajului.

Marcajul se poate modifica prin intermediul metodei *mark* care transferă valoarea poziției în marcaj.

O operație asemenătoare operației *reset* este *rewind*. În urma acestei operații poziția devine 0 și marcajul nedefinit.

Cu ajutorul acestor ultime două metode se pot ignora datele citite până la un moment dat sau se poate relua operația de citire, în funcție de necesități.

Am văzut cum se pot citi și scrie date într-o zonă tampon, dar ce se întâmplă dacă dorim să o reutilizăm, deoarece asta se dorește de la o zonă tampon?

Pentru a reutiliza o zonă tampon trebuie să existe o modalitate de a modifica limita și în sus nu doar în jos, cum se poate observa destul de ușor pe baza celor prezentate în secțiunea curentă.

O modalitate de a modifica limita este utilizarea metodei *clear*, care restaurează starea inițială a zonei tampon, starea ulterioară creării acesteia, pierzându-se datele care au rămas necitite sau neprelucrate (cazul unei citiri parțiale a zonei tampon).

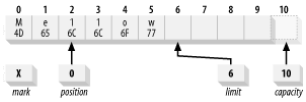

Figura 5: Buffer citit parțial

Pentru a evita acest lucru se poate executa o secvență de operații de tipul *buf.position(buf.limit())*, *buf.limit(buf.capacity())*, unde *buf* este un obiect de tipul *Buffer*, metoda *position* cu un argument setează poziția curentă, metoda *limit* fără argumente returnează valoarea curentă a limitei, metoda *limit* cu un argument setează valoarea limitei, iar metoda *capacity* returnează capacitatea *Buffer-ului*.

Această modalitate de a păstra datele neprocesate în zona tampon este ineficientă deoarece păstrează și datele procesate, și astfel se diminuează spațiul disponibil pentru alte operații de scriere în zona tampon.

Există însă o metodă mult mai eficientă pentru aceasta, și anume folosirea metodei *compact* care mută datele dintre *position* și *limit* la începutul zonei tampon (de la poziția 0), face poziția egală cu cantitatea de date mutate și limita egală cu capacitatea.

În general zonele tampon sunt modificabile, așadar se pot utiliza metodele *put*.

Figura 6: Buffer după compact

Pentru a verifica dacă o zonă tampon poate fi modificată, se poate apela metoda *isReadOnly*.

Un *buffer* care nu poate fi modificat reprezintă un *view* (vizualizare) a unui alt *buffer* de același tip.

Un *view* al unui *buffer* reprezintă un obiect care are același tip ca și *buffer-ul* și care partajează același tablou de elemente primitive.

Prin intermediul unui *view* nu este neapărat necesar să se poată accesa tot *buffer-ul* inițial, ci doar o parte a acestuia care prezintă interes.

Pentru a crea un *buffer* care reprezintă un *view* al întregului *buffer* curent se folosește metoda *duplicate*.

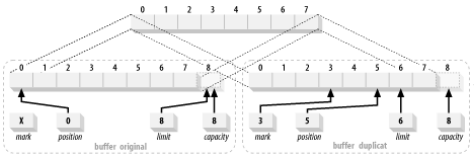

Figura 7: Buffer-e după duplicate

Mai există o metodă similară cu *duplicate*, și anume *slice* în urma căreia *buffer*-ul rezultat este modificabil dacă *buffer*-ul curent este modificabil (rezultă două *buffer*-e care partajează același tablou și care poate fi modificat concurent).

Metoda *slice* returnează un obiect *buffer* modificabil care poate accesa zona dintre poziția și limita actuală a tabloului de date primitive și are deci capacitatea *remaining*.

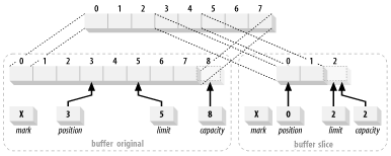

Figura 8: Buffer-e după slice

Zonele tampon pot fi alocate în spațiul de memorie al mașinii virtuale sau în spațiul de memorie al sistemului de operare. Oricare din cele două metode poate fi folosită.

Zonele tampon alocate în afara mașinii virtuale *Java* îmbunătățesc ratele de transfer ale datelor dintre sistemul de operare și mașina virtuală *Java*.

Apare acum întrebarea cum comunicăm cu sistemul de operare prin intermediul acetor zone tampon? Ei bine răspunsul este cu ajutorul canalelor de comunicație pe care le vom prezenta în continuare.

Canale

Un canal reprezintă o conexiune deschisă spre o entitate capabilă să realizeze mai multe operații de intrare și ieșire distincte cum sunt citirea și scrierea.

Astfel de entități sunt *fișierele* și *socket*-urile.

În *Java* canalele operează doar cu *Buffer*-e.

Canalele sunt de mai multe tipuri, dar voi aminti aici pe cele mai importante dintre ele: *FileChannel*, *DatagramChannel*, *SocketChannel* și *ServerSocketChannel*.

Ultimele două tipuri de canale se folosesc pentru comunicația pe rețea iar primul pentru accesarea conținutului fișierelor.

Clasa *FileChannel* poate fi folosită pentru a bloca un fișier sau pentru a construi un tip mai special de zonă tampon și anume *MappedByteBuffer* pentru care tabloul unidimensional este reprezentat de o zonă din memorie în care sunt încărcate date dintr-un fișier și dacă se scrie în acea zonă tampon, datele vor fi scrise până la urmă în fișier, într-un mod care este transparent programatorului, deci zona tampon creată reprezintă o imagine a unui fișier.

Canalele au la rândul lor mai multe proprietăți, dar o voi aminti pe cea care mi se pare mai importantă și anume *selectabilitatea* despre care vom face unele precizări în cele ce urmează.

Citirile și scrierile din și pe canale se pot efectua cu blocarea programului (după cum am amintit la începutul articolului) sau fără blocarea acestuia.

Pentru anumite canale acest comportament poate fi setat, adică pot fi accesate și cu blocare și fără blocare și implicit orice canal care poate fi accesat fără blocare poate fi accesat și cu blocare (folosind eventual cod adițional).

În *Java*, canalele care pot fi accesate fără a bloca programul trebuie derivate din clasa *SelectableChannel* sau din clasa *AbstractSelectableChannel* în funcție de necesități.

Comportamentul de blocare a programului, pentru acest tip de canale poate fi setat prin intermediul metodei *configureBlocking* cu un parametru de tip logic care specifică modul în care se va seta canalul.

Ce înseamnă blocare? Să presupunem că dorim să transmitem prin rețea *n* octeți care au fost stocați într-un vector. Folosind vechea tehnică, cea cu *stream*-uri de ieșire operația de scriere ar fi blocat programul până ar fi fost scriși toți octeții.

Folosind noua tehnică, operația de scriere ar citi dintr-o zonă tampon atâția octeți câți ar putea scrie fără a bloca programul și ar lăsa în zona tampon datele care nu au putut fi scrise. Datele rămase pot fi transmise prin apeluri succesive ale metodei de scriere pe rețea.

În cazul unei citiri din rețea, dacă se folosea blocarea și nu erau disponibile date, atunci programul aștepta până când erau date disponibile spre a fi citite, folosind *java.nio* putem evita foarte ușor acest impediment, noua procedură de citire de pe rețea putând fi configurată să citească atâția octeți câți sunt disponibili astfel încât să nu se depășească dimensiunea zonei tampon.

Să revenim acum la *selectabilitate*. Să luăm tot cazul transmisiei în rețea. Presupunem că avem un server și zece clienți conectați la un moment dat, deci zece canale configurate fără blocare. Prin intermediul unor indicatoare (*flag*-uri) putem decide care clienți au transmis date și putem citi acele date. Aceasta reprezintă *selectabilitatea* și se referă în principiu la citire.

În paragraful anterior, pe lângă selectabilitate mai apare un concept nou și anume *multiplexarea*, concept ilustrat prin citirea datelor de la mai mulți clienți în același fir de execuție, evitând astfel crearea excesivă de fire

de execuție și utilizarea irațională a resurselor sistemului.

Poate fi ilustrată foarte ușor și selectabilitatea la scriere. Canalele selectabile au metode prin care pot fi setate indicatoarele.

Să presupunem ca avem o unitate de program care procesează cererile de la clienți în mod concurent (folosind un *thread pool* - piscină de fire de execuție).

În momentul în care o cerere a fost procesată și a fost elaborat răspunsul, pentru a nu scrie direct pe rețea, deoarece nu este scopul unității de față, acesta este pus într-o structură și canalului corespunzător clientului îi este setat indicatorul de scriere (date disponibile pentru scriere).

Selectoare

Pentru a evita o astfel de implementare a selectabilității (iterația care interoga indicatorul de citire) de la secțiunea anterioară care este ineficientă din mai multe puncte de vedere, pachetul *java.nio* are o implementare mai optimă și anume clasa *Selector*.

Orice canal selectabil, după ce a fost configurat să nu blocheze programul, poate fi înregistrat la unul sau mai multe selectoare.

Fiecare canal are atașată o cheie pentru fiecare selector în parte.

Selectoarele au o metodă numită *select* care returnează numărul de chei canalelor pentru care există indicatoare setate.

Pentru a lua toate cheile atașate canalelor pentru care există indicatoare setate se poate folosi metoda *selectedKeys*.

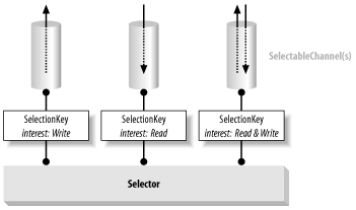

Figura 8: Selector

Metoda *select* blochează până în momentul în care există cel puțin un canal care are indicatoare setate însă există și metoda *selectNow* care revine imediat chiar dacă nu sunt canale disponibile pentru procesare.

Acest mecanism poate fi folosit pentru multiplexare și reducerea resurselor utilizate de sistem.

Ca o proprietate a selectorului din *Java*, acesta reprezintă implementarea șablonului de proiectare *Reactor*.

Pe lângă operațiile de citire și scriere, *Java* oferă suport și pentru alte două indicatoare și anume *acceptare* și *conectare*.

Acceptarea este operația în urma căreia unui client îi este permis să se conecteze la un server, iar această operație are loc pe server.

Conectarea este operația prin care un client a primit răspunsul de acceptare de la server și este deci conectat.

În continuare voi enumera canalele selectabile amintite în secțiunea anterioară și indicatoarele care reprezintă operațiile suportate de aceste clase:

- *ServerSocketChannel*: *OP_ACCEPT*;
- *SocketChannel*: *OP_ACCEPT*, *OP_READ*, *OP_WRITE*;
- *DatagramChannel*: *OP_READ*, *OP_WRITE*.

Seturi de caractere

Operațiile de citire și scriere pe rețea folosesc zone tampon de tipul *ByteBuffer*. Majoritatea programatorilor trebuie să realizeze aplicații în care comunicarea dintre client și server să se facă cu ajutorul șirurilor de caractere.

În această secțiune vom vedea cum se face transformarea din șir de octeți în șir de caractere într-un mod elegant, folosind *java.nio* și diferite seturi de caractere.

Pachetul *java.nio* conține un set de clase care să ne oferă posibilitatea de a transforma conținutul unei zone tampon din șir de octeți în șir de caractere și apoi de a depune rezultatul într-o zonă tampon de tipul *CharBuffer*.

Pentru a putea folosi un set de caractere trebuie creată o instanță a clasei *Charset*.

Acest lucru se realizează prin apelul metodei *Charset.forname* având ca parametru numele setului de caractere care se dorește a fi folosit (de exemplu, *Charset.forname("ASCII")*).

Pentru a putea transforma octeții care au fost citiți de pe un canal în șir de caractere avem nevoie de un decodificator care este constituit în *Java* de clasa *CharsetDecoder*.

Un *CharsetDecoder* poate fi obținut printr-un apel al metodei *newDecoder* asupra *Charset*-ului construit.

Această clasă are o metodă numită *decode* care are trei parametri, și anume: un parametru de tip *ByteBuffer*, care reprezintă zona tampon folosită la citirea de pe rețea (canal), un parametru de tip *CharBuffer*, care reprezintă zona tampon în care se dorește transferul rezultatului și un parametru de tip logic prin care i se indică decodificatorului dacă s-a ajuns la sfârșitul datelor (transmisiei) sau nu.

Metoda *decode* returnează un obiect de tipul *CoderResult* care poate fi interogat pentru a vedea dacă au apărut erori de conversie.

Pentru a transforma în sens invers, adică din șir de caractere stocat într-un *CharBuffer* în șir de octeți avem nevoie de un codificator constituit în *Java* de clasa *CharsetEncoder* care ne pune la dispoziție metodele pereche ale celor oferite de clasa *CharsetDecoder.*

O instanță a lui *CharsetEncoder* se poate obține folosind metoda *newEncoder.*

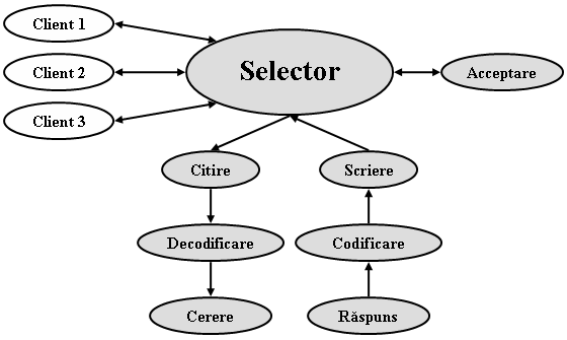

Figura 9: Model de server care folosește java.nio