Persistència en fitxers

Josep Cañellas Bornas

Accés a dades

2.3 Fluxos eficients: Channels i Buffers

L'acrònim *nio* prové de *new input/output*.

Una de les crítiques constants que ha rebut Java en relació amb la biblioteca d'entrada i sortida és la seva baixa eficiència. Des de la versió 1.4 es va introduir un nova biblioteca i es va reestructurar l'antiga per tal de guanyar velocitat en

l'intercanvi de dades en els processos d'entrada i sortida. La biblioteca que dóna major eficiència es troba en el paquet *java.nio* i se sustenta bàsicament sobre dues jerarquies principals, les classes que implementen la interfície `Channel` i la jerarquia que hereta de *Buffer*. La reestructuració de la biblioteca antiga implica només canvis interns: s'han substituït estructures antigues per les classes del paquet *nio* a fi d'incrementar l'eficiència de totes les classes d'entrada i sortida Java.

2.3.1 Conceptes

Channel té el paper de connector a la font de dades, però no en el sentit dels fluxos, sinó més aviat com la porta d'accés al magatzem de dades. La missió d'un *buffer* seria doble. D'una banda, la d'introduir o extreure informació d'un *Channel*, i de l'altra, la de dotar de totes aquelles utilitats necessàries per gestionar còmodament l'intercanvi de dades des del punt de vista del programador (utilitat de conversió dels tipus bàsics, seriació i deseriació d'objectes, suport de diversos formats d'emmagatzematge, gestió del flux, etc.).

La importància del paquet *nio*, però, se sustenta en el fet que tant les implementacions de *Channel* com *Buffer* i els seu derivats utilitzen internament utilitats molt eficients, pròpies de cada sistema operatiu. És a dir, que per exemple la classe `FileChannel` o la classe `ByteBuffer` podrien tenir rendiments diferents en diferents sistemes operatius, ja que cada màquina virtual específica disposa del seu paquet *nio* reescrit i adaptat a les característiques de la plataforma on correrà.

2.3.2 Instanciació d'un Channel

En la versió 1.6 de Java, els Channels no es poden instanciar usant la típica instrucció *new*. Per obtenir un `Channel` és necessari que un altre objecte l'instanciï i ens el retorni en executar algun dels seus mètodes. L'objecte instanciador se sol conèixer amb el nom de *factory* quan la seva funció principal es limita a "fabricar" instàncies.

Es tracta d'una tècnica força usada en programació orientada a objectes, ja sigui per independitzar interfícies o classes abstractes, de les classes finals que les implementin o per restringir i gestionar totes les possibles instàncies generades en una aplicació. En el cas que ens ocupa es compleixen ambdós requisits; desconeixem la classe final, ja que com hem dit en darrer terme depèn del sistema operatiu i, a més, cal evitar, per raons d'eficiència i coherència de dades, que es multipliquin el nombre de canals associats a una mateixa font de dades.

En el context dels fitxers, el paper de *fabricants de canals* és assumit pels objectes de la jerarquia *Stream* Com ja s'ha comentat, la biblioteca *java.io* s'ha reestructurat de manera que internament es fan servir `Channels` específics, concretament

instàncies de la classe `FileChannel`. Des de la versió 1.4, les classes d'entrada i sortida disposen d'un mètode anomenat `getChannel` que retorna el canal usat internament.

Per obtenir un `FileChannel`, doncs, necessitarem sempre la instància de l'*Stream* corresponent, i si no la tinguéssim, hauríem de crear-la. Vegeu tot seguit com obtenir un `FileChannel` a partir d'un `FileInputStream` ja creat:

```
1 FileInputStream istream;
2 ...
3 FileChannel channel = istream.getChannel();
4 ...
```

Vegeu ara com obtenir un `FileChannel` a partir d'un `FileInputStream` que no necessitem:

```
1 FileChannel channel = new FileInputStream(ruta).getChannel();
2 ...
```

Per introduir o treure dades d'un canal ens cal usar un *Buffer*. La introducció es farà executant algun dels mètodes *read*, i l'extracció, usant `write`. A més, `FileChannel` disposa de mètodes per realitzar transferències massives de dades entre canals, de mètodes de posicionament dins el fitxer i de mètodes per gestionar el bloqueig de fitxers.

2.3.3 Instanciació d'un Buffer

ByteBuffer segueix la mateixa tècnica constructora que `FileChannel`, és a dir, no disposa de constructor públic i per tant precisa d'una classe instanciadora que faci el paper de *factory*. En aquest cas, és la pròpia classe `ByteBuffer`la que pren aquest paper per tal de poder servir la classe específica del sistema operatiu amfitrió. Disposa de dos mètodes instanciadors. Són mètodes estàtics anomenats `allocate` i `allocateDirect`.

```
1 ByteBuffer byteBufffer = ByteBuffer.allocate(1024);
2 ...
3 ByteBuffer byteBufffer = ByteBuffer.allocateDirect(1024);
```

La principal diferència entre `allocate` i `allocateDirect` és que el segon genera instàncies que estan molt més lligades al sistema operatiu. En funció del sistema amfitrió, la instància aconseguida amb `allocateDirec` pot arribar a ser molt eficient, però sempre gastarà més recursos de memòria i processament que la instància aconseguida amb *allocate*.

2.3.4 Buffers de bytes

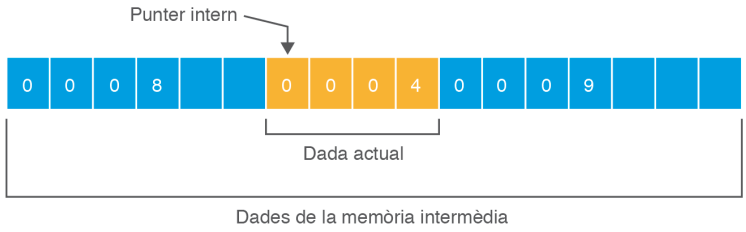
La principal classe de tipus `Buffer` responsable de l'intercanvi és `ByteBuffer`. S'anomena així perquè internament les dades estan representades en bytes. Tot i això, cal dir que aquesta classe ja disposa de mètodes conversors per a tots els tipus bàsics de Java, tant de lectura (`getInt`, `getFloat`, `getDouble` o `getChar`) com d'escriptura (`putInt`, `putFloat`, `putDouble` o `putChar`).

Accés absolut versus accés relatiu al buffer

Tant els mètodes d'escriptura com els de lectura admeten un accés relatiu o absolut. Internament, el *buffer* incorpora un punter a les dades que avança a mida que es llegeix o s'escriu (usant el mètode `getXXX` o `putXXX`), però és possible escriure o llegir una posició determinada del *buffer* indicant-la com a paràmetre. En aquest cas parlem de lectura o escriptura absoluta, ja que l'acció es realitza a la posició indicada. A més, cap de les accions absolutes incrementen el punter intern.

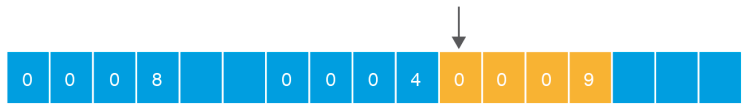
Sigui l'esquema de la figura 2.6, la representació d'un *buffer* amb el punter intern assenyalant un enter (int).

Figura 2.6. Buffer amb el punter intern assenyalant un enter



Si sobre aquest *buffer* executem `getInt()` ens retornarà el valor 4 i mourà el punter intern fins a la següent dada (figura 2.7).

Figura 2.7. Resultat d'executar getInt()



En canvi, si executem `getInt(0)`, ens retornarà el valor 8, però el punter intern no es mourà. De manera que, si seguidament tornem a executar `getInt()`, obtindrem el valor 9.

Quelcom de semblant passa amb els mètodes *put* l'execució de `putDouble(3.5)`: escriuria 8 bytes a partir de la posició del punter intern i el desplaçaria al byte situat a la posició ubicada just després del darrer byte escrit. Per contra, `putDouble(4, 9.5)` escriuria 8 bytes començant en el 5è byte del *buffer*, però no mouria el punter intern.

Vegeu a continuació dos exemples equivalents d'escriptura relativa i absoluta respectivament:

```
...
ByteBuffer byteBuffer = ByteBuffer.**allocate**(MIDA_BYTES);

...
for (int i = 0; i< (MIDA_BYTES/BYTES_PER_INT); i++){
    byteBuffer.**putInt**(i);
}

...
for (int i = 0; i< MIDA_BYTES; i+=BYTES_PER_INT){
    byteBuffer.**putInt**(i, i/BYTES_PER_INT);
}
...
```

2.3.5 Buffers de tipus específics

En cas de treballar amb un únic tipus de dada primitiu, és possible obtenir, a partir d'una instància de `ByteBuffer`, diversos *buffers* específics per a un tipus de dada concret. És a dir, `ByteBuffer` actuaria de `Factory` generant instàncies específiques per treballar amb dades de tipus `char`, `long`, `double`, etc.

```
ByteBuffer bufferOriginal = ByteBuffer.allocate(MIDA_MAXIMA);

...
LongBuffer bufferDeLongs = bufferOriginal.asLongBuffer();

...
CharBuffer bufferDeChars = bufferOriginal.asCharBuffer();

...
DoubleBuffer bufferDeDoubles = bufferOriginal.asDoubleBuffer();

...
```

El principal avantatge de treballar amb *buffers* específics consisteix en el fet que els mètodes d'assignació i obtenció de dades genèriques (*get* i *put*) són específics per al tipus de dada concreta de treball del *buffer*. Així, el mètode `get` d'un `LongBuffer` retornarà *longs*, mentre que el d'un `DoubleBuffer`retornarà *doubles*. De la mateixa manera, el mètode *put* d'un *ShortBuffer* acceptarà només *shorts*, i el d'un `FloatBuffer`, per contra, acceptarà només *floats*.

Malgrat que seria possible crear directament un *buffer* especific usant el mètode instanciador anomenat *allocate*, el més comú és obtenir les instàncies a partir d'un ByteBuffer. La raó és ben senzilla: totes les instàncies generades per un mateix *ByteBuffer* comparteixen les mateixes dades en la mateixa ubicació de memòria, fet que significa que qualsevol canvi en les dades d'algun d'ells repercuteix també en les dades de tots els altres.

Feu la prova. Instancieu tres*buffers*: un de tipus *ByteBuffer*, un segon de tipus *ShortBuffer* i el darrer de tipus *CharBuffer*. Implementeu un algoritme que escrigui, usant el *buffer* de tipus *short,*tots els valors compresos entre *32* i *127*. Seguidament, feu que l'algorisme recuperi els valor assignats, però fent servir el *buffer* de tipus *char*. Observareu que, malgrat que s'hagin entrat valors numèrics,

recuperareu tots els caràcters compresos entre el codi *32* i el codi *127*.

Cal destacar que els *buffers* comparteixen només les dades, no pas els seus estats interns. És a dir, cada *buffer* manté de forma independent la seva posició, les seves marques internes, etc. Vegeu-ho en el següent exemple, una variant de la prova que us hem proposat on es van recuperant de forma independent les dades de cada *buffer*:

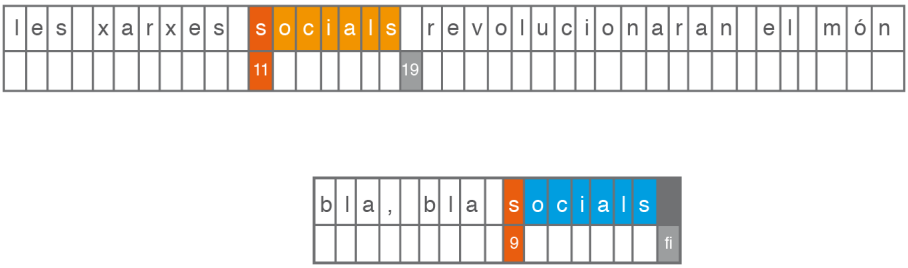
```
ByteBuffer buffer = ByteBuffer.allocate(500);
CharBuffer charBuffer = buffer.asCharBuffer();
ShortBuffer shortBuffer = buffer.asShortBuffer();

//clear, neteja el buffer de qualsevol dada que pogués haver–hi
shortBuffer.clear();

for(short value=32;value<128; value++){
    shortBuffer.put(value); //Assignem cada valor usant put
}

/* flip, marca la posició actual com el limit de dades entrades
 * i endarrereix la posició actual del buffer a la posició
 * inicial. En el nostre cas marcarà la posició 94 com a límit
 * de les dades i es situarà a la posició zero*/
shortBuffer.flip();

/* Com que es tracta de buffers independents cal indicar al
 * buffer de caràcters que el seu limit és el mateix que el del
 * shortBuffer. EL mètode limit, retorna la posició límit si no
 * se li passa cap paràmetre, però assigna com a nou límit, el
 * valor passat per paràmetre. Així, el valor retornat per
 * limit() del shortBuffer es assignat com a limit en el
 * charBuffer. */
charBuffer.limit(shortBuffer.limit());

System.out.println("Cars:");
int i=0;

/* Volem escriure matriu (de 6x32) de correspondència entre cada
 * caràcter i el seu codi, de manera que a les línies senars
 * aparegui el valor del codi i la les línies parells el
 * caràcter corresponent aliniat correctament.*/
while(i<charBuffer.limit()){
    for(int j=0; j<32; j++){
        //shortBuffer avança 32 posicions, però no pas charBuffer
        System.out.printf("%4d", shortBuffer.get());
    }
    System.out.println();
    for(int j=0; j<32; j++){
        //charBuffer avança 32 posicions, però no pas shortBuffer
        System.out.printf("%4c", charBuffer.get());
        i++;
    }
    System.out.println();
}
System.out.println();
```

2.3.6 Treball combinat de Buffers i Channels

Quan es fan treballar conjuntament *buffers* i *channels*, cal tenir en compte que cada un d'ells disposa d'un cursor o punter de posicionament propi i independent. El procés de lectura o escriptura intenta arribar, sempre que sigui possible, fins

al final del *buffer*. Per exemple, mireu la figura 2.8: si la taula més gran (color groc) representés un fitxer amb el seu punter posicionat al byte 11, i la taula més petita (color blau) representés un *buffer* amb el seu punter posicionat al byte 9, el procés de lectura del `Channel`, mètode `read`, aconseguiria escriure, en absència de bloquejos, la paraula "socials" en el *buffer* i omplir-lo totalment des de la posició 9 fins al final. La posició del `FileChannel` avançarà també fins al byte 19.

Figura 2.8. Treball combinat de Buffers i Channels



Cal tenir en compte que els bloquejos dels fitxers poden alterar aquest comportament. De fet, tant el procés de lectura com el d'escriptura s'interrompran en intentar operar en alguna zona bloquejada del fitxer. Per assegurar l'escriptura sencera del *buffer* caldria controlar que el procés arribi al final.

```
public void escriu(ByteBuffer buffer, FileChannel channel) throwsIOException{
    //Mentre no s'hagi escrit tot el contingut del buffer...
    while(buffer.position()<buffer.**limit**()){
        channel.**write**(buffer);
    }
}
```

De forma semblant caldrà controlar també la lectura, però en aquest cas cal tenir en compte que la lectura parcial pot ser deguda al fet que s'ha arribat al final del fitxer. Aprofitarem el fet que el procés de lectura retorna el valor -1 quan s'intenta llegir més enllà del límit del fitxer.

```
public void llegeix(int mida, ByteBuffer buffer, FileChannel channel)
    throws IOException{
    int llegit = 0;
    int totalLLegit=0;
    //Mentre no s'hagi llegit la quantitat desitjada...
    while(llegit!=-1 && totalLLegit<mida){
        llegit = channel.read(buffer);
        totalLLegit+=llegit;
    }
}
```

La utilitat anterior llegirà el contingut del fitxer des de la posició del cursor del `Channel` fins que s'hagin llegit el nombre de bytes indicats pel paràmetre mida, o bé fins que s'hagi arribat al final del fitxer, cas en què la variable llegida prendrà el valor -1.

Cal tenir en compte que `FileChannel` facilita també una versió absoluta del procés de lectura i del d'escriptura. És a dir, una versió on s'ignora la posició del

cursor del Channel, ja que es començarà a operar a partir de la posició indicada per paràmetre.

```
public void escriu(long offsetFitxer, ByteBuffer buffer, FileChannel channel)
    throws IOException{
    int escrit = 0;
    //Mentre no s'hagi escrit tot el contingut del buffer...
    while(escrit<buffer.limit()){
        escrit += channel.write(buffer, offsetFitxer+escrit);
    }
}

public void llegeix(long offsetFitxer, int mida, ByteBuffer buffer,
    FileChannel channel) throws IOException{
    int llegit = 0;
    int totalLLegit=0;
    //Mentre no s'hagi llegit la quantitat desitjada...
    while(llegit!=-1 && totalLLegit<mida){
        llegit = channel.**read**(buffer, offsetFitxer+totalLLegit);
        totalLLegit+=llegit;
    }
}
```

La diferència entre aquests dos mètodes i els dos anteriors és que, en el cas d'*escriu,* aquesta implementació emmagatzema el contingut del *buffer* a la posició indicada per *offsetFitxer* sense canviar la posició interna del cursor del `FileChannel`. En canvi, l'anterior emmagatzemava el contingut a la posició del cursor del `FileChannel` i en acabar, el cursor avançava la quantitat de bytes emmagatzemada.

Quelcom de semblant passa amb *llegeix*. La lectura en la darrera implementació s'inicia a la posició indicada per *offsetFitxer*, i, com és de suposar, el cursor del `FileChannel` no es mou de lloc.

2.3.7 Transferència directa

Encara hi ha una operació més del paquet *nio* que val la pena esmentar. Es tracta de la transferència directa de dades entre *Channels*. Són dues operacions equivalents anomenades `transferFromo` `transferTo`, que aprofiten al màxim la capacitat de còpia massiva dels sistemes operatius. Són operacions en què es realitza una connexió directa a través dels recursos específics del sistema operatiu. Aquests mètodes no fan servir *buffers* gestionats des de Java, sinó que es delega directament sobre les eines externes. Aquestes eines són ideals per realitzar còpies massives entre fitxers quan la màquina virtual corri en sistemes potents. En cas contrari, no se'n treu rendiment.

Acabarem de completar la classe `Utilitats` que hem anat implementant afegint un mètode de còpia massiva. Tant el mètode `transferTo`com `transfetFrom` són sensibles als fitxers bloquejats i poden interrompre la còpia sense haver acabat si es troben un bloqueig durant l'execució. Caldrà assegurar la còpia sencera del fitxer usant un bucle adequat. En el codi podeu veure aquesta instrucció ressaltada.

```
public void copiaFitxersDeBytesNio(File origen, File desti)
```

```
    throws IOException {
    long size;
    long count = 0;
    FileChannel input = null;
    FileChannel output = null;
    try {
        //Obrim el canal origen
        input = new FileInputStream(origen).getChannel();
        //Obrim el canal destí
        output = new FileOutputStream(desti).getChannel();
        //Calculem la mida
        size = input.size();
        //Mentre no s'hagi copiat tot el fitxer...
        while (count < size) {
            count += output.transferFrom(input, 0, size – count);
        }
    } finally {
        //Tancar els canals
        intentarTancar(input);
        intentarTancar(output);
    }
}
```