Programmation Réseau

Java NIO

E/S non-bloquantes

Jean-Baptiste.Yunes@liafa.jussieu.fr

Coloriages: François Armand
armand@informatique.univ-paris-diderot.fr

UFR Informatique

2011-2012

Entrées/Sorties non-bloquantes

On considère en premier chef que les bloquantes :

tenter une telle opération met le demandeur

être réalisée (ou se révèle impossible)

Une entrée/sortie non-bloquante correspond à la situation dans laquelle on souhaite ne pas attendre et faire autre chose, quitte à revenir tenter plus tard

ce mode de fonctionnement est très utile voire nécessaire dans certains cas, par exemple :

Thread puisse lire depuis deux sockets

si la stratégie est bloquante, il est possible Sockets alors que les messages arrivent

les entrées/sorties de java.io.* et java.net.* sont bloquantes

le package java.nio a été créé pour satisfaire ce besoin (entre autres)

mais aussi pour garantir une certaine

Les canaux (Channels)

le premier concept important (ici) de java.nio est celui de Channel. Sans entrer dans les détails, il en existe essentiellement trois types intéressants (toujours pour ce cours) :

DatagramChannel

ServerSocketChannel

SocketChannel

et dont on devine bien les rôles respectifs

aucune de ces classes ne possède de constructeur directement accessible, il faut passer par une méthode usine :

```
DatagramChannel DatagramChannel.open();

ServerSocketChannel ServerSocketChannel.open();

SocketChannel SocketChannel.open();
```

qui créent un objet correspondant

les canaux correspondant sont créés mais pas connectés, il faut donc, si nécessaire, récupérer la Socket sous-jacente et effectuer via :

```
DatagramSocket socket();

ServerSocket socket();

Socket socket();
```

qui renvoient un objet du bon type

pour des raisons pratiques il existe dans chaque type de Sockets une méthode permettant de retrouver le Channel associé existe :

```
*Channel getChannel();
```

alors modifier le caractère bloquant ou non de ces objets via la méthode suivante (héritée de SelectableChannel)

```
SelectableChannel configureBlocking(boolean);
```

on peut aussi tester ce caractère via

```
boolean isBlocking();
```

Les sélecteurs (Selectors)

outre les Channels, le second point important

possibles

Selector)

Selector

Selector();

un sélecteur permet donc de sélectionner un certain

toute opération souhaitée sur un Channel doit être

de la méthode des Channels (et héritée de SelectableChannel)

SelectionKey register(Selector,int);

où :

souhaitées sur le Channel

le retour est la clé de sélection qui pourra être utilisée

les opérations souhaitées doivent être compatibles avec le Channel choisi et sont toujours parmi les 4 suivantes (et qui peuvent être combinées par addition) :

SelectionKey.OP_ACCCEPT,
SelectionKey.OP_READ,
SelectionKey.OP_WRITE,
SelectionKey.OP_CONNECT

On remarquera que chaque type de Channel fournit

int validOps();

une fois le sélecteur configuré, on peut soit :

opérations soit réalisable par appel à

```
int select();
```

des opérations est réalisable par appel à

```
int selectNow();
```

le retour indique le nombre de clés sur lesquelles des opérations sont réalisables

la liste des clés concernées peut être extraite par appel à

Set<SelectionKey> selectedKeys();

un itérateur peut ensuite être employé pour parcourir cette liste afin de retrouver pour chaque clé :

les opérations possibles via

int readyOps();

et le canal concerné via

SelectableChannel channel();

Important : lors du parcours, il est essentiel de penser

connexions entrantes sur deux ports en même temps

```
public static ServerSocketChannel
            createServerChannel(Selector s,int port) {
  try {
    // créer un canal serveur
    ServerSocketChannel ssc = ServerSocketChannel.open();
   // le positionner en mode non bloquant
    ssc.configureBlocking(false);
    // configurer la Socket sous-jacente
    ServerSocket ss = ssc.socket();
    ss.bind(new InetSocketAddress(port));
    // configurer le sélecteur
    ssc.register(s,ssc.validOps());
    return ssc;
  } catch(Exception e) {
    e.printStackTrace();
    System.exit(1);
  }
  return null;
}
```

```
public static void main(String []args) {
  try {
    // Gets a selector
    Selector s = Selector.open();
    // Create the channels
    ServerSocketChannel ssc1 = createServerChannel(s,50123);
    ServerSocketChannel ssc2 = createServerChannel(s,50124);
    // Catch incoming connection requests
    while (true) {
      System.out.println("Waiting for incoming connections");
      s.select();
      Iterator<SelectionKey> it = s.selectedKeys().iterator();
      while (it.hasNext()) {
        // Get one key, and remove it from the set
        SelectionKey sk = it.next(); it.remove();
        // Get the channel
        ServerSocketChannel ssc = (ServerSocketChannel)sk.channel();
        // Do accept();
        Socket sock = ssc.socket().accept();
        // Launch a server with this service Socket
        new Server(sock,ssc.socket().getLocalPort()).start();
      }
    }
  } catch(Exception e) { e.printStackTrace(); System.exit(1); }
}
```

université
PARIS
PARIS DIDEROT