Threads Java 1.5 – Synchronisation Avancée

Arnaud Labourel
Courriel : arnaud.labourel@lif.univ-mrs.fr

Université de Provence

10 février 2011

Architecture Executor

Définition

Un executor est une interface permettant d'exécuter des taches.

```
public interface Executor{
        void execute(Runnable command);
}
```

Un Executor permet de découpler ce qui doit être fait (une tâche définie par un Runnable) de quand et comment le faire explicitement (par un Thread).

Architecture Executor

- Tâches à exécuter
- Pool de threads :
  - réutilisation de threads
  - schémas producteurs/consommateurs
- Contraintes d'exécution:
  - dans quel thread exécuter quelle tâche?
  - dans quel ordre (FIFO,LIFO, par priorité)?
  - combien de tâches peuvent être exécutée concurremment? Combien de Thread au maximum ?
  - combien peuvent être mises en attente?
  - en cas de surcharge, quelle tâche sacrifier?
  - gestion de ressources

Pools I

Méthodes statiques de `Executors`

- `ExecutorService newFixedThreadPool(int n)` renvoie un pool de taille fixée *n*, rempli par ajout de threads jusqu'à atteindre la limite, les threads qui meurent sont remplacés au fur et à mesure.
- `ExecutorService newCachedThreadPool()` : renvoie un pool dynamique qui se vide ou se remplit selon la charge. On crée un nouveau Thread que si aucun thread n'est disponible. Les Thread sans tache pendant une minute sont détruits.

Pools II

Méthodes statiques de `Executors`

- `ExecutorService newSingleThreadExecutor()` renvoie un pool contenant un seul thread ($\Longrightarrow$ séquentialisation), remplacé en cas de mort inattendue.
  La différence avec `newFixedThreadPool(1)` est que le nombre de Threads ne pourra pas être reconfiguré.
- `ExecutorService newScheduledThreadPool(int n)` : renvoie un pool de taille fixée n qui peut exécuter des tâches avec délai ou périodiquement.

`Executor` et Cycle de Vie d'une Tâche

- `void execute(Runnable command)` : soumettre une tâche à l'exécuteur

L'exécution est asynchrone : une tâche est successivement

soumise $\longrightarrow$ en cours $\longrightarrow$ terminée.

`ExecutorService` et Cycle de Vie d'une Tâche

L'`ExecutorService` offre un "service" et propose des méthodes de gestion fines :

- `void shutdown()` : arrêt "propre" : aucune nouvelle tâche ne peut être acceptée mais toutes celles soumises sont exécutées.
- `List<Runnable> shutdownNow()` : tente d'arrêter les tâches en cours (par `InterruptedException`) et renvoie la liste des tâches soumises mais n'ayant pas débutées.

Tâches avec Résultats : `Callable` et `Future`

Un équivalent de `Runnable` mais permettant de renvoyer une valeur.
Une seule méthode pour l'interface générique `Callable<V>`

- `V call() throws Exception`

Pour la soumettre à un `ExecutorService`

- `Future<V> submit(Callable<V> tache)`
  soumettre `tache` à l'exécuteur, le `Future<V>` renvoyé permet de manipuler de manière asynchrone la tâche.

Tâche en Attente : Interface `Future<V>`

Implémentée dans `FutureTask<V>` :

- `V get()`: renvoie la valeur du résultat associée à la tâche en bloquant jusqu'à ce que celui-ci soit disponible,
- `boolean isDone()` : renvoie `true` si la tâche est terminée.
- `boolean cancel(boolean même_en_cours)` : annule la tâche si celle n'a pas encore commencé. Si `même_en_cours`, tente de l'arrêter aussi même si elle a déjà débuté,
- `boolean isCancelled()` : renvoie `true` si la tâche a été annulée.

Rappels: Structure de Données Partagées

- Une structure de données est partagée si elle (ou ses éléments) peut être accédée par plusieurs activités,
- Une structure de données ne peut, en général, pas se trouver dans un état arbitraire :
  Ex: liste chaînée pour représenter un ensemble.

Mutabilité des Variables Partagées

Différents problèmes :

- atomicité: éviter les "interférences" entre exécution des threads conduisant à des objets dans des états aberrants,
- visibilité: la manière dont est vue une variable partagée par plusieurs threads peut ne pas être consistante.

Solutions de base :

- `synchronized` (exclusion mutuelle)$\Longrightarrow$ atomicité et visibilité
- `volatile` $\Longrightarrow$ visibilité

Thread-safe : Sûr dans un environnement multi-processus légers.

Publication des Objets Partagés

Le problème vient qu'il est impossible de verrouiller sur un objet pendant sa construction $\Longrightarrow$ attention aux fuites

- objets incomplètement construits ("publiés" dans le constructeur)
- pas de `start()` dans un constructeur...

Publication des Objets Partagés

```
public class Attention{
  private int n;
  public Attention{int n){
    this.n = n;
  }
  public void assertSanity(){
    if ( n != n )
      throws new AssertionError
                  ("Quelle horreur!");
  }
}
```

Immutabilité

Un objet est *immutable* s'il n'est jamais modifié après sa construction.
Rappel : mot-clé `final`

Un objet immutable est toujours thread-safe.

Confinement

Le *confinement* consiste à ne laisser accessible un objet que par un seul thread.

La plupart des bibliothèques d'interface graphique utilisent le confinement de threads (un seul thread gère les événements graphiques).

Outils pour la Concurrence

- structures de données classiques
  - à protéger
  - déjà *synchronizés*
  - optimisées pour la concurrence
- objets atomiques
- primitives de synchronisation

Préférez les structures de données existantes.

... et seule la javadoc fait foi.

Collections Non Synchronizées

A protéger avec `synchronized(objet)`!

- Interface `List<E>` : listes
  - `ArrayList<E>`
  - `LinkedList<E>`
  - `CopyOnWriteArrayList<E>` : threadsafe mais modifications très coûteuses.
- Interface `Set<E>` : ensembles
  - `HashSet<E>`
  - `TreeSet<E>`
  - `CopyOnWriteArraySet<E>` : threadsafe mais modifications très coûteuses.
  - attention si les éléments sont mutables...

Collections Non Synchronizées

A protéger avec `synchronized(objet)`!

- Interface `Map<K,V>`: table d'association clef-valeur.
  - `HashMap<K,V>`
  - `TreeMap<K,V>` ordonné (arbre rouge-noir)

Collections Synchronisées I

L'implémentation de ces objets utilisent des verrous (implicites).

- `Vector<E>`: "tableau dynamique"
  - `boolean add(E elt)`: ajoute `elt` en fin de tableau,
  - `boolean add(int i,E elt)`: ajoute `elt` en position *i* du tableau,
  - `E get(int index)`: renvoie l'élément en position *i* du tableau,
  - `boolean remove(E elt)`: retire la première occurence de `elt`, retourne vrai s'il y en a une,
  - `E remove(int i)`: retire l'élément en position *i* et retourne celui-ci.

Collections Synchronisées II

- `Stack<E> extends Vector<E>`: pile
  - `E peek()`: renvoie le premier élément de la pile sans l'enlever,
  - `E pop()`: renvoie le premier élément de la pile et l'enlève de celle-ci,
  - `E push(E elt)`: met `elt` sur la pile et retourne `elt`
- `Hashtable<K,V>`: table de hachage
  - `V get(Object clef)`: renvoie l'objet indexé par `clef`,
  - `V put(K clef, V valeur)`: associer `valeur` à `clef` dans la table.

Itérateurs et Accès Concurrents

Depuis Java 1.5, il est possible d'utiliser des "itérateurs":

```
Vector<E> liste;

for(E elt: liste)
  elt.traitement();
```

Les objets et méthodes précédentes sont à "défaillance rapide": si une modification simultanée est détectée lors de l'utilisation d'un itérateur, l'exception `ConcurrentModificationException` est reçue.
$\Longrightarrow$ à utiliser uniquement pour détecter les problèmes, pas un mécanisme de gestion de la concurrence.

java.util.concurrent

Il est à la fois pénible et source d'erreurs de recoder des primitives de synchronisation de haut niveau. $\Longrightarrow$ `java.util.concurrent`

- Collections spécialisées: `BlockingQueue`
- Objets threadsafe mais sans verrous
- Implémentation efficace de variables atomiques
- Outils de synchronisation
- Verrous (avec timeout)
- Granularité temporelle à la nanoseconde `System.nanotime`
- Environnement d'ordonnancement de tâches : `Executor` et `Callable`

Interface `BlockingQueue<E>` I

Implémentations de files d'attente *bloquantes* (ie les accès bloquent si la file est vide ou pleine)

Rappel: bloquant $\Longrightarrow$

```
throws InterruptedException
```

Interface `BlockingQueue<E>` II

- `ArrayBlockingQueue<E>` file de taille bornée
  - `ArrayBlockingQueue(int capacite)` construit une file de taile `capacite`,
  - `ArrayBlockingQueue(int capacite, boolean fair)` construit une file de taile `capacite` et équitable : les accès sont FIFO,
  - `void put(E elt)` ajoute `elt` à la file, en attendant si celle-ci est pleine,
  - `E take()` renvoie le premier élément de la file et le retire, en bloquant tant que la file est vide,
  - `E poll(long delai, TimeUnit unit)` renvoie le premier élément de la file et le retire, en attendant au plus *delai* `unit`,
  - `offer(E elt)` ajoute `elt` à la file, en retournant immédiatement sans ajouter si la file est pleine.

Interface BlockingQueue<E> III

- LinkedBlockingQueue<E> file optionnellement bornée
  - LinkedBlockingQueue() crée une file d'attente (de taille maximale Integer.MAX_VALUE),
  - LinkedBlockingQueue(int capacite) crée une file d'attente de taille maximale capacite éléments,
  - et méthodes identiques.
- SynchronousQueue<E> file de capacité nulle $\Longrightarrow$ rendez-vous.
  - SynchronousQueue()
  - SynchronousQueue(boolean fair), avec accès FIFO si fair est vrai,
  - même méthodes (certaines sont trivialisées).

NB: objets n'acceptant pas l'élément null

ConcurrentLinkedQueue<E>

Une file d'attente non bornée, threadsafe et sans attente.

- `boolean add(E elt)` ajoute `elt` et renvoie `true`,
- `E poll()` retourne et enlève le premier élément de la file,
- `E peek()` retourne sans enlever le premier élément de la file.

Très appropriée à un environnement multi-threads.

NB: `size()` est de complexité non constante mais $O(n)$.

`ConcurrentHashMap<K,V>` I

Table de hachage pour accès concurrents:

- threadsafe,
- sans verrous (bon débit),
- impossible de verrouiller (`synchronized`) l'accès à toute la table,
- pas de `ConcurrentModificationException` avec les itérateurs
- même si ceux-ci sont prévus pour être mono-threads

ConcurrentHashMap<K,V> II

- `ConcurrentHashMap(int capInitiale, float facteurCharge, int niveauConcurrence)` crée une table de capacité initiale `capInitiale`, avec `facteurCharge` de paramètre de rééquilibrage interne dynamique, et pour environ `niveauConcurrence` threads modifiant la table simultanément.
  les rééquilibrages internes sont coûteux.

Remplace très avantageusement `Hashtable` en général.

Que Choisir?

Objets synchronizés (bloquants ou potentiellement bloquants) vs concurrents (non bloquants)??
Paramètres

- débit de traitement
- taux de concurrence
- type de traitement
  - intensif en E/S
  - intensif en CPU
- prévisibilité de comportement

$\Longrightarrow$ Bien lire les documentations et bien modéliser le problème (combien de threads modifient la structure, combien ne font que lire, problèmes de débit ...)

Sémaphore

Définition

Un sémaphore est un objet de haut niveau permettant de gérer l'accès concurrent à une ressource partagée, qui accepte au plus n accès concurrents.

Constructeurs :

- `Semaphore(int tickets)` : déclare un sémaphore `tickets`-aire.
- `Semaphore(int permits, boolean fair)` : déclare un sémaphore `tickets`-aire, avec attente (presque) FIFO si `fair` est `true`.

Accès au Sémaphore

- `void acquire()` : bloque jusqu'à ce que moins de n threads possèdent un "ticket" du sémaphore.
- `void acquire(int tickets)` : bloque jusqu'à ce que moins de n threads possèdent un "ticket" du sémaphore et consomme `tickets` tickets de sémaphore.

Attention : les threads peuvent également être débloqués par une `InterruptedException`.

- `void release()` libère un ticket,
- `void release(int tickets)` libère `tickets` tickets.

Un exemple d'utilisation

```
try{
   sem.acquire();
   // Utilisation de la ressource
   // protégée par sem
     ...
}
catch(InterruptedException e){
   ...
}
finally{ // toujours éxecuté
   sem.release();
}
```

Autres Primitives

Par tentative : l'appel n'est plus bloquant et la valeur retournée est `true` si l'accès est possible.

- `boolean tryAcquire()`
- `boolean tryAcquire(int tickets)`
- `boolean tryAcquire(long timeout, TimeUnit unit)`
- `boolean tryAcquire(int tickets, long timeout,TimeUnit unit)`

  Attention : Interruption possible par `InterruptedException`.

Acquisition sans pouvoir être interrompus :

- `acquireUninterruptibly()`
- `acquireUninterruptibly(int tickets)`

Variables Atomiques

C'est une généralisation de `volatile` :

- `AtomicBoolean`,
- `AtomicInteger`,
- `AtomicLong`,
- `AtomicReference`,
- `AtomicLongArray`,
- `AtomicReferenceArray`.

Accès aux Variables

L'accès est thread-safe et sans étreintes fatales

- `get` : renvoie la valeur,
- `set` : affecte la valeur,

Ces opérations utilisent des instructions de très bas niveau et sont donc sujettes à variations selon les plateformes (et notamment ne sont pas, à strictement parler, non bloquantes).

Opérations Particulières I

Il s'agit d'opérations combinant, de manière atomique, un test et une affectation éventuelle (ici, dans le cas d'un `AtomicLong`):

- `boolean compareAndSet(long expect, long update)` : teste l'égalité de la valeur avec `expect` et affecte `update` dans le cas positif. Revient à lire puis écrire un `volatile`
- `boolean weakCompareAndSet(long expect, long update)` : idem, sauf que l'accès n'est séquentialisé que cet objet

Opérations Particulières II

- `long getAndSet(long newValue)` : affecte à la nouvelle valeur et renvoie l'ancienne
- `long getAndIncrement()` : incrémente de manière atomique
- `long getAndDecrement()` : décrémente de manière atomique
- `long incrementAndGet()` : incrémente de manière atomique
- `long decrementAndGet()` : décrémente de manière atomique
- `long getAndAdd(long delta)` : ajoute `delta` atomiquement

La Classe `CyclicBarrier`

Une barrière est un système permettant de bloquer un thread en un *point* jusqu'à ce qu'un nombre prédéfini de threads atteigne également ce point.
La barrière est cyclique car elle est "réutilisable".

Pour une barrière utilisée une seule fois, préférer `CountDownLatch`.

- `CyclicBarrier(int parties)` : barrière pour `parties` threads,
- `CyclicBarrier(int parties, Runnable barrierAction)` : barrière pour `parties` threads, `barrierAction` est exécutée avant de lever la barrière.

Méthode `await`

- `int await()` : attendre que tous les threads aient atteint la barrière ou que
  - l'un des threads en attente soit interrompu
  - l'un des threads en attente atteigne son délai d'attente
  - la barriere soit réinitialisée
- `int await(long timeout, TimeUnit unite)` : idem avec un délai d'attente de `timeout`
- **valeur de retour** : ordre inverse d'arrivée à la barrière, c'est-à-dire le nombre de threads restant à attendre.

Méthode `reset`

- `int getNumberWaiting()` : renvoie le nombre de threads en attente sur la barrière.
  $\Longrightarrow$ pour déboguer
- `void reset()` : réinitialise la barrière (les éventuels threads en attente reçoivent `BrokenBarrierException`)

Si une barrière est interrompue autrement que par `reset()` et `BrokenBarrierException`,il est difficile de la réinitialiser correctement.

Dans ce cas, il est conseillé de créer une *nouvelle barrière*.

Utilisation d'une barrière

```
// une seule fois
CyclicBarrier barriere =
    new CyclicBarrier(N);
```

```
// pour chaque thread
    ...
try {
  barriere.await();
} catch (InterruptedException ex) {
  return;
} catch (BrokenBarrierException ex) {
  return;
}
```

```
// réinitialisation de la barrière
barriere.reset();
```

CountDownLatch

Attente sur des événements (“compte à rebours”) et non plus sur l'arrivée d'autres threads.

- `CountDownLatch(int n)` crée une barrière de *n* éléments,
- `void await()` attendre que le compte à rebours soit terminé,
- `void await(long delai, TimeUnit unit)` attendre au plus *delai* `unit` que le “compte à rebours” soit terminé,
- `void countdown()` décrémente le compteur. Si 0 est atteint, tous les threads bloqués sont libérés,
- `long getCount()` renvoie l'état du compteur.