Programmation Répartie : Gestion des threads en Java Executor et Fork/Join

1 Executor

2 Fork/Join

Combien des threads ?

- Autant de threads que de tâches
  - Chaque tâche ou sous tâche donne lieu à un thread
  - Créer autant de threads que l'on souhaite
- Nombre de thread fixe

Problèmes possibles

- Autant de threads que de tâches
  - La création de thread a un coût
  - Pour des petites tâches, le coût de création d'un thread est supérieur au coût d'exécution
  - La création d'un trop grand nombre de thread peut ralentir le programme
- Nombre de thread fixe
  - Difficile de gérer les problèmes où le nombre de tâches est dynamique

Exemple

Parallélisation d'un serveur web ?

- Chaque requête est traitée par un nouveau thread
- Quels sont les problèmes ?

Exemple : serveur http

Parallélisation d'un serveur web : problèmes

- Serveur http qui traite chaque requête par un nouveau thread
- Si le serveur reçoit plus de requête qu'il ne peut traiter immédiatement, le système ne répondra plus à cause du surcoût lié aux threads

Avec un ensemble fixe de threads, elle ne traitera pas plus vite, mais les traitera aussi vite que le système peut. Ces threads devront pouvoir s'occuper de plusieurs tâches successivement.

1 Executor

2 Fork/Join

Executor

Executor

- La framework `Executor` répond à ces problématiques
- Pas de création manuelle de thread
- Chaque `Executor` possède un **ensemble de thread**, qui accepte des **tâches** à exécuter. Un thread pourra exécuter plusieurs tâches différentes successivement.
- Les tâches sont définies à l'aide du type `Runnable`
- Différentiation entre la tâche à réaliser, et ce qui l'exécute

Définition en Java

Executor

```
public interface Executor {

    void execute(Runnable command);
}
```

Définition en Java

ExecutorService : Extension de la première interface

```
public interface ExecutorService extends Executor {

    // Job submission
    public void submit(Runnable job);

    // Lifecycle management
    public void shutdown();
    public void shutdownNow();
    public boolean isShutdown();
    public boolean awaitTermination
                        (long timeout, TimeUnit unit);
    // (...)

}
```

Différence pour l'exécution de thread

Soit une tâche `r` de type `Runnable`

Sans `Executor`

- (new Thread(r)).start() ;
- Crée un thread et le démarre

Avec `Executor e`

- e.execute(r) ;
- La tâche est donnée à un thread disponible dans `Executor e`

Instanciation via la factory **Executors**

Factory `Executors`

- `Executors` : factory pour instancier des objets de type `Executor`
- Il existe plusieurs implantations d'`ExecutorService` qui diffèrent suivant le comportement de l'ensemble des threads.

`https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html`

Implantations d'`ExecutorService`

`newCachedThreadPool()`

- Toute nouvelle tâche est donnée à un thread présent dans le pool, s'il est libre
- S'il n'y a pas de thread libre, un nouveau thread est créé
- Les threads inactifs plus de 60s sont détruits

Implantations d'`ExecutorService`

`newFixedThreadPool(int nThreads)`

- Fixe un nombre maximal de threads à la construction
- Il est possible qu'une tâche confiée à cette réserve ne puisse pas être exécutée immédiatement
- Une file d'attente gère les tâches non encore attribuées à un thread.

`newSingleThreadExecutor()`

- Similaire au précédent, mais avec un seul thread possible

exemple

```
import java.util.concurrent.*;
public class NewExecutorService {

    public static void main(String[] args) throws InterruptedException{

        Runnable job = new Runnable() {
            public void run() {
            System.out.println("thread : "
                                + Thread.currentThread().getName());

            }
        };

        // Pool avec 4 threads
        ExecutorService pool = Executors.newFixedThreadPool(4);
        pool.submit(job);
        pool.submit(job);
        pool.shutdown();
        pool.awaitTermination(Long.MAX_VALUE,TimeUnit.SECONDS);

    }

}
```

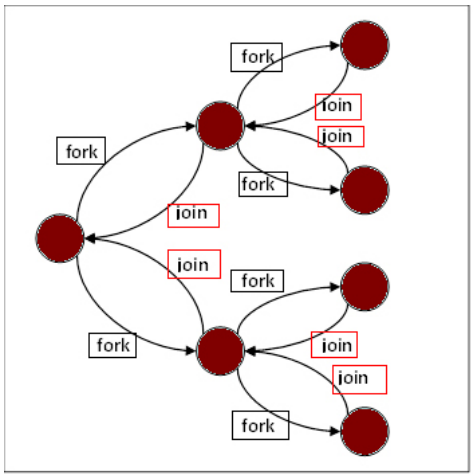
1 Executor

2 Fork/Join

Fork/Join

- Traiter un grand nombre de données
- Les tâches à réaliser sont découpées **récursivement** en paquets
- Chaque paquet est traité indépendemment, et fournit un résultat partiel
- Ces résultats intermédiaires sont regroupés, pour fournir le résultat global

Modélisation très fréquente en parallélisme



Implantation en Java

Implantation

- Implantation de `ExectorService`
- Distribue les tâches aux threads appartenant à un ensemble de threads
- Utilise le vol de tâches

Vol de tâches

- Chaque thread à une pile de tâches à accomplir. Des tâches peuvent générer d'autres tâches.
- Lorsqu'un thread a terminé son ensemble de tâches, il peut récupérer (voler) des tâches d'un autre thread

Modélisation d'un problème avec Fork/Join

Algorithme

```
Si le travail est assez petit
  exécuter le travail directement
Sinon
  diviser le travail en  sous-tâches
  démarrer les nouvelles tâches
  attendre le résultat
```

Syntaxe en Java

Objets java

- `ForkJoinTask` : tâche unique, envoyée au pool de thread. Cette tâche peut elle même générer d'autres tâches du même type. C'est une classe abstraite. Deux classes concrètes en héritent : `RecursiveTask` et `RecursiveAction`
- `ForkJoinPool` : gère l'ensemble des threads. Elle reçoit les tâches, et attribue la tâche à un thread grâce à la méthode `invoke`. Généralement seule la première tâche est donnée au pool, le reste est ajouté par les tâches.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html

`ForkJoinTask`

- Les nouvelles tâches sont créées par les tâches elle même. La méthode `fork()` ajoute une nouvelle tâche au pool de la tâche parente

RecursiveTask

Classe représentant une tâche retournant un résultat. Le type de ce résultat est la paramètre générique de la classe

- `class T extends RecursiveTask<Integer>`
- `compute` : La méthode à redéfinir pour effectuer le calcul. Le type de retour est le type générique
- `join` : permet de récupérer le résultat d'un calcul d'une tâche

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RecursiveTask.html

`RecursiveAction`

Classe représentant une tâche ne retournant pas de résultat.

- `class T extends RecursiveAction`
- `compute` : méthode à redéfinir, qui effectue le calcul.

`https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RecursiveAction.html`

Fibonacci

```
import java.util.concurrent.RecursiveTask;

class Fibonacci extends RecursiveTask<Integer> {
  final int n;
  Fibonacci(int n) { this.n = n; }
  public Integer compute() {
    if (n <= 1)
       return n;
    Fibonacci f1 = new Fibonacci(n - 1);
    f1.fork();
    Fibonacci f2 = new Fibonacci(n - 2);
    return f2.compute() + f1.join();
  }
}
```

Fibonacci

```
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args){
        ForkJoinPool pool = new ForkJoinPool();
        Fibonacci fibo = new Fibonacci(10) ;
        System.out.println("Fibonnacci 10 "+pool.invoke(fibo));
    }



}
```

Generation matrices : Tâche

```
public class MatrixRandomValueTask extends RecursiveAction {
    private Matrix m ;
    private int i0, iF, j0, jF ;
    // methode imposee par RecursiveAction
    public void compute() {
    // la methode shouldSplit() decide si la sous-matrice courante doit
    // encore etre divisee ou pas
        if (shouldSplit()) {
            // ces methodes realisent la division
            int iLim = subHeight(m, i0) ;
            int jLim = subWidth(m, j0) ;
            // creation de quatre sous-taches, une par sous-matrice
            MatrixRandomValueTask subTask11 =
                new MatrixRandomValueTask(m, i0, iLim, j0, jLim) ;
            MatrixRandomValueTask subTask21 =
                new MatrixRandomValueTask(m, iLim, iF, j0, jLim) ;
            MatrixRandomValueTask subTask12 =
                new MatrixRandomValueTask(m, i0, iLim, jLim, jF) ;
            MatrixRandomValueTask subTask22 =
                new MatrixRandomValueTask(m, iLim, iF, jLim, jF) ;

            // lancement de chaque tache
            subTask21.fork() ;
            subTask12.fork() ;
            subTask22.fork() ;
            // execution du calcul pour cette tache
            subTask11.process() ;
        } else {
            // traitement de la sous-matrice dans laquelle on se trouve
            process() ;
        }
```

Generation matrices : Tâche

```
    public void process() {

        for (int i = i0 ; i < iF ; i++) {
            for (int j = j0 ; j < jF ; j++) {
                m.line(i)[j] = rand.nextDouble() ;
            }
        }
    }

    public boolean shouldSplit(){
        return (iF -i0> 5);
    }

    public int subHeight(){
        return (m.height - i0)/2;
    }
    public int subWidth(){
        return (m.width - j0)/2;
    }
}
```

Generation matrices : Main

```
import java.util.concurrent.*;

public class Matrix {

    public static Matrix randomMatrix(int height, int width) {
        ForkJoinPool pool = new ForkJoinPool();
        Matrix m = new Matrix(height, width);
        MatrixRandomValueTask task = new MatrixRandomValueTask(m);
        pool.invoke(task);
        return m;
    }

    public static void main(String[] args){
        randomMatrix(1000,1000);
    }



}
```

Conclusion

Nouveau moyen d'exécuter des threads

- Les threads ne sont plus explicitement gérés par le programmeur
- Définition de tâches
- Plusieurs comportements possibles

Liens

- Tutoriel Java sur la concurrence
  `https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html`
  - Executors `https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html`
  - Fork/Join `https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html`
- Java Concurrent Animated
  `http://sourceforge.net/projects/javaconcurrenta/`