Programmation réseau et concurrente

Benoît Barbot

Département informatique, Université Paris-Est Créteil, M1

Mardi 15 mars 2016, Cours 6 : Gestion des Threads de haut niveau

Plan

Corrigé TP6

Projet

- Rendu – vendredi 8 avril sur Eprel
- Soutenance – vendredi 15 avril

Motivation

Scénario d'utilisation des Threads

- Gérer une tâche indépendante
  ex : servir une page web)
- Lancer une tâche indépendante, résultat attendu
  ex : Télécharger un fichier
- Découper une tache et effectuer un traitement en parallèle
  ex : traitement d'image

Motivation

Scénario d'utilisation des Threads

- Gérer une tâche indépendante
  ex : servir une page web)
- Lancer une tâche indépendante, résultat attendu
  ex : Télécharger un fichier
- Découper une tache et effectuer un traitement en parallèle
  ex : traitement d'image

Limitation de l'interface Runnable

Pas de contrôle sur le résultat du code exécuté par le Thread.

```
public void run();
```

Motivation2

Cout de fabrication d'un thread

Fabrication et destruction des Thread couteuses en temps et en mémoire ($\approx$ 5000 cycles)

Motivation2

Cout de fabrication d'un thread

Fabrication et destruction des Thread couteuses en temps et en mémoire ($\approx$ 5000 cycles)

Thread Concurrent

Inutile d'avoir plus de Threads actifs que de processeur

Solution

Pool de thread

- Fabriquer une fois un ensemble de Threads.
- Créer une file d'attente concurrente de travaux à exécuter.
- Les jobs sont lancés sur les threads libres

Solution

Pool de thread

- Fabriquer une fois un ensemble de Threads.
- Créer une file d'attente concurrente de travaux à exécuter.
- Les jobs sont lancés sur les threads libres

Avantage

- Le coût de fabrication/destructions des Threads n'est payé qu'une fois
- Les travaux en cours ne sont pas affectés par de nouveaux travaux

Solution

Pool de thread

- Fabriquer une fois un ensemble de Threads.
- Créer une file d'attente concurrente de travaux à exécuter.
- Les jobs sont lancés sur les threads libres

Avantage

- Le coût de fabrication/destructions des Threads n'est payé qu'une fois
- Les travaux en cours ne sont pas affectés par de nouveaux travaux

Interface Executor

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

Solution

Pool de thread

- Fabriquer une fois un ensemble de Threads.
- Créer une file d'attente concurrente de travaux à exécuter.
- Les jobs sont lancés sur les threads libres

Avantage

- Le coût de fabrication/destructions des Threads n'est payé qu'une fois
- Les travaux en cours ne sont pas affectés par de nouveaux travaux

Interface Executor

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

Demo : Fabriquer Executor

Implémentation Java dans java.util.concurrent

factory :

- Executors.newFixedThreadPool(n)
  Construis et maintiens n Threads
- Executors.newCachedThreadPool()
  Gestion dynamique des threads
- Executors.newWorkStealingPool()
  Une file par Threads, #Thread = #Processeur
- Executors.newSingleThreadExecutor()
  Garantis l'absence de concurrence

Implémentation Java dans java.util.concurrent

factory :

- Executors.newFixedThreadPool(n)
  Construis et maintiens n Threads
- Executors.newCachedThreadPool()
  Gestion dynamique des threads
- Executors.newWorkStealingPool()
  Une file par Threads, #Thread = #Processeur
- Executors.newSingleThreadExecutor()
  Garantis l'absence de concurrence

Demo

Callable et Future

Callable

```
public interface Callable<V> {
    V call() throws Exception;
} //  Runnable ≈ Callable<Void>
```

Callable et Future

Callable

```
public interface Callable<V> {
    V call() throws Exception;
} //  Runnable ≈ Callable<Void>
```

Future

```
public interface Future<V> {
  boolean cancel(boolean mayInterruptIfRunning);
  boolean isCancelled();
  boolean isDone();
  V get()
   throws InterruptedException, ExecutionException;
  V get(long timeout, TimeUnit unit)
   throws InterruptedException, ExecutionException,
          TimeoutException;
}
```

ExecutorService

```
public interface ExecutorService extends Executor{
  <T> Future<T> submit(Callable<T> task);
   void shutdown();
   List<Runnable> shutdownNow();
   boolean awaitTermination(long timeout,
     TimeUnit unit) throws InterruptedException;
   ...
}
```

ExecutorService

```
public interface ExecutorService extends Executor{
  <T> Future<T> submit(Callable<T> task);
   void shutdown();
   List<Runnable> shutdownNow();
   boolean awaitTermination(long timeout,
     TimeUnit unit) throws InterruptedException;
   ...
}
```

Propriétés

- File de Callable
- Gestion des fins de taches
- Toutes les factory de java.util.concurrent.Executors sont des ExecutorService

ScheduledExecutor

Interface

```
public interface ScheduledExecutorService extends
    ExecutorService {
  public <V> ScheduledFuture<V> schedule(Callable<V> callable,
    long delay, TimeUnit unit);
  public ScheduledFuture<?> scheduleAtFixedRate(
    Runnable command,
    long initialDelay,
    long period,
    TimeUnit unit);
    ...
}
```

ScheduledExecutor

Interface

```
public interface ScheduledExecutorService extends
    ExecutorService {
  public <V> ScheduledFuture<V> schedule(Callable<V> callable,
    long delay, TimeUnit unit);
  public ScheduledFuture<?> scheduleAtFixedRate(
    Runnable command,
    long initialDelay,
    long period,
    TimeUnit unit);
    ...
}
```

Propriété

- Invoque une tache après un temps donné
- Invoque une tache de manière répétée.

ForkJoin

Schéma de calcul récursif

- Le calcul est suffisamment petit pour être exécuté.
- Le calcul est découpé en plusieurs morceaux(Fork) les résultats sont rassemblés (Join).

ForkJoin

Schéma de calcul récursif

- Le calcul est suffisamment petit pour être exécuté.
- Le calcul est découpé en plusieurs morceaux(Fork) les résultats sont rassemblés (Join).

Plusieurs implémentations

- RecursiveAction
- RecursiveTask<V>
- ....

ForkJoin Exemple

```
public class TestForkJoin extends RecursiveTask<String> {
  private int n;
  TestForkJoin(int n){ this.n=n; }

  @Override
  protected String compute() {
    if(n<=4) { return ""+n;}
    else {
      TestForkJoin f1= new TestForkJoin(n/2);
      f1.fork();
      TestForkJoin f2= new TestForkJoin(n - n/2);
      try {
        return "<" + f1.get() + "|" + f2.compute() +">";
      } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
        return null;
      }
    }
  }
  public static void main(String[] args)
      throws InterruptedException, ExecutionException {
    ForkJoinPool fjp = new ForkJoinPool();
    TestForkJoin tfj = new TestForkJoin(100);
    fjp.invoke(tfj);
    System.out.println(tfj.get());
  }
}
```

Stream Parallèle

Parallèle Map, Parallèle reduce

```
<R> Stream<R> map(
   Function<? super T, ? extends R> mapper);
<U> U reduce(
   U identity,
   BiFunction<U, ? super T, U> accumulator,
   BinaryOperator<U> combiner);
```

Stream Parallèle

Parallèle Map, Parallèle reduce

```
<R> Stream<R> map(
   Function<? super T, ? extends R> mapper);
<U> U reduce(
   U identity,
   BiFunction<U, ? super T, U> accumulator,
   BinaryOperator<U> combiner);
```

Condition

```
combiner.apply(u, accumulator.apply(identity, t))
  == accumulator.apply(u, t)
```

Stream Parallèle

Demo

```
String test =
  Stream.iterate(0,x -> x+1)
  .limit(10)
  .parallel()
  .map( x -> x+1 )
  .reduce("",
    (s,i)-> s+","+i,
    (s,t)-> "<"+s+"|"+t+">" );
```