TP 4 : Mini-présentation --- Executors, Thread Pools, Fork / Join

LIU Yuanyuan && GUO Xiaoqing

1. Introduction Executors, Thread Pool, Fork / Join

Framework Executor ou Executors

Java 5 propose le framework Executor pour la gestion abstraite des threads, à cause des inconvénients liés au fait que la classe Thread, par exemple, aucun pool de threads n'est proposé en standard. Donc c'est mieux que de remplacer la classe utilisant le framework Executor qui peut créer et gérer les threads.

Les fonctionnalités de Executors:

- La création des threads, il fournit diverses méthodes pour créer des threads, le plus utilisé est Thread Pools qui peut être utilisé à exécuter des tâches simultanément;
- La gestion des threads, il gère le cycle de vie des threads dans Thread Pools;
- La soumission et l'exécution des tâches, il y a des méthodes pour soumettre des tâches qui seront exécutées dans Thread Pools, il permet aussi d'exécuter les tâches périodiques ou planifiées.

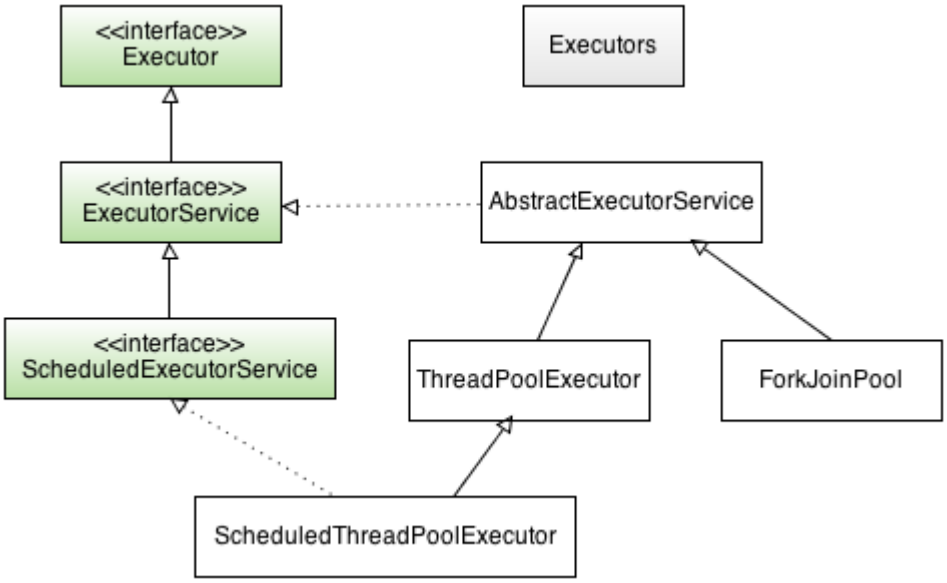
Les interfaces du framework Executor:

- Executor peut exécuter des "Runnables", c-à-d que l'exécution différée de tâches implémentées sous la forme de Runnable;
- ExecutorService est un service d'exécution des tâches "Runnables" et "Callables";
- ScheduledExecutorService permet de planifier l'exécution d'une tâche après un certain délai ou son exécution répétée avec un intervalle de temps fixe, les tâches exécutées dans 10 secondes par exemple.

Les classes d'implémentations:

- ThreadPoolExecutor, l'implémentation du pool de threads de base et standard;
- ForkJoinPool, un ThreadPool pour soumettre le travail en petites tâches;
- ScheduledThreadPoolExecutor, il hérite de l'interface ThreadPoolExecutor et implémente les méthodes des tâches de minutage associées dans ScheduledExecutorService.

Vous pouvez trouver ci-dessous le diagramme de classe du framework Executor, il présente la relation parmi les interfaces, l'interface ExecutorService hérite de l'interface Executor et l'interface ScheduledExecutorService hérite de l'interface ExecutorService.



Thread Pools

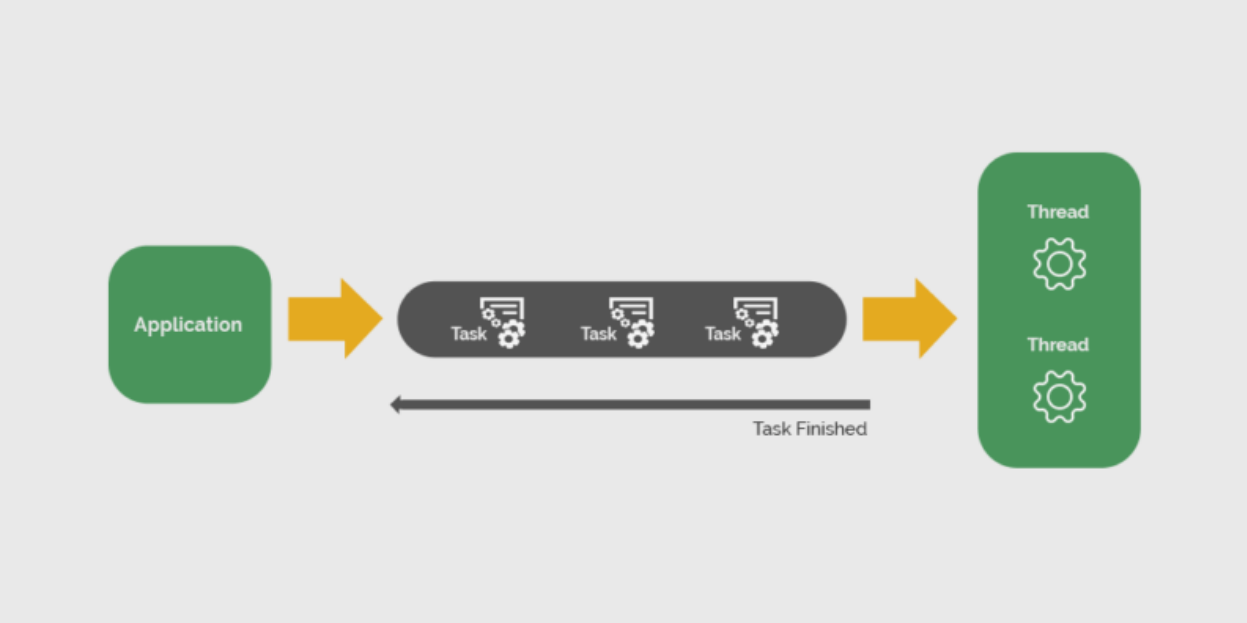
La création d'un ou deux threads est facile, mais cela devient un problème lorsque l'application nécessite la création de 20 ou 30 threads pour exécuter des tâches simultanément car elle nécessite que le système d'exploitation alloue les ressources nécessaires aux threads en même temps. Il faut utiliser Thread Pool pour utiliser efficacement les ressources et augmenter les performances.

La plupart d'implémentation d'Executor utilise Thread Pool pour l'exécution des tâches, un Thread Pool est une collection des threads qui existent des tâches "Runnable" ou "Callable" gérées par Executor.

Thread Pool conserve un certain nombre de threads inactifs prêt à exécuter des tâches selon les besoins. Une fois qu'un thread a terminé l'exécution d'une tâche, il ne meurt pas. Au lieu de cela, il reste inactif dans le pool en attendant d'être choisi pour exécuter de nouvelles tâches.

Il y en a plusieurs implémentations en standard de l'interface ExecutorService:

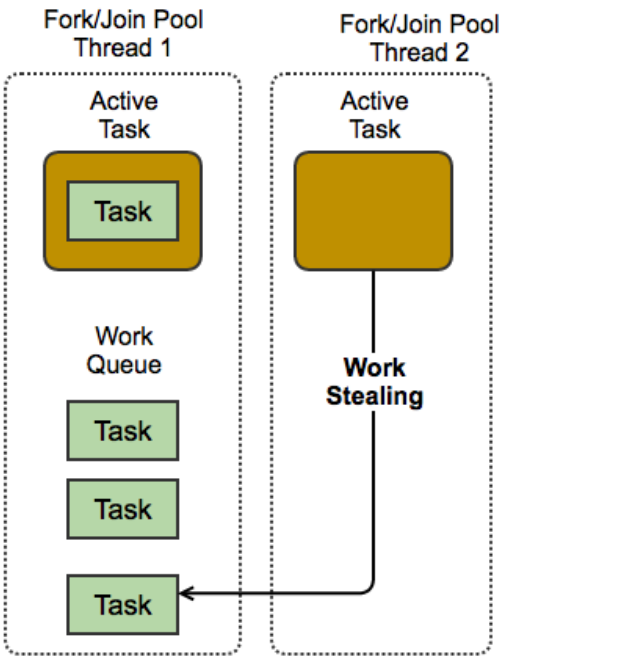
- Single Thread Executor ne contient qu'un seul thread, les tâches soumises sont exécutées de manière séquentielle;
- Cached Thread Pool contient plusieurs threads, le nombre de threads varie selon les besoins, les tâches sont exécutées en parallèle;
- Fixed Thread Pool contient un nombre fixe de threads, les tâches sont soumises à un Thread Pool via une file d'attente appelée "Blocking queue". S'il y a plus de tâches que le nombre de threads actifs, elles sont insérées dans la file de blocage pour attendre jusqu'à ce qu'un thread soit disponible. Si la file de blocage est pleine, les nouvelles tâches sont rejetées, les tâches sont exécutées en parallèle;
- Schedule Thread Pool contient plusieurs threads pour exécuter les tâches planifiées;
- Single Thread Scheduled Pool ne contient qu'un seul thread pour exécuter les tâches planifiées.



Fork / Join

Le framework fork / join est une implémentation de l'interface ExecutorService qui nous aide à tirer parti de plusieurs processeurs. Il est conçu pour le travail qui peut être divisé en petits morceaux de manière récursive. L'objectif est d'utiliser toute la puissance de traitement disponible pour améliorer les performances de votre application.

Supposons que nous devions incrémenter les valeurs d'un tableau de N nombres. Il faut d'abord diviser le tableau par deux en créant deux sous-tâches. Puis diviser à nouveau chacun d'eux en deux sous-tâches supplémentaires, et ainsi de suite jusqu'à nous obtenons les petites tâches qui peuvent être exécutées en parallèle par les multiples processeurs principaux disponibles.



2. Les implémentations d'Executors, Thread Pool et Fork / Join

1) Executors :

Executor Interface: L'interface *Executor* fournit une méthode unique, *execute*, conçue pour remplacer un idiome commun de création de threads. Si *r* est un objet *Runnable*, et *e* est une objet *Executor*, on peut remplacer < *(new Thread(r)).start();* > par < *e.execute(r);* >

ExecutorService Interface: L'interface *ExecutorService* complète *execute* avec une méthode *submit*. Comme *execute*, *submit* accepte les objets *Runnable*, mais accepte également les objets *Callable*, qui permettent à la tâche de renvoyer une valeur. La méthode *submit* renvoie un objet *Future*, qui est utilisé pour récupérer la valeur de retour *Callable* et pour gérer l'état des tâches *Callable* et *Runnable*.

Plusieurs implémentations sont fournies en standard :

- java.util.concurrent.ThreadPoolExecutor
- java.util.concurrent.ScheduledThreadPoolExecutor
- java.util.concurrent.ForkJoinPool (depuis Java 7)

ScheduledExecutorService Interface: L'interface *ScheduledExecutorService* complète les méthodes de son parent *ExecutorService* avec *schedule*, qui exécute une tâche *Runnable* ou *Callable* après un délai spécifié.

2) Thread Pool :

On peut utiliser *java.util.concurrent.ThreadPoolExecutor* ou *java.util.concurrent.ScheduledThreadPoolExecutor* pour instancier un pool de threads.

On peut également utiliser Executors pour créer une classe de fabrique pour les pools de threads. Les méthodes courantes sont les suivantes:

- *newFixedThreadPool (int nThreads)* crée un pool de threads avec une taille fixe et une file d'attente de tâches illimitée.

  Le nombre de threads principaux = le nombre maximum de threads.

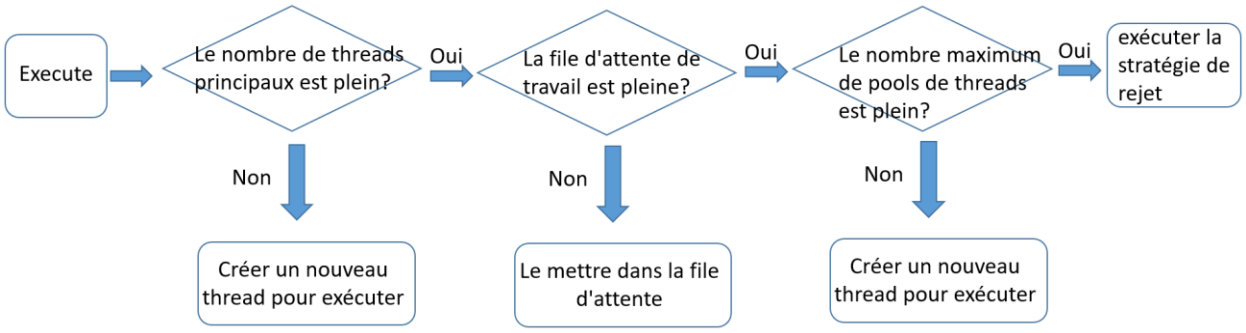
- *newCachedThreadPool()* crée un pool de threads tamponné de taille illimitée.

  Sa file d'attente de tâches est une file d'attente synchronisée. Les tâches sont ajoutées au pool. S'il y a des threads inactifs dans le pool, ils sont exécutés avec des threads inactifs, et s'il n'y en a pas, de nouveaux threads sont créés pour être exécutés. Les threads du pool sont inactifs pendant plus de 60 secondes et seront détruits et libérés. Le nombre de threads varie avec le nombre de tâches. Il convient à l'exécution de tâches asynchrones qui prennent moins de temps. Le nombre de threads principaux du pool = 0, le nombre maximum de threads = Integer.MAX_VALUE

- *newSingleThreadExecutor()* n'a qu'un seul thread pour exécuter le pool de threads unique de la file d'attente de tâches illimitée.

  Le pool de threads garantit que les tâches sont exécutées une par une dans l'ordre dans lequel elles ont été ajoutées. Lorsque le seul thread est abandonné en raison d'une tâche, un nouveau thread est créé pour continuer à exécuter les tâches suivantes. La différence avec newFixedThreadPool (1) est que la taille du pool d'un seul pool de threads est codée en dur dans la méthode newSingleThreadExecutor et ne peut pas être modifiée.

Processus d'exécution de la tâche:



1. Le nombre de threads principaux est-il plein? S'il n'est pas plein, créez un thread de travail pour effectuer la tâche.

2. La file d'attente de travail est-elle pleine? S'il n'est pas plein, la tâche nouvellement soumise est stockée dans la file d'attente de travail.
3. Le nombre maximum de pools de threads est-il plein? S'il n'est pas plein, un nouveau thread de travail est créé pour exécuter la tâche.
4. Enfin, implémentez une stratégie de rejet pour gérer cette tâche.

3) Fork / Join:

pseudo - code

```
if ( problem.size() > DEFAULT_SIZE) {
        divideTasks();
        executeTask();
        taskResults = joinTasksResult();
        return taskResults;
} else {
        taskResults = solveBasicProblem();
        return taskResults;
}
```

Enveloppez ce code dans une sous-classe ForkJoinTask, en utilisant généralement l'un de ses types plus spécialisés, soit RecursiveTask (qui peut renvoyer un résultat), soit RecursiveAction.

Une fois que votre sous-classe ForkJoinTask est prête, créez l'objet qui représente tout le travail à effectuer et passez-le à la méthode invoke () d'une instance ForkJoinPool.

4) Tests:

< github: https://github.com/GuoJulie/S9_Java-Performance/tree/main/TP4%20Mini-presentation%20sur%20le%20theme%20Java%20Performance/Executors_Test/src >

[1] Ex: Executors + Thread Pool

Fichier: *TestThreadPoolExecutor.java*

a) test1 :

- executor: Interface: *ExecutorService* --> *submit(Runnable)*
- return: *Future* object
- thread pool: *ThreadPoolExecutor()* --> corePoolSize:2; maximumPoolSize:4; keepAliveTime:60s
- thread real: 1

code :

```
// test1 + test2
ExecutorService executorService = new ThreadPoolExecutor(2, 4, 60,
    TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
```

Lorsque l'exécution d'une tâche est confiée à un thread du pool, si le nombre de threads présent dans le pool est inférieur au maximum alors un nouveau thread est créé et ajouté au pool même si un ou plusieurs threads sont inactifs dans le pool.

Si la queue interne est pleine et que le nombre de threads est supérieur à corePoolSize et inférieur à maximumPoolSize alors un nouveau thread est créé et ajouté dans le pool pour exécuter des tâches.

On configure corePoolSize=2 et maximumPoolSize=4.

```
// test1 + test3
Future future = executorService.submit(new Runnable() { // test1
executorService.execute(new Runnable() {    // test3
    @Override
    public void run() {
        System.out.println("debut tache " + Thread.currentThread().getName());
        try {
          Thread.sleep(5000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        System.out.println("fin tache");
    }
});
```

Juste 1 tâche est soumise pour exécution au pool. On utilise la méthode submit (Runnable) pour exécuter une tâche qui implémente l'interface Runnable et renvoyer un objet de type Future qui permet de déterminer si l'exécution de la tâche est terminée. Une tâche soumise à un ExecutorService sera exécutée par un thread inactif du pool.

```
// test1
try {
  System.out.println("resultat=" + future.get());
} catch (InterruptedException e) {
  e.printStackTrace();
} catch (ExecutionException e) {
  e.printStackTrace();
}
```

La méthode get() de l'instance de type Future obtenue en invoquant la méthode submit() avec un Runnable renvoie toujours null.

```
executorService.shutdown();
executorService.awaitTermination(300, TimeUnit.SECONDS); // permet d'attendre de man
```

On utilise la méthode shutdown() pour fermer ExecutorService lorsque ExecutorService n'a plus d'utilité ou lorsqu'il doit être arrêté. shutdown() : une fois invoquée, le service ne peut plus prendre de nouvelles

tâches, les tâches en cours d'exécution se poursuivent, les tâches non encore exécutées sont supprimées et enfin libèrent les ressources une fois toutes les tâches exécutées. Si la méthode shutdown() n'est pas invoquée alors la JVM continuera indéfiniment de s'exécuter même si les traitements du thread principal sont terminés.

La méthode awaitTermination() permet d'atteindre de manière bloquante la fin de l'exécution de toutes les tâches soumises.

résultat :

```
debut tache pool-1-thread-1
fin tache
resultat=null
Fin thread principal
```

b) test2:

- executor: Interface: *ExecutorService* --> *submit(Runnable)*
- return: *Future* object --> not set
- thread pool: *ThreadPoolExecutor()* --> corePoolSize:2; maximumPoolSize:4; keepAliveTime:60s
- thread real: 5

code :

```
// test1 + test2
ExecutorService executorService = new ThreadPoolExecutor(2, 4, 60,
    TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());

// test2
for (int i = 0; i < 5; i++) {
    executorService.submit(new Runnable() {
        @Override
        public void run() {
            System.out.println("debut tache " + Thread.currentThread().getName());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("fin tache");
        }
    });
}
```

On configure corePoolSize=2 et maximumPoolSize=4. Le nombre de threads du pool ne dépasse pas 2. La propriété maximumPoolSize vaut 4 et mais 5 tâches sont soumises pour exécution au pool. Ceci est lié au

fait que la collection de type LinkedBloquingQueue n'étant pas pleine (sa méthode offer() ne renvoie pas false), le ThreadPoolExecutor ne crée pas de nouveaux threads.

résultat :

```
debut tache pool-1-thread-2
debut tache pool-1-thread-1
fin tache
fin tache
debut tache pool-1-thread-2
debut tache pool-1-thread-1
fin tache
fin tache
debut tache pool-1-thread-2
fin tache
Fin thread principal
```

c) test3 :

- executor: Interface: *ExecutorService* --> execute(Runnable)
- return: none
- thread pool: *Executors.newSingleThreadExecutor()* --> PoolSize:1
- thread real: 1

code :

```
// test3
ExecutorService executorService = Executors.newSingleThreadExecutor();

// test1 + test3
Future future = executorService.submit(new Runnable() { // test1
executorService.execute(new Runnable() {    // test3
    @Override
    public void run() {
        System.out.println("debut tache " + Thread.currentThread().getName());
        try {
          Thread.sleep(5000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        System.out.println("fin tache");
    }
});
```

On utilise Executors.newSingleThreadPool() pour créer un seul thread et pour exécuter le pool de threads unique de la file d'attente de tâches illimitée.

On utilise la méthode execute (Runnable) pour exécuter et il n'a pas de valeur de retour. La méthode execute(Runnable) fait exécuter la tâche fournie en paramètre par le pool de threads.

résultat :

```
debut tache pool-1-thread-1
fin tache
Fin thread principal
```

d) test4 :

- executor: Interface: *ExecutorService* --> *submit(Callable)*
- return: *Future* object
- thread pool: *Executors.newFixedThreadPool()* --> PoolSize:3
- thread real: tache_1:10 + tache_2:10

code :

```
// test4
ExecutorService executorService = Executors.newFixedThreadPool(3);

// test4
Future<String> future1 = executorService.submit(new Callable<String>() {
    public String call() throws Exception {
        int i = 0;
        System.out.println("debut tache 1");
        while (i < 10 && !Thread.currentThread().isInterrupted()) {
            Thread.sleep(1000);
            i++;
        }
        System.out.println("fin tache 1");
        return "Tache 1";
    }
});
Future<String> future2 = executorService.submit(new Callable<String>() {
    public String call() throws Exception {
        int i = 0;
        System.out.println("debut tache 2 ");
        while (i < 10 && !Thread.currentThread().isInterrupted()) {
            Thread.sleep(500);
            i++;
        }
        System.out.println("fin tache 2");
        return "Tache 2";
    }
});
```

```java
// test4
try {
    executorService.awaitTermination(1, TimeUnit.HOURS);
    System.out.println("result1 = " + future1.get());
    System.out.println("result2 = " + future2.get());
} catch (InterruptedException ie) {
    ie.printStackTrace();
} catch (ExecutionException ee) {
    ee.printStackTrace();
}
```

On utilise Executors.newFixedThreadPool() pour créer un pool de threads avec une taille fixe (ici est 3) et une file d'attente de tâches illimitée.

La méthode submit(Callable) permet de soumettre l'exécution d'un tâche de type Callable et renvoie un objet de type Future qui permet d'obtenir des informations sur l'exécution.

résultat :

```
debut tache 1
debut tache 2
fin tache 2
fin tache 1
result1 = Tache 1
result2 = Tache 2
Fin thread principal
```

Il s'agit de deux pools de threads, qui exécutent la tâche 1 et la tâche 2, chacune avec 10. La différence est que l'intervalle entre les threads est différent. La tâche 2 est exécutée en premier.

[2] Ex: Fork / Join

Fichier: *TestThreadPoolExecutorForkJoin.java*

test5:

Test objectif: 1 à 100 pour la somme

- executor: Interface: *ExecutorService* --> invoke() / submit()
- return: *Future* object
- thread pool: *ForkJoinPool()* --> corePoolSize: available processors [choose 2] (*ForkJoinPool* --> public pool)
- thread real: realsize / threshold (--> fork)

code :

```java
public static final int threshold = 2;
```

```java
@Override
protected Integer compute() {
    int sum = 0;
    // determine whether the work is performed or split by the array length
    boolean canCompute = (end - start) <= threshold;
    if (canCompute) {
        for (int i = start; i <= end; i++) {
            sum += i;
        }
    } else {
        int split = (start + end) / 2;
        TestThreadPoolExecutorForkJoin demo1 = new TestThreadPoolExecutorForkJoin(start, split);
        TestThreadPoolExecutorForkJoin demo2 = new TestThreadPoolExecutorForkJoin(split + 1, end);

        demo1.fork();
        demo2.fork();

        int result1 = demo1.join();
        int result2 = demo2.join();
        sum = result1 + result2;

          List<Future<Integer>> futures = invokeAll(new TestThreadPoolExecutorForkJoin(start, split),
                    new TestThreadPoolExecutorForkJoin(split + 1, end));

          for (Future<Integer> future : futures) {
            sum = +future.get();
          }
    }

    return sum;
}
```

On implémente la méthode abstraite compute (), qui effectue directement le calcul de sommation ou le divise en deux tâches plus petites. Un simple seuil de longueur de tableau permet de déterminer si le travail est effectué ou fractionné.

```java
ForkJoinPool forkJoinPool = new ForkJoinPool();

TestThreadPoolExecutorForkJoin forkjoindemo = new TestThreadPoolExecutorForkJoin(1, 100);

long startTime = System.currentTimeMillis();
  forkJoinPool.invoke(forkjoindemo);
Future<Integer> future = forkJoinPool.submit(forkjoindemo);
long endTime = System.currentTimeMillis();

try {
    System.out.println("sum(1,100):" + future.get());
} catch (Exception e) {
    e.printStackTrace();
}

System.out.println("Treatment took " + (endTime - startTime) +
        " milliseconds.");
```

Enveloppez le code de la méthode compute () dans une sous-classe ForkJoinTask, en utilisant généralement l'un de ses types plus spécialisés, soit RecursiveTask (qui peut renvoyer un résultat), soit RecursiveAction.

Une fois que la sous-classe ForkJoinTask est prête, créez l'objet qui représente tout le travail à effectuer et transmettez-le à la méthode invoke() (ou submit() / execute()) d'une instance ForkJoinPool.

résultat :

```
Objectif: 1 à 100 pour la somme
Threshold is 2
4 processors are available
sum(1,100):5050
Treatment took 1 milliseconds.
```

3. Les impacts sur les performances

La programmation en parallèle peut effectivement augmenter les performances de l'application.

Avantages de ThreadPool:

- Gain de temps
- Réutilisation des threads existants
- Contrôler le nombre maximum de threads simultanés
- Augmenter le taux d'utilisation des ressources systèmes
- Éviter une concurrence excessive et un blocage des ressources
- Fournir des fonctions telles que l'exécution de la synchronisation, l'exécution régulière etc
- Améliorer la vitesse de réponse

Risques de ThreadPool:

- Fuite de threads, si un thread est supprimé du Thread Pool pour effectuer une tâche, mais il n'est pas renvoyé lorsque la tâche est terminée, une fuite de thread se produit;
- Deadlock, dans le Thread Pool en cours d'exécution, le thread attend la sortie du bloc, le thread en attente dans la file d'attente en raison de l'indisponibilité du thread pour l'exécution, il y a un cas de deadlock;
- Limitation des ressources: plus de threads que le nombre optimal requis peut entraîner des problèmes de famine et conduire à des ressources insuffisantes

ForkJoinPool possède une meilleur performance que les autres Thread Pools d'ExecutorService:

- ForkJoinPool est une implémentation spécifiée de ExecutorService, il réalise un work-stealing algorithme, les threads inactifs volent les tâches des threads occupés, il peut donner de meilleurs performances;
- Supposons que vous spécifiez la capacité initiale du Thread Pool,
    - ForkJoinPool ajuste dynamiquement la taille du pool pour tenter de maintenir suffisamment de threads actifs à un moment donné;
    - Les threads du ForkJoinPool sont les threads démons, donc son pool n'a pas à être explicitement arrêté comme le ExecutorService "executorService.shutdown".

4. Références :

https://www.jmdoudoux.fr/java/dej/partie4.htm

https://www.jmdoudoux.fr/java/dej/chap-executor.htm

http://geekrai.blogspot.com/2013/07/executor-framework-in-java.html

http://sdz.tdct.org/sdz/le-framework-executor.html

https://www.edureka.co/blog/thread-pool-in-java/

https://www.java-success.com/10-%E2%99%A6-executorservice-vs-forkjoin-future-vs-completablefuture-interview-qa/

https://www.jmdoudoux.fr/java/dej/chap-executor.htm

https://docs.oracle.com/javase/tutorial/essential/concurrency/exinter.html

https://www.youtube.com/watch?v=MttQSEmRUEA

https://jueee.github.io/ConcurrencyWithJava/src/main/java/com/app/jueee/concurrency/chapter07/C1Fork-Join%E6%A1%86%E6%9E%B6%E7%AE%80%E4%BB%8B.html