APIs Concurrentes

Rémi Forax

APIs Concurrentes

En Java, il existe déjà des APIs concurrente; à utiliser au lieu de ré-inventer la roue

- java.lang
  - ThreadLocal
  - ClassValue
- java.util.concurrent
  - Verrous plus évolué
  - Collections concurrentes
  - Calcul asynchrone

Le problème – trouver le bug !

```
public class IOUtil {
  private Scanner scanner;

  public void open(Path path) {
    try {
      scanner = new Scanner(path);
    } catch (IOException e) {
      throw new UncheckedIOException(e);
    }
  }

  public String nextLine() {
    if (!scanner.hasNextLine()) {
      return null;
    }
    return scanner.nextLine();
  }

  public void close() {
    scanner.close();
  }
}
```

```
public static void main(String[] args) {
    IOUtil io = new IOUtil();
    for (String arg : args) {
      Path path = Paths.get(arg);
      new Thread(() -> {
        io.open(path);
        int count = 0;
        for (; io.nextLine() != null; count++)
          ;
        io.close();
        System.out.println(count);
      }).start();
    }
  }
```

ThreadLocal<T>

Déclarer une variable dont la valeur est spécifique à une thread

comme une Map<Thread, T> mais sur la thread courante de façon implicite

API

- T get() renvoie la valeur pour la thread courante
- set(T t) change la valeur pour la thread courante

L'implantation stock la valeur dans java.lang.Thread donc pas besoin de verrou

Avec un ThreadLocal

```
public class IOUtil {
  private final ThreadLocal<Scanner> scanner = new ThreadLocal<>();

  public void open(Path path) {
    try {
      scanner.set(new Scanner(path));
    } catch (IOException e) {
      throw new UncheckedIOException(e);
    }
  }

  public String nextLine() {
    if (!scanner.get().hasNextLine()) {
      return null;
    }
    return scanner.get().nextLine();
  }

  public void close() {
    scanner.get().close();
    scanner.remove();
  }
}
```

```
public static void main(String[] args) {
    IOUtil io = new IOUtil();
    for (String arg : args) {
      Path path = Paths.get(arg);
      new Thread(() -> {
        io.open(path);
        int count = 0;
        for (; io.nextLine() != null; count++)
          ;
        io.close();
        System.out.println(count);
      }).start();
    }
  }
```

Non mutable, c’est mieux !

```
public class IOUtil implements Closable {
  private final Scanner scanner;

  public IOUtil(Path path) {
    try {
      scanner = new Scanner(path);
    } catch (IOException e) {
      throw new UncheckedIOException(e);
    }
  }

  public String nextLine() {
    if (!scanner.hasNextLine()) {
      return null;
    }
    return scanner.nextLine();
  }

  public void close() {
    scanner.close();
  }
}
```

```
public static void main(String[] args) {
    for (String arg : args) {
      Path path = Paths.get(arg);
      new Thread(() -> {
        try(IOUtil io = new IOUtil(path)) {
          int count = 0;
          for (; io.nextLine() != null; count++)
            ;
          System.out.println(count);
        }
      }).start();
    }
  }
```

ClassValue<T>

Met en cache une valeur spécifique à une classe

comme une Map<Class, T> + putIfAbsent

On ne peut pas utiliser un objet Class comme clé d'une Map car cela empêche le déchargement de la classe.

API

- T get(Class<?> type) renvoie la valeur pour type, appel computeValue (qu'il faut redéfinir) si la valeur n'a pas été calculée

Si la classe est déchargée, la valeur associée disparait

Thread et calcul asynchrone

Problème de l’API java.lang.Thread

- thread.start() est lent
- Pour obtenir une valeur calculée par un Runnable, il faut utiliser thread.join()

Solutions:

- On pool les threads: Executor

  (création à la demande, thread non-détruite après un calcul)
- Un calcul est un Callable<T>

  T call() throws Exception
- Le résultat d’un calcul asynchrone est un Future<T>

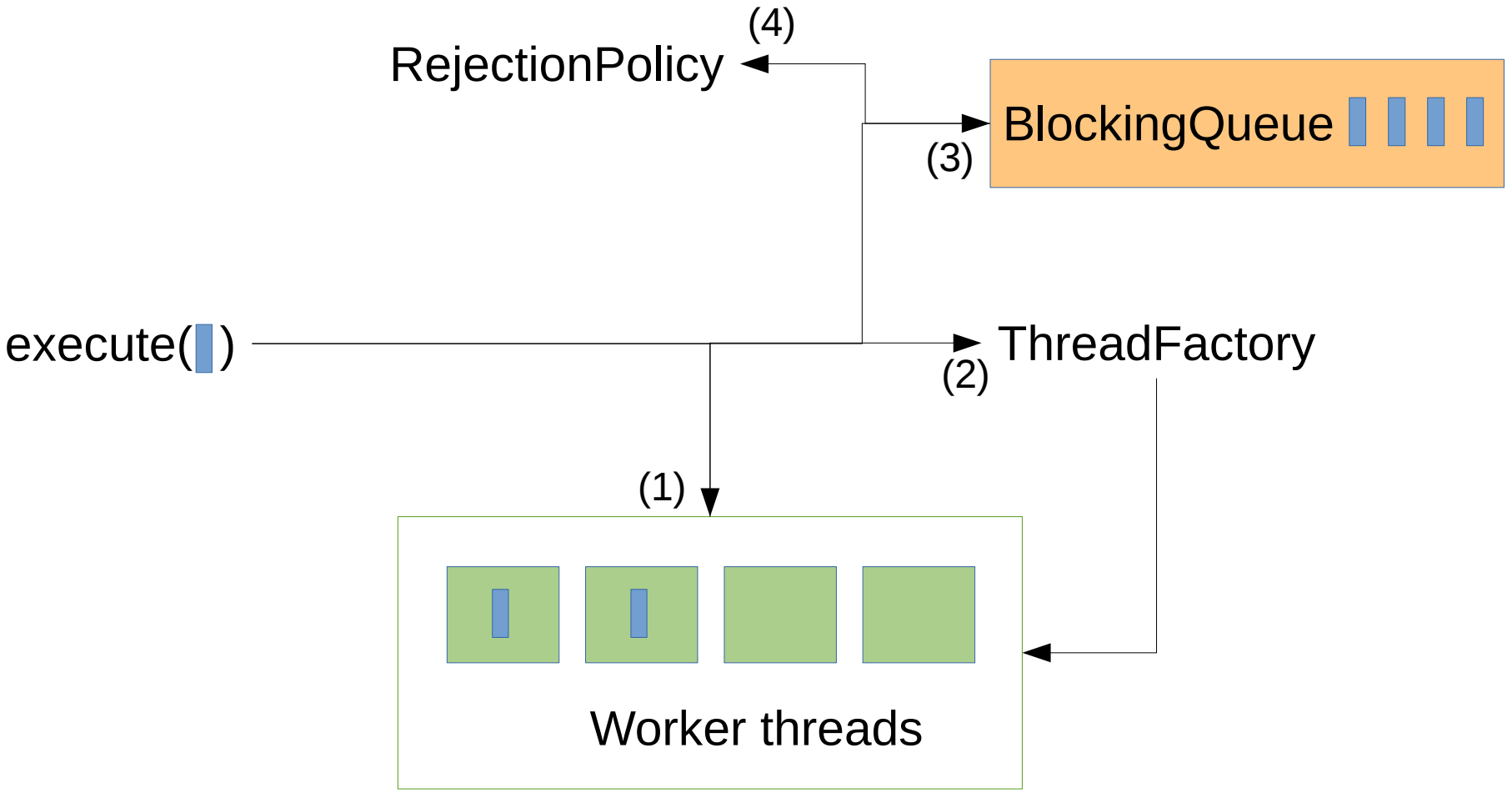
Exemple simple

```
BlockingQueue<Runnable> queue =
    new ArrayBlockingQueue<>(100);
Executor executor =
    new ThreadPoolExecutor(/* corePoolSize */          2,
                           /* maximumPoolSize */ 4,
                           /* keep alive */               1,
                                                         TimeUnit.MINUTES,
                           /* work queue */              queue);

executor.execute(() -> {
    System.out.println("hello 1");
});
executor.execute(() -> {
    System.out.println("hello 2");
});

executor.shutdown();
```

ThreadPoolExecutor



1. on essaye de donner à une thread libre. 2. on essaye de créer une thread.
3. on stocke dans la queue. 4. on rejete la tâche/met en attente/lève une exception

shutdown()

Un ThreadPoolExecutor crée des worker threads qui par défaut sont pas marqués deamon

La VM meurt que si il n’existe plus que des threads deamon

Si on ne fait rien, le ThreadPoolExecutor maintient la VM en vie, donc le programme ne s’arrête jamais

executor.shutdown()

- Il n’est plus possible de soumettre de nouvelle tâche
- Une fois toutes les tâches terminées, les worker threads sont arrêtés

puis la VM s’arrête

Exemple de calcul

```
BlockingQueue<Runnable> queue =
    new ArrayBlockingQueue<>(100);
ExecutorService executor =
    new ThreadPoolExecutor(/* corePoolSize */          2,
                           /* maximumPoolSize */ 4,
                           /* keep alive */               1,
                                                         TimeUnit.MINUTES,
                           /* work queue */              queue);

Callable<String> task = () -> "hello executor";
Future<String> future = executor.submit(task);
executor.shutdown();
try {
      System.out.println(future.get());
} catch(ExecutionException e) {
   // il y a eu une erreur pendant l'exécution du calcul
   // l'erreur est stockée dans le cause de l'exception
   throw new RuntimeException(e.getCause());
}
```

Future<T>

Promesse d’une valeur future et façon de contrôler l’exécution la tâche correspondante

API

- Demande la valeur, bloque si le calcul est pas fini
  - T **get**()
- Demande l’arrêt de la tâche
  - boolean **cancel**(boolean callInterruptIfRunning)
    (revoie false si la tâche est déjà fini)

ExecutorService

Interface qui hérite de Executor

- Soumettre un calcul
  - Future<T> **submit**(Callable<T>)
- Soumission de tâche interdite, l'Executor s'arrètera une fois tous les tâches en cours effectuée
  - void **shutdown**()
- Soumettre plein de tâches en même temps
  - List<Future<T>> **invokeAll**(Collection<? extends Callable<T>>)
- Soumettre plusieurs calculs, garder le premier qui répond, on arrête les autres
  - T **invokeAny**(Collection<? extends Callable<T>>

Créer un ExecutorService

new ThreadPoolExecutor(blah, blah, ...)

mais il y a plein de paramètres

Utilser les *static factory methods*

- Thread pool avec 1 thread unique
  - Executors.newSingleThreadExecutor()
- Thread pool avec n-threads
  - Executors.newFixedThreadPool(int nThreads)
- Thread pool qui alloue autant de thread que nécessaire avec un keepAlive infini
  - Executors.newCachedThreadPool()

Exemple de Ping

```
public static void main(String[] args)
                throws InterruptedException, UnknownHostException {
  InetAddress host = InetAddress.getByName("www.google.fr");

  ExecutorService executor = Executors.newCachedThreadPool();
  Future<Long> future = executor.submit(() -> {
    long time = System.nanoTime();
    host.isReachable(2000);
    return System.nanoTime() - time;
  });

  executor.shutdown();

  try {
    System.out.println("reach " + host + " in " + future.get() + " ns");
  } catch (ExecutionException e) {
    throw new RuntimeException(e.getCause());
  }
}
```

Et avec plusieurs hosts

```
private static Callable<Long> newCallable(InetAddress host) {
    return () -> {
      long time = System.nanoTime();
      host.isReachable(2000);
      return System.nanoTime() - time;
    };
  }

  public static void main(String[] args)
                                        throws InterruptedException, ExecutionException {
    List<InetAddress> hosts = …
    List<Callable<Long>> callables =
        hosts.stream().map(host -> newCallable(host)).collect(toList());

    ExecutorService executor = Executors.newFixedThreadPool(2);
    List<Future<Long>> futures = executor.invokeAll(callables);
    executor.shutdown();

    for (int i = 0; i < hosts.size(); i++) {
     System.out.println("reach " + hosts.get(i) + " in " + futures.get(i).get() + " ns");
    }
  }
```

CompletableFuture<T>

Sorte de Future sous steroid (ils héritent de Future),
ils permettent de chaîner les calculs comme avec un Stream

Création

- CompletableFuture.**runAsync**(Runnable, Executor)
- CompletableFuture.**supplyAsync**(Supplier, Executor)

Tranformation (équivalent du map)

- **thenApply**(Function<T, R>)
- **thenApplyAsync**(Function<T, R>, Executor)

Accept (opération terminal)

- **thenAccept**(Consumer<T>)
- **thenAcceptAsync**(Consumer<T>, Executor)

Exemple de CompletableFuture

```
private static Supplier<Long> newSupplier(InetAddress host) {
    return () -> {
      long time = System.nanoTime();
      host.isReachable(2000);    // FIXME catch IOException
      return System.nanoTime() - time;
    };
  }

  public static void main(String[] args)
                      throws InterruptedException, UnknownHostException, ExecutionException {
    InetAddress host = InetAddress.getByName("www.google.fr");
    ExecutorService executor = Executors.newCachedThreadPool();
    Supplier<Long> supplier = newSupplier(host);

    CompletableFuture<Long> future = CompletableFuture.supplyAsync(supplier, executor);
    executor.shutdown();

    System.out.println("reach " + host + " in " + future.get() + " ns");
    future.thenAccept(elapsedTime -> {
      System.out.println("reach " + host + " in " + elapsedTime + " ns");
    });

  }
```

Verrous améliorés

Verrous amélioré

Exchanger

- Echange 2 valeurs entre 2 threads

CountDownLatch/CyclicBarrier

- Point de RDV entre plusieurs threads

Semaphore

- Verrou avec plusieurs permis

ReentrantReadWriteLock

- 2 locks, un pour la lecture, un pour l’écriture

StampedLock

- Verrou optimiste

Collections concurrentes

Liste concurrente

```
List<Integer> list = ...

ExecutorService executor = Executors.newFixedThreadPool(5);
for(int n = 0; n < 5; n++) {
    executor.execute(() -> {
        for(int i = 0; i < 1_000; i++) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                return;
            }
            list.add(i);
        }
    });
}

executor.shutdown();
executor.awaitTermination(1, TimeUnit.DAYS);

System.out.println(list.size());
```

j.u.ArrayList	
Collections.synchronizedList(new ArrayList())	
j.u.Vector	
j.u.c.CopyOnWriteArrayList	

Liste concurrente

```
List<Integer> list = ...

ExecutorService executor = Executors.newFixedThreadPool(5);
for(int n = 0; n < 5; n++) {
    executor.execute(() -> {
        for(int i = 0; i < 1_000; i++) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                return;
            }
            list.add(i);
        }
    });
}

executor.shutdown();
executor.awaitTermination(1, TimeUnit.DAYS);

System.out.println(list.size());
```

j.u.ArrayList	**??**
Collections.synchronizedList(new ArrayList())	**5000**
j.u.Vector	**5000**
j.u.c.CopyOnWriteArrayList	**5000**

Liste concurrente & iteration

```
List<Integer> list = ...

new Thread(() -> {
    for(int i = 0; i < 1_000; i++) {
        Thread.sleep(1);   // FIXME add catch InterruptedException
        list.add(i);
    }
}).start();

new Thread(() -> {
    for(;;) {
        int sum = 0;
        for(int value: list) {
            sum += value;
        }
        System.out.println(sum);
        ...
    }
}).start();
```

j.u.ArrayList	
Collections.synchronizedList(new ArrayList())	
j.u.Vector	
j.u.c.CopyOnWriteArrayList	

Liste concurrente & iteration

```
List<Integer> list = ...

new Thread(() -> {
    for(int i = 0; i < 1_000; i++) {
        Thread.sleep(1);   // FIXME add catch InterruptedException
        list.add(i);
    }
}).start();

new Thread(() -> {
    for(;;) {
        int sum = 0;
        for(int value: list) {
            sum += value;
        }
        System.out.println(sum);
        ...
    }
}).start();
```

j.u.ArrayList	**CME !**
Collections.synchronizedList(new ArrayList())	**CME !**
j.u.Vector	**CME !**
j.u.c.CopyOnWriteArrayList	**OK**

Structures Concurrentes

- j.u.HashMap, TreeMap, HashSet, ArrayList, ArrayDeque
  - Pas concurrente
- j.u.Hashtable, Vector, Stack
  - Concurrente sauf itération (et deprecated !)
- Collections.synchronizedSet/List/Map()
  - Concurrente sauf itération
- j.u.c.ConcurrentHashMap, ConcurrentSkipListMap, CopyOnWriteArraySet, CopyOnWriteArrayList, ConcurrentLinkedDeque
  - Concurrente

java.util.concurrent

Toutes les collections de j.u.c sont concurrentes

- Attention, size() est pas forcément en temps constant (isEmpty() toujours)

Deux sémantiques d'itération

- Snapshot: (CopyOnWrite...)
  - l'iterateur voit les objets présent à la création de l'itérateur
- Weekly consistent (toutes les autres)
  - l'iterateur peut voir les objets ajoutés après la création de l'itérateur (ou pas).