Stream et Optional

Rémi Forax

java.util.stream.Stream

Abstraction d’un flux d’élement sur lequel on veut faire des calculs

- Ce n’est pas une Collection car un Stream ne contient pas d’élement
- Ce n’est pas un Iterator car un Stream correspond à un calcul complet et pas à une étape du calcul
  - Un Stream à une vision plus globale du traitement

Sources d’un Stream

Comme un Stream ne stocke pas les données, elles proviennent d’une source

- A partir de valeurs
  - Stream.empty(), Stream.of(E… element), Stream.ofNullable(E element)
- A partir d’une collection
  - collection.stream()
- A partir d’un fichier
  - Files.lines(Path path)

Sources d’un Stream (2)

Et aussi

- A partir d’un interval
  - IntStream.range(int start, int end)
- A partir d’un tableau
  - Arrays.stream(E[] array)
- A d’une liste chainée
  - Stream.iterate(head, e -> e != null, e -> e.next)

L’API des Streams

Basée sur les opérations sans état (*stateless*)

- filter, map/flatMap, reduce

Capable d’arrêter le calcul si résultat est trouvé (*short circuit*)

- limit(), findFirst()/findAny(), takeWhile()

Les opérations intermédiaire modifie le Stream, les opérations terminales lancent le calcul

Opérations intermédiaires (1/2)

Sélectionne si un élement reste dans le Stream

```
Stream<E> filter(Predicate<? super E>)
```

Transforme un élément

```
<R> Stream<R> map(
                    Function<? super E, ? extends R)
```

Tranforme un élement en une série d'élements

```
<R> Stream<R> flatMap(
        Function<? super E, ? extends Stream<R>>)
```

Saute des éléments, Sélectionne les premiers éléments

```
Stream<E> skip(int length)
Stream<E> limit(int maxSize)
```

Opérations intermédiaires (2/2)

Supprime les doublons

Stream<E> distinct()

Trie les élements

Stream<E> sorted(Comparator<? super E>)

Obtenir les éléments au milieu du Stream (pour débugger)

Stream<E> peek(Consumer<? super E>)

Sélectionner/supprimer des élements (stop après)

Stream<E> takeWhile(Predicate<? super E>)
Stream<E> dropWhile(Predicate<? super E>)

Opérations Terminales (1/2)

Compte les éléments

long count()

Appel le consumer pour chaque élement

Stream<E> forEach(Consumer<? super E>)
Stream<E> forEachOrdered(Consumer<? super E>)

Vrai si tout les/au moins un élement(s) qui match

allMatch(Predicate<? super E>)
anyMatch(Predicate<? super E>)

Opérations Terminales (2/2)

Trouver le/un premier élement

Stream<E> findFirst()
Stream<E> findAny()

Créer un tableau

E[] toArray(IntFunction<E[]>)

aggège les données (sans mutation)

reduce(T seed, BinaryOperator<T> reducer)

aggège les données (mutable)

T collect(Collector<E,A,T>)

Ré-utilisation d’un Stream

Un Stream permet de faire un calcul, pas plusieurs

```
Stream<String> stream = …
Stream<String> s1 = stream
        .map(String::toLowerCase);
IntStream s2 = stream
        .mapToInt(String::length);  // IllegalStateException
```

Il n’est donc pas possible de réutiliser un Stream, il faut créer plusieurs streams sur la même source

Effet de bord sur la source

Un Stream ne doit pas faire d'effet de bord sur la source de ses élements

- List<String> list = …
  list.stream()
  .forEach(list::add); // aaaaaaah

Pour les collections non concurrentes,
elles leveront ConcurrentModificationException

Pour les autres sources, la sémantique est pas définie, la boucle infinie est possible !

Où est le bug ?

Le code suivant ne compile pas :(

```
public static int sum(int[][] table) {
    return Arrays.stream(table)
                 .flatMap(Arrays::stream)
                 .sum();
 }
```

Le résultat doit être un IntStream

Il faut utiliser flatMapToInt, pas flatMap !

```
public static int sum(int[][] table) {
    return Arrays.stream(table)
                 .flatMapToInt(Arrays::stream)
                 .sum();
 }
```

Des Streams de type primitif

IntStream, LongStream et DoubleStream

- Évite le boxing
- Possède des méthodes spécifiques (sum, average, etc.)

Sur un Stream, il existe plusieurs version de map(), mapToInt, mapToDouble, mapToLong qui renvoie des Stream de type primitif (même chose pour flatMap)

Un Stream peut être infinie

Afficher les valeurs inférieur à 100 de la suite

- $U_0 = 1$
- $U_n = 2 * U_{n-1} + 1$

```
public static void main(String[] args) {
   IntStream
      .iterate(1, n -> 2 * n + 1)         // stream infinie !
      .takeWhile(v -> v < 100)
      .forEach(System.out::println);
}
```

Collector

Aggrège les données de façon mutable

- Supplier qui crée un container () → A
- Un accumulateur mutable (A, T) → void
- Un combiner fonctionnel (A, A) → A

Exemples:

- Le Collectors.toList() est équivalent à
  - Collector.of(ArrayList::new,
    ArrayList::add,
    (c1, c2) → { c1.addAll(c2); return c1; }

Exemple de Collector

```
public class SmallSet<E> {
  private int mask;
  private final ArrayList<E> list = new ArrayList<>();

  public void add(E element) {
    int hash = element.hashCode();
    if ((hash | mask) != mask || !list.contains(element)) {
      mask |= hash;
      list.add(element);
    }
  }

  public SmallSet<E> appendAll(SmallSet<? extends E> set) {
    set.list.forEach(this::add);
    return this;
  }

  public static <T> Collector<T, SmallSet<T>, SmallSet<T>> toSmallSet() {
    return Collector.of(SmallSet::new,
                        SmallSet::add,
                        SmallSet::appendAll);
  }
```

collect

Equivalent à faire un reduce() mais pour les objets mutable

Permet de renvoyer un container mutable contenant tous les élements du Stream

java.util.stream.Collectors définie des Collector prédéfinie

Collectors

- toList(), toSet()
- toCollection(Supplier(? Extends Collection<E>>)) → Collection<E>
- toMap(Function<T, K> keyMapper, Function<T, V valueMapper) → Map<K,V>
- groupingBy(Function<...> mapper)
- joining(separator, first, last)

Operations “statefull”

Certaines opérations demande à garder un état pour effectuer le calcul

- distinct()
  - On doit garder les éléments déjà vu dans un Set
- sorted()
  - On doit garder les éléments dans une List avant de pouvoir les trier

Ces opérations vont allouées de la mémoire linéairement par rapport au nombre d’élements

Stream parallèle

L’API des Streams est la même que le calcul soit fait en série ou en parallèle

On obtient un stream parallèle

- Soit en appelant collection.parallelStream()
- Soit rendant le Stream parallèle Stream.parallel()

Attention, un Stream parallèle ne va pas forcément plus vite qu’un Stream séquentiel

- Le temps de dstribution du calcul et de ré-agréggation peut être plus lent que le temps du calcul lui-même

Effet de bord

Faire des effets de bord avec un Stream parallèle fait n'importe quoi

- Avec effet de bord

```
List<String> list = …
ArrayList<String> result = new ArrayList<>();
list.parallelStream()
    .map(String::toLowerCase)
    .forEach(result::add);    // aaaaaaaaaaaaaaah
```

- Sans effet de bord

```
List<String> list = …
list.parallelStream()
    .map(String::toLowerCase)
    .collect(Collectors.toList());
```

Relacher la contrainte de l’ordre

Paralléliser un calcul dont le résultat dépend d’un ordre est en rêgle général pas efficace

- distinct()
  - On doit partagée le Set des éléments déjà vus:(
- limit()
  - On peut utiliser unordered() pour indiquer que l’ordre est pas important
- findFirst()
  - On utilise findAny() à la place
- forEachOrdered()
  - On utilise forEach() à la place

Spliterator

java.util.Spliterator

Abstraction caché derrière un Stream

- Cela évite de ré-implanter les 30+ méthodes de l'interface Stream pour chaque collection

Un Spliterator est un iterateur *push* (java.util.Iterator est *pull*) qui sait se séparer en 2 (split)

Un Spliterator retient les caractéristiques de la source du Stream pour optimiser les traitements

- Il essaye aussi d'estimer sa taille pour séparer en 2 parties à peu près égale

Implanter un Spliterator<T>

trySplit(): Spliterator<T>

- Essaye de se couper en 2, renvoie l'autre partie ou null

tryAdvance(Consumer<? super T> action)

- Si il reste au moins un élement, appel le consumer sur l'élement courant, passe au suivant et renvoie vrai sinon renvoie faux

int characteristics()

- Renvoie les caractéristiques du spliterator

long estimateSize()

- La taille estimée ou Long.MAX_VALUE
  (si infinie ou trop long à calculer)

Exemple à partir d'un Iterateur

```
public static <T> Spliterator<T> fromIterator(Iterator<? extends T> it) {
    return new Spliterator<T>() {
      @Override
      public Spliterator<T> trySplit() { return null; }

      @Override
      public boolean tryAdvance(Consumer<? super T> consumer) {
        if (it.hasNext()) {
          consumer.accept(it.next());
          return true;
        }
        return false;
      }

      @Override
      public int characteristics() { return 0; }

      @Override
      public long estimateSize() { return Long.MAX_VALUE; }
    };
  }
```

Exemple à partir d'un Iterateur

java.util.stream.StreamSupport permet de créer un Stream à partir d'un Spliterator

```
Iterator<Integer> it = …
boolean parallel = ...
Stream<Integer> stream =
      StreamSupport.stream(fromIterator(it),
                           parallel);
```

Caracteristiques d’un Spliterator

CONCURRENT (La source est concurrente)

IMMUTABLE (La source est non mutable)

NONNULL (Pas d’élement null)

ORDERED (Les éléments sont ordonnées)

DISTINCT (Pas deux fois le même élement)

SORTED (Les données sont triées)

SIZED (Le nombre d’éléments est connu)

SUBSIZED (Après un trySplit(), les Spliterators connaissent leur taille)

Exemple à partir d'un Tableau

```
@SafeVarargs
public static <T> Spliterator<T> fromArray(int start, int end, T... array) {
  return new Spliterator<T>() {
     private int i = start;

    @Override
    public Spliterator<T> trySplit() { return null; }

    @Override
    public boolean tryAdvance(Consumer<? super T> consumer) {
      if (i < end) { consumer.accept(array[i++]);  return true; }
      return false;
    }

    @Override
    public int characteristics() { return SIZED; }

    @Override
    public long estimateSize() { return end - i; }
  };
}
```

Avec le trySplit

```
@SafeVarargs
public static <T> Spliterator<T> fromArray(int start, int end, T... array) {
  return new Spliterator<T>() {
    private int i = start;

    @Override
    int middle = (end - i) >>> 1;
      if (middle == i) {
        return null;
      }
      Spliterator<T> spliterator = fromArray(i, middle, array);
      i = middle;
      return spliterator;
    }

    @Override
    public int characteristics() { return SIZED | SUBSIZED; }

    ...
  };
}
```

Methodes additionnelles

Methodes dont ils existent une implantation par défaut et qui peuvent être redéfinie si il existe une implantation plus efficace

- forEachRemaning(Consumer<? super T> consumer)
  - Parcours les éléments restants comme un forEach
- Comparator<? super T> getComparator()
  - Renvoie le comparateur utiliser par la collection (pour TreeSet ou TreeMap.keySet())

java.util.Spliterators

Fournit des implantations de Spliterator par défaut

- Spliterator<T> spliterator(T[] array)
- Spliterator<T> spliteratorUnknownSize(
  Iterator<? extends T> iterator)

et une méthode inverse

- Iterator<T> iterator(
  Spliterator<? extends T> spliterator)

Exemple complet: FizzBuzz

Comment écrire FizzBuzz avec des Streams ?

FizzBuzz:
pour les nombres de 1 à 100

- On affiche Fizz, si le nombre est un multiple de 3
- On affiche Buzz, si le nombre est un multiple de 5
- On affiche FizzBuzz, si c'est un multiple de 15
- Sinon on affiche le nombre

Note: il existe des solutions très simples
pour implanter FizzBuzz sans Stream

FizzBuzz

On utilise plusieurs streams infinis

ints	cycle("Fizz")	cycle("Buzz")	zip(+)
0	Fizz	Buzz	FizzBuzz
1	""	""	""
2	""	""	""
3	Fizz	""	Fizz
4	""	""	""
5	""	Buzz	Buzz
6	Fizz	""	Fizz
7	""	""	""
8	""	""	""
9	Fizz	""	Fizz
…	…	…	…

FizzBuzz: cycle

```
public static <T> Stream<T> cycle(
                    int length, T element, T empty) {

    return IntStream
        .iterate(0, x -> (x + 1) % length)
        .mapToObj(x -> x == 0? element: empty);
  }
```

FizzBuzz: zip()

```
public static <T, U, V> Stream<V> zip(Stream<T> s1, Stream<U> s2,
                                      BiFunction<? super T, ? super U, ? extends V> merger) {
    Spliterator<T> spliterator1 = s1.spliterator();
    Spliterator<U> spliterator2 = s2.spliterator();
    return StreamSupport.stream(new Spliterator<V>() {
      @Override
      public int characteristics() {
        return spliterator1.characteristics() & spliterator2.characteristics();
      }

      @Override
      public long estimateSize() {
        long size1 = spliterator1.estimateSize();
        long size2 = spliterator2.estimateSize();
        return (size1 == Long.MAX_VALUE || size2 == Long.MAX_VALUE)?
                Long.MAX_VALUE: Math.min(size1, size2);
      }

      @Override
      public Spliterator<V> trySplit() { return null; }

      ...
    }, false);
  }
```

FizzBuzz: zip()

```
public static <T, U, V> Stream<V> zip(Stream<T> s1, Stream<U> s2,
                                      BiFunction<? super T, ? super U, ? extends V> merger) {
    Spliterator<T> spliterator1 = s1.spliterator();
    Spliterator<U> spliterator2 = s2.spliterator();
    return StreamSupport.stream(new Spliterator<V>() {
      ...
      @Override
      public boolean tryAdvance(Consumer<? super V> action) {
        class Box { T t; }
        Box box = new Box();
        return spliterator1.tryAdvance(t -> box.t = t) &&
          spliterator2.tryAdvance(u -> action.accept(merger.apply(box.t, u)));
      }
    }, false);
  }
```

FizzBuzz: main

```
Stream<String> fizzbuzz =
    zip(
      IntStream.iterate(0, x -> x + 1).boxed(),
      zip(
        cycle(3, "Fizz", ""),
        cycle(5, "Buzz", ""),
        String::concat),
        (i, s) -> s.isEmpty()? i.toString(): s);


fizzbuzz.skip(1)
            .limit(100)
            .forEach(System.out::println);
```

Optional

java.util.Optional

Si le résultat d'un calcul peut ne pas existé, au lieu d'utiliser null, on utilise Optional qui oblige l'utilisateur à gérer le cas où il n'y a pas de valeur

Exemples:

- Trouver la première valeur d'un Stream<E>
  - Stream.findFirst() renvoie Optional<E>
- Calculer le maximum d'un IntStream
  - intStream.max() renvoie un OptionalInt

Comme les Streams, il existe des Optional pour les types primitif (OptionalInt, OptionalLong, OptionalDouble)

Optional et calcul

Optional est un monad comme Stream, en plus de représenter une valeur ou non d'un calcul, il est possible d'effectuer des opérations directement sur un Optional

Au lieu de

```
Optional<String> opt = …
if (opt.isPresent()) {
  System.out.println(opt.get());
}
```

on demande à l'Optional de faire le calcul

```
Optional<String> opt = …
opt.ifPresent(v -> System.out.println(v));
```

API

Effet de bord

void **ifPresent(**Consumer<? super T>)

Test

Optional<E> **filter**(Predicate<? super T>)

Transformation

Optional<R> **map**(Function<? super T, ? extends R>),

Optional<R> **flatMap**(Function<? super T, ? extends Optional<? extends R>)

Composition

Optional<E> **or**(Supplier<? extends Optional<? extends T>>)

Obtenir la valeur

get() throws NSEE, **orElse**(T), **orElseGet**(Supplier<? extends T>), **orElseThrow**(Supplier<? extends Throwable>)

Pont vers un Stream

Stream<T> **stream**()

Créer un Optional

Optional possède des méthodes statiques de création (**static factory methods**)

- Créer un Optional sans valeur

  Optional.empty()

- Créer un Optional avec une valeur non null

  Optional.of(E element)

- Créer un Optional qui peut être null

  Optional.ofNullable(E elementOrNull)

Exemple

Au lieu de

```
OptionalInt result = ...
if (result.isPresent()) {
  System.out.println("result is " + result.get());
} else {
  System.out.println("result not found");
}
```

On écrit

```
OptionalInt result = …
System.out.println(result
        .map(value - > "result is " + value)
        .orElse("result not found"));
```

Utilisation d’Optional

Attention, il ne faut jamais !!

Stocker un Optional dans un champ !

- Double dé-référencement inutile

Avoir une Collection ou Map de Optional

- Autant ne pas mettre les trucs qui existe pas dans la collection

Stocker Optional dans un champ

Au lieu de

```
public class Foo {
  private final Optional<Bar> bar;

  public Foo(Optional<Bar> bar) { this.bar = bar; }

  public Optional<Bar> getBar() { return bar; }
}
```

on écrit

```
public class Foo {
  private final Bar bar;  // maybe null

  public Foo(Optional<Bar> bar) { this.bar = bar.orElse(null); }

  public Optional<Bar> getBar() {
    return Optional.ofNullable(bar);
  }
}
```

Collection ou Map d'Optional

On va éviter de stocker des trucs qui peuvent ne pas exister

Au lieu de

```
List<Foo> foos = …
List<Optional<Bar>> bars =
    foos.stream()
        .map(foo - > foo.getBar())
        .collect(Collectors.toList())
```

on écrit

```
List<Foo> foos = …
List<Bar> bars =
    foos.stream()
        .flatMap(foo - > foo.getBar().stream())
        .collect(Collectors.toList())
```