Les Gatherers

Rémi Forax

Les Gatherers ?

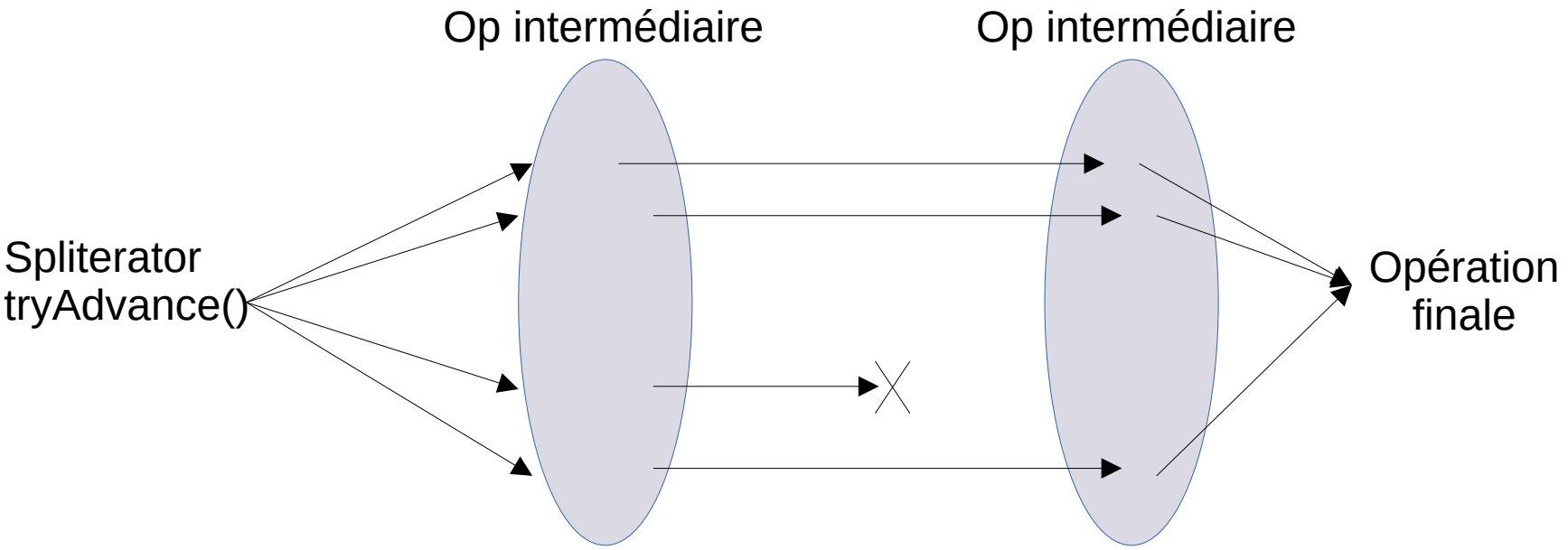
Un Stream possède

- des méthodes static de création (+ collection)
  - utilisent en interne un spliterator
- des méthodes intermédiaires
  - peuvent être généralisées par un ???
- des méthodes terminales
  - peuvent être généralisées par un Collector

Pipeline

Un Stream est organisé comme un pipeline

Les données sont poussées (*push*) dans le pipeline



Opérations intermédiaires

- Opérations sans état
  - filter(), map(), flatMap()/mapMulti(), takeWhile()
- Opérations short-circuit
  - takeWhile(), limit()
- Opérations avec état
  - distinct(), sorted(), limit(), skip()
- Opérations ordonnées (obligatoirement séquentiel)
  - limit(), skip()

Les Gatherers

Un Stream possède

- des méthodes static de création (+ collection)
  - utilisent en interne un spliterator
- des méthodes intermédiaires
  - peuvent être généralisées par un **Gatherer**
- des méthodes terminales
  - peuvent être généralisées par un Collector

Gatherer<E, A, T>

Création

- Gatherer.ofSequential(initializer?, integrator, finisher?)
- Gatherer.of(initializer?, integrator, combiner?, finisher?)

Fonctions

- initializer: creation d’un état (optionnel)
- integrator: pousse dans l’état suivant
- combiner: merge les états (optionnel pour parallel)
- finisher: pousse à la fin (optionnel)

Integrator<E, A, T>

La fonction integrator

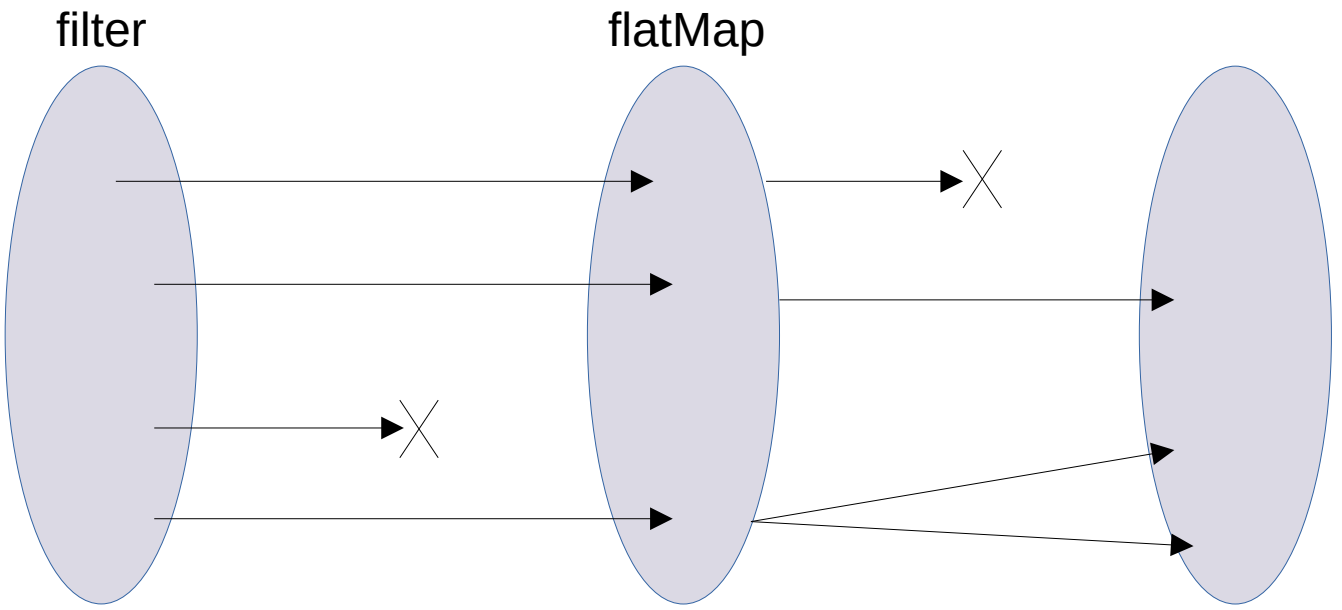
- boolean integrate(A state, E element, Downstream<T> downstream)

L’interface Downstream

- Correspond à l’opération intermédiaire suivante
- Similaire à un Consumer mais renvoie un boolean

Integrator

Une opération intermédiaire poussent ou pas des éléments dans le stage suivant

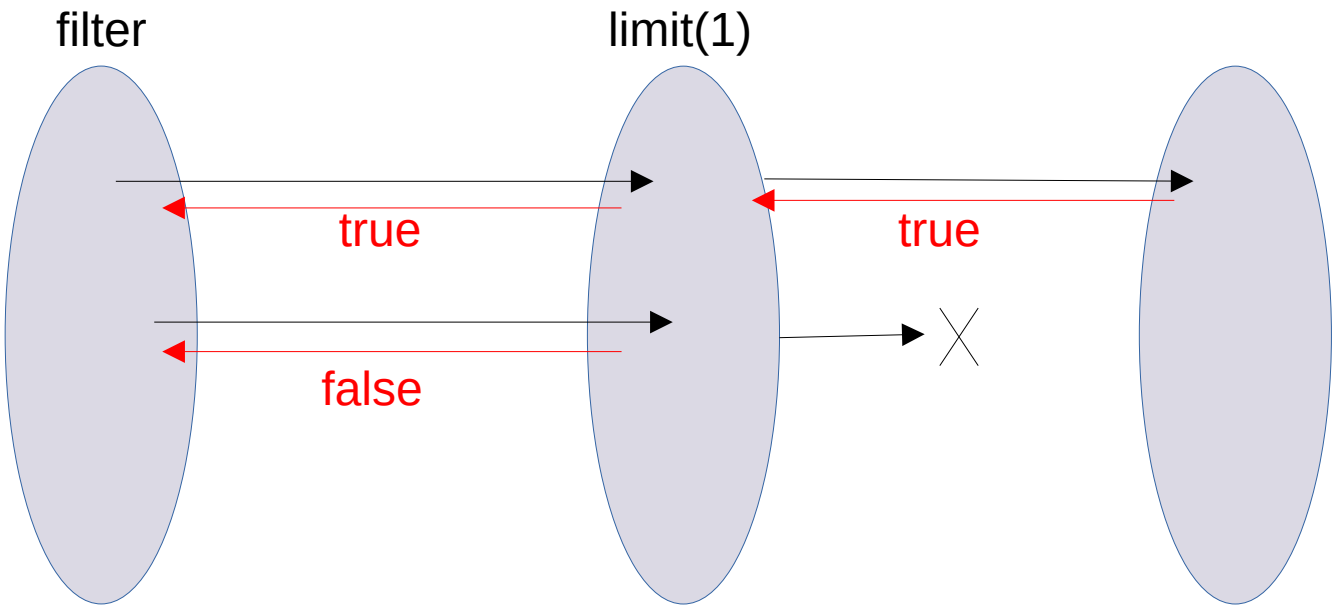


Exemple de filter()

```
<T> Gatherer<T, ?, T> filter(Predicate<? super T> predicate) {
    return Gatherer.of(
        (_, element, downstream) -> {
          if (predicate.test(element)) {
            return downstream.push(element);
          }
          return true;
        });
}
```

Integrator / back propagation

Une opération intermédiaire renvoie si il faut continuer (true) ou s’arrêter (false)



Exemple de takeWhile()

```
<T> Gatherer<T, ?, T> takeWhile(Predicate<? super T> predicate) {
    return Gatherer.ofSequential(
        (_, element, downstream) -> {
          if (predicate.test(element)) {
            return downstream.push(element);
          }
          return false;  // short-circuit
        });
}
```

Optimisation avec isRejecting()

L’interface Downstream

- a une méthode boolean push(T)
- a une méthode isRejecting()

- isRejecting() indique que l’opération intermédiaire suivante va renvoyé false indépendamment de l’élément envoyer
  - Évite la création d’un élément si celui-ci coûte chère

Optimization avec Integrator.greedy

- Si notre integrator n’est pas short-circuit, on peut l’indiquer à l’implantation
  - Integrator.ofGreedy(integrator)

- Cela permet à l’implantation de
  - Merger les Gatherers successifs
  - Merger un Gatherer suivi d’un Collector

Exemple de filter()

```
<T> Gatherer<T, ?, T> filter(Predicate<? super T> predicate) {
    return Gatherer.of(
        Gatherer.Integrator.ofGreedy(  // ofGreedy: optimization
            (_, element, downstream) -> {
              if (predicate.test(element)) {
                return downstream.push(element);
              }
              return true;
        }));
}
```

Gestion d’un état / initializer

- L’initializer initialise un état
  - Supplier<A> initializer
    - void get()

Exemple de limit()

On créé un *state* mutable

```
<T> Gatherer<T, ?, T> limit(int limit) {
  return Gatherer.ofSequential(
    () -> new Object() { int count; },
    (counter, element, downstream) -> {
        if (counter.count++ < limit) {
          return downstream.push(element);
        }
        return false;
      });
}
```

Gestion d’un état / finisher

- L’initializer initialise un état
  - Supplier<A> initializer
- Le finisher est appelé pour “vider” l’état si nécessaire
  - BiConsumer<A, Downstream<T>>
    - void accept(A, DownStream<T>)

Gestion d’un état / combiner

- L’initializer initialise un état
  - Supplier<A> initializer
- Le combiner merge deux états (si parallèle)
  - BinaryOperator<A>
    - A apply(A, A)
- Le finisher est appelé pour “vider” l’état si nécessaire
  - BiConsumer<A, Downstream<T>>

Exemple de count()

On ajoute un combiner et un finisher

```
<T> Gatherer<T, ?, Integer> count(int limit) {
  class Counter {
    private int count;

    Counter(int count) { this.count = count; }
  }
  return Gatherer.of(
      () -> new Counter(0),
      (counter, element, downstream) -> {
        counter.count++;
        return true;
      },
      (c1, c2) -> new Counter(c1.count + c2.count),
      (counter, downstream) -> downstream.push(counter.count));
}
```

Gatherer<E, A, T>

- L’initializer initialise un état
  - Supplier<A> initializer
- L’integrator pousse les éléments dans le stage suivant
  - Integrator<E, A, T> integrator
    - boolean integrate(A state, E element, Downstream<? super T> downstream)
- Le combiner merge deux états (si parallèle)
  - BinaryOperator<A>
- Le finisher est appelé pour “vider” l’état si nécessaire
  - BiConsumer<A, Downstream<T>>

```
interface Downstream<T> {
  boolean push(T element);
  boolean isRejecting();
}
```

Design sur 3 axes

- Notion d’ordre ?
  - Gatherer.ofSequential() vs Gatherer.of() + combiner
- Sans état vs avec état
  - (integrator) vs (initializer, integrator, finisher?)
- Short-circuit vs Greedy
  - Integrator vs Integrator.ofGreedy(integrator)

Exemple de windowFixed(size)

Il faut faire attention à bien allouer une nouvelle liste

```
<T> Gatherer<T, ?, List<T>> windowFixed(int size) {
  class State {
     ArrayList<T> list = new ArrayList<>();
  }
  return Gatherer.ofSequential(
      State::new,
      Gatherer.Integrator.ofGreedy((state, element, downstream) -> {
        var list = state.list;
        list.add(element);
        if (list;size() == size) {
          state.list = new ArrayList<>();
          return downstream.push(list);
        }
        return true;
      }),
      (state, downstream) -> {
        if (!state.list.isEmpty()) {
          downstream.push(state.list);
        }
      });
}
```

java.util.stream.Gatherers

Méthodes statiques des gatherers prédéfinies

- fold(Supplier<A>, BiFunction<A, E, A>)
  - Implique un ordre
- scan(Supplier<A>, BiFunction<A, E, A>)
  - Pousse les états intermédiaires du fold
- windowFixed(int size)
  - On découpe avec une fenêtre de taille size
- windowSliding(int size)
  - Découpage avec une fenêtre glissante
    (les sizes premiers, size seconds, size troisièmes, etc)

Problème des Gatherers

- Comme les Collectors, il n’y a pas de spécialization pour les types primitifs
- Les caractéristiques des Spliterators ne sont pas propagés/propageable (ahhh)