Java 8 Streams & Collectors
Speaker : José Paumard
Track : Patterns, performance, parallélisation
Jeudi 16/04/2014 13h30 – 16h30

Nouvelle API en Java 8
Lié aux Lambdas (cf. conf Rémi Forax)
Dans le livre Design Patterns du GOF, le pattern Iterator est décrit. Ce pattern prend du plomb dans l'aile avec les Collectors.

Conf en 2 grandes parties : slides + codes à base de 3 exemples
Questions sur Twiter #Stream8

Qu'est-ce qu'un Stream ? Interface paramétrée
A quoi ça sert ? Traiter petits et grands volumes de données. Efficace sur des petites collections / hashmap. Simplifier l'écriture du code.

Efficacement ?

1. exploiter le multicoeur
2. pipelines

Pourquoi l'API Stream n'a pas été mise directement dans l'API Collections ?
Ils auraient pu le faire. Arguments :

- arguments dans les collections
- les développeurs connaissent l'API collections

Pourquoi une nouvelle API :

- Ca permet d'avoir les mains libres
- Permet de ne pas polluer l'API Collection avec ces nouveaux concepts. Séparation.

Quel est le concept d'un Stream ?

1. Objet servant à définir des opérations : map, filtre …
2. Ne possède pas les données qu'il traite ➔ conséquences
    a. Se connecte à une source de données
3. S'interdit de modifier les données qu'il traite
4. Traite les données en « une passe ».
5. Optimisé du point de vue algorithmique (défini dans les spécs)

Il existe plein de patterns pour construire un Stream

Méthode référence = fonction lambda

System.out ::println
forEach prend un Consumer<T> qui est une interface fonctionnelle

Interface fonctionnelle

Une interface fonctionnelle peut être annotée avec @FunctionalInterface. Permet aux compilateurs de lever des erreurs dans l'IDE. Peut comporter des méthodes par défaut.
Exemple : dans Collection, la méthode stream a été ajoutée

Héritage multiple

Existait déjà sur les types.
Héritage multiple d'implémentation amené par Java 8
Héritage multiple d'états (pb du diamant) en C++ posant pb.
Erreur à la compilation lorsqu'on implémente 2 interfaces comprenant chacune la même méthode par défaut.
Corrections possibles :

1. Création d'une implémentation dans la classe (peut faire appel à A comme B) = > la classe gagne
2. On peut choisir l'implémentation la plus spécifique

Consumer<String> c1 = liste::add ;
Consumer<String> c2 = System.out ::println;

Consumer<String> c3 = c1.*andThen(*c2);

Méthode Peek

Peek permet de chainer les appels. Points d'intermédiaires. Prend un consommateur et renvoie un nouveau Stream.

Méthode Filter

Prend un prédicat en paramètre. Interface fonctionnelle Predicate + méthode boolean test(T t).
Un Predicate a des méthodes par défaut : and, or, negate.
Attention : ordre des méthodes importants lors de .or().and() …

Predicate.isEquals(« deux ») ;
Méthode statique de Stream : Stream.of

La méthode filter envoie un nouveau Stream.
Ce nouveau Stream ne comporte pas les données filtrées puisque les Stream ne portent pas les données.
liste.stream() et liste.filter() sont seulement des déclarations : aucune opération n'est réalisée.

L'appel à filter() est un appel lazy.
D'une façon générale, un appel qui retourne un Stream est lazy => ce qui est implicite puisqu'un Stream n'a pas de donnée.
Sur un Stream, il n'y a quasiment que des appels lazy.
Si on avait mis les Stream dans les Collections, on aurait les 2 concepts mélangés.

Javadoc : une opération qui retourne un Stream est une opération intermédiaire.

Peek() est un appel intermédiaire : une simple déclaration de fonction.

Opérations de mapping (suite)

Opération de mapping retourne un Stream
L'interface Function<T, R> est fonctionnelle : apply, compose, andThen, identity

Opération flatMap

Permet de mapper et de mettre à plat. Prend une fonction. Construit un Stream de Stream.
La fonction transforme des éléments T en éléments Stream<R>

Réduction

Dernière étape d'opérations. Etape où sont effectués les traitements.
stream.reduce(0, (age1,age2) => age1+age2)
A chaque fois on prend les 2 premiers éléments d'un tableau qu'on somme. On prend ensuite le 3^{ième} élément qu'on ajoute à la somme.
Terme de folding
Les opérations associatives sont très facile à paralléliser => on prend un tableau qu'on coupe en deux.
Lorsqu'on développe des fonctions de réduction non associatives => résultats faux sans erreur de compilation.
Dans l'exemple ci-dessus, le 0 est la valeur par défaut pour le cas où la source est vide. 0 est la valeur par défaut raisonnable de la somme.

Quel est le type de retour pour la réduction ?
Lorsqu'on paralléliste et que T1 est vide. Donc T = T2.
La réduction d'un ensemble vide est l'**élément neutre**.
Zéro n'est pas l'élément neutre pour la réduction max. Il n'y a pas d'élément neutre pour le max. Il n'y a donc pas de valeur par défaut du max. Quel type de retour du max ?
Si on pend int, alors la valeur par défaut est 0.

Nouveau concept dans le JDK : **Optional**
Il se peut que le Integer n'est pas de valeir

Optional<String> opt =
opt.isPresent
opt.get()
opt.orElseThrow(MyException ::new)

On a donc 2 versions du reduce : avec et sans valeur par défaut.

Une réduction ne retourne pas de Stream.
Réductions : min, max, count, allMath(), noneMatch, anyMatch, findFirst
Dans tous les cas, une réduction retourne une valeur. Déclenche un traitement. **Opération terminale**.

Exempe : .min(Comparator.naturalOrder())
Une fois un Stream utilisé, il faut créer un nouveau Stream. Il est fusillé.

.allMatch(length < 20)
Avec une boucle on peut sortir tout de suite : avantage.

Stream : un objet qui permet de définir des traitements sur des jeux de données arbitrairement grands.

Comment un Stream est-il fait ?
définie dans l'interface Collections. Commun à toutes les collections. Utilise un **Spliterator** spécifique à chaque type de collections (ex : ArrayListSpliterator). La partie algorithmique est définie dans le Head. On n'y touche pas.

Fonction de l'interface Spliterator
De nombreuses méthodes par défaut (ex : estimateSize …)
Méthode int characteristics() : **bits** ORDERED, SIZED, DISTINCT, SORTED, IMMUTABLE …
Les opérations changent la valeur des **caractéristiques**.
Ces bits permettent d'optimiser l'opération terminale.

Apparté sur le Comparator

NaturalOrderComparator : exemple étrange de SINGLETON.
Comparator.comparing(Person ::getLastName).thenComparing(Person ::getFirstName)

Opérations stateless / statefull

Opérations avec états
persons.limit(1_000) => sélectionne les 1000 premières personnes.
Cette fonction nécessite un compteur partagé par tous les threads.
Le compteur est un point de contention. Attention aux perfs.

De la même manière, lorsque l'ordonnancement doit être conservé, la parallélisation va être plus compliquée.

Stream & Performance

En complément du Stream<T> , il existe des **IntStream** permettant d'éviter le boxing / unboxing.
Utilisation possible de la méthode mapToInt(Integer ::getValue)
La méthode sum() existe sur le IntStream.
Le max() renvoie un Optional, d'où la présence d'un OptionalInt

On peut utiliser un ForkJoinPool(2)

Fjp.submit().get() ;

Réduction

Peut être vue comme une agrégation au sens SQL.
Cette définition peut être élargie.

Exemple : la somme
Un « conteneur » résultat
Une opération « ajouter »
Valeur initiale = élément neutre de la réduction

Définition :

1. Création d'un conteneur résultat => Supplier (ex : StringBuffer)
2. Ajout d'un élément à un container => Function (ex : append entre StringBuffer et String)
3. Fusion de deux containers => Function (append)

La méthode terminale collect peut encapsuler ces 3 opérations => Collector
La réducition des chaines de caractères existe déjà : Collectors.joining.
La classe Collectors propose plein de méthodes statiques : groupingBy, mapping, joigning.
Permet d'empiler les Collectors les uns dans les autres

Dans la Javadoc, notion de « downsream collector » => ce n'est pas un Collector mais un Stream.

2ième partie de la conférence

Cas d'utilisation sur github.com/JosePaumard/jdk8-lambda-tour

Sur le MapEntry, on a des comparateurs clés en main

Map.Entrey.comparingByValue()

Comparator.comparing(entry --> -entry.getValue()) pour un comparator descendant