

本日の目的

Java8のStreamAPIでどんなことができるようになったのかを説明し、こんな機能があるんだなというレベルで持ち帰ってもらう。

そのため、駆け足での説明になってしまう。

資料は、寄せ集め、Webに存在するページを拝借し、AdobeのAcrobatでつぎはぎなので、不要な文言が入っているのでご了承ください。

Stream API

Stream APIの概要

■仕様

- JEP107 **Bulk Data Operations for Collections**
- IOストリーム とかあるのに、Stream APIとか言っちゃうんだ、、、

■できるようになったこと

- コレクション等一連の要素の集まりに対して、ラムダ式を使用した抽出/演算/変換等の一括操作の提供、および、逐次処理・並列処理(※)の提供

■影響

- コレクションに対する操作をfor/whileを使った方式から、Stream APIを使用した新しい方式で書けるようになる。
- 利点
 - 宣言的であり、読みやすく、間違いが少ない
 - 並列処理への切り替えが容易
 - 配列・リスト・入出力などが同じ方法で扱える
- 欠点
 - Stream APIを使いこなすには覚える事が多い
 - 実用するには色々と機能が足りてない

例題

■ 例題

- 文字列のリストから整数とみなせる文字列を抽出し、整数に変換し、
正の整数のみを抽出し、
3倍する。

従来の方法(example.StreamBasic.java)

■ 例題

- 整数リストから偶数の要素のみ抽出し、数を2倍した要素を取得す

```
List<String> list =  
    Arrays.asList("A001", "100", "-200", "ABC", "92");  
List<Integer> res = new ArrayList<>();  
for (String s : list) {  
    if (s.matches("[ -+]?¥¥d+")) {  
        int i = Integer.parseInt(s);  
        if (i > 0) {  
            res.add(i * 3);  
        }  
    }  
}  
System.out.println(res); //300, 276
```

- このような**手続き型**のコードは、条件分岐がネストしたすと、コードをよく見ないと、要件に合致する処理が書いてあるかわからない。

Stream API による解法(example.StreamBasic.java)

■ 例題

- 整数リストから偶数の要素のみ抽出し、数を2倍した要素を取得す

```
List<String> list =  
    Arrays.asList("A001", "100", "-200", "ABC", "92");  
List<Integer> res2 = list.stream()           // Streamを, 生成  
    .filter(s -> s.matches("[ -+]?¥¥d+")) // 整数のみ抽出  
    .map(s -> Integer.parseInt(s))         // 文字列→整数  
    .filter(i -> i > 0)                     // 正の整数のみ抽出  
    .map(i -> i * 3)                         // 3倍する  
    .collect(Collectors.toList());          // 結果をListとして取得  
System.out.println(res2);
```

■ 解説

- Streamという一連の要素の集まりに、抽出(filter)・演算(map)のような操作を連結して繰り返し処理の内容を組み立てる。最終的にcollectメソッドで、操作した要素をList等に変換する。よくわからなければ、SQLでイメージしてもらいたいかもしれません。
- 操作の内容はラムダ式で何をするのかを書く。
- なんとなく、例題の処理内容とラムダ式の内容が揃ってる感じがしませんか。
- あと変数も少ないですね。

Stream APIとは

■ 大雑把にいうと

- ものすごいイテレーター。
 - 繰り返し処理に、条件抽出や変換、件数指定などの操作を幾らでも設定できる。
- 繰り返し処理の方法はStream APIの中に隠蔽される。
 - 今まで繰り返しは、for/while/拡張for文などを選んでいたが、その必要がない。
 - 繰り返し処理の最適化をStreamに任す。
 - プログラマはStreamにどのような操作を行うか中心に書く。(宣言的なプログラム)

■ 性質

- コレクションとは異なるクラス
 - 生成 - Streamを使うには既存のコレクションや配列から、別途変換が必要
- Stream の各種メソッドは大別して以下の2種類に分かれる。
 - 中間操作 - 生成で構築した集合に対する演算を適用する。
 - 終端操作 - ストリームの集合から、コレクション、配列等への変換を行う。
- 逐次処理/並列処理
 - 繰り返し処理の方法を簡単に並列処理にできる。
(処理の仕方を、プログラマが書く必要がないため。)
- 遅延処理
 - 繰り返し処理は最後に一度だけ実行される。

生成・中間操作・終端操作

- 生成で様々なオブジェクトからStreamを作る
- 中間操作でStreamに設定を追加する
- 終端操作でStreamを処理して結果を得る
 - 繰り返し処理や並列処理の詳細はStreamの中だけで完結する



Stream の生成

■ 生成

- 色々なデータからStreamを生成するメソッドが、多数追加された。
 - コレクションから - Collection#stream, parallelStreamメソッド
 - 配列 - Arrays#stream
 - IOストリーム、ファイル - BufferedReader#lines, Files#lines
 - 任意の可変長引数で - Stream#of
 - 文字列から文字のストリーム - String#codePoint
 - 無限数列 - Stream#iterate, Stream#repeat
 - 範囲生成(1から100までとか) - IntStream#range
 - 他多数
- ストリームを構築するコードを書く必要がありますが、逆に、元がコレクション、配列、ファイル、その他、何であっても一旦ストリームにしてしまえば、後は同じ方法で扱う事ができる。

Stream の中間操作

■ 中間操作(抜粋)

- Streamの要素に対する何らかの演算の指定を行うメソッド。

中間操作は必ずStreamが戻り値なので、メソッドチェーンで連結できる。

- filter - 条件ラムダ式に一致する要素のみを抜き出す。
 - map - 要素をラムダ式に適用して計算・変換する。
 - flatMap - 要素をStreamを生成するラムダ式に適用し、要素の増減を行う。
 - **parallel** - 繰り返し処理を並列処理で行う指示をする。
 - limit - 繰り返し処理を行う件数を制限する。
 - skip - 繰り返し処理を件数分スキップする。
-
- 以下は**状態がある中間操作**と呼ばれ、全ての要素の評価が終わらないと実行できない。
そのため、性能に影響を与える可能性がある。
 - sort - 要素を並べ替える。
 - distinct - 要素の重複を除外する。

Stream の終端操作

■ 終端操作 (抜粋)

- Streamから結果を生成する操作。

終端操作を行った時点でStreamに設定された中間操作が順次実行され、終端操作の内容で結果が作られる。

1つのStreamには、1回だけ終端操作を行うことができる。

- **anyMatch** - ラムダ式の条件に合致する要素があるかbooleanで返す。※
- **allMatch** - 全ての要素がラムダ式の条件に合致するかbooleanで返す。※
- **max, min** - Comparatorを渡し、要素の最大・最小値を返す。
- **reduce** - 全ての要素の合計値を出すなど、1つにまとめた結果を返す。
- **forEach** - コンソール出力等、任意の副作用を実行する。
- **count** - 件数取得
- **findFirst** - 最初の1件を返す ※
- **collect** - 汎用的な集積操作。Collectorsユーティリティによく使う操作が定義済み。
 - **Collectors#toList** - 要素をListへ変換する。
 - **Collectors#joining** - 要素を1つの文字列にする。接続句を指定可能。
 - **Collectors#partitionBy** - 要素をラムダ式の条件を満たすものと満たさないものに分割する。
 - **Collectors#groupingBy** - 要素をラムダ式の条件にしたがって複数のグループに分割する。
- ※ **anyMath, allMath, findFirst**などはショートサーキット評価であり、条件が一致した時点で要素を全て評価せず処理を打ち切る。

ラムダ式がいきなり出てきたので、ラムダ式について説明する。

ラムダ式 (Lambda Expressions)

ラムダ式は、関数型インタフェースの実装匿名クラスの簡易記法です。ラムダ式は以下の構文で記述します。

```
(引数) -> { 処理 }
```

ラムダ式の引数は、実装する関数型インタフェースのメソッドの引数と同じになります。

ラムダ式導入のメリットは、後述する Stream API でラムダ式を使用することにより、内部イテレータを使用した並列処理が簡潔に記述可能になることです。

ラムダ式の構文

基本形

```
MyFunction func = (int x, int y) -> { return x + y; };
```

引数の型が自明な場合は引数の型を省略可能

```
MyFunction func = (x, y) -> { return x + y; };
```

引数が1つの場合は引数リストの括弧を省略可能

```
MyFunction func = x -> { return x; };
```

処理が1文の場合は波括弧と return を省略可能

```
MyFunction func = (x, y) -> x + y;
```

実質的に final

ラムダ式では匿名クラスと同様に、エンクロージング環境のローカル変数を参照可能ですが、final 修飾子が付与されている変数しか参照することができません。

但し、Java SE 8 では、値の変更が行われていない変数は実質的に final として扱われます。

```
public static void main(String[] args) {  
    // 実質的 final なローカル変数  
    int value1 = 10;  
    List<Integer> values = Arrays.asList(10, 20, 30);  
    // ラムダ式  
    values.stream().  
        forEach(value2 -> System.out.println(value1 + value2));  
    // 20, 30, 40  
}
```

Stream API による解法(example.StreamBasic.java)

■ 例題

- 整数リストから偶数の要素のみ抽出し、数を2倍した要素を取得す

```
List<String> list =  
    Arrays.asList("A001", "100", "-200", "ABC", "92");  
List<Integer> res2 = list.stream()           // Streamを, 生成  
    .filter(s -> s.matches("[ -+]?¥¥d+")) // 整数のみ抽出  
    .map(s -> Integer.parseInt(s))         // 文字列→整数  
    .filter(i -> i > 0)                     // 正の整数のみ抽出  
    .map(i -> i * 3)                         // 3倍する  
    .collect(Collectors.toList());          // 結果をListとして取得  
System.out.println(res2);
```

■ 解説

- Streamという一連の要素の集まりに、抽出(filter)・演算(map)のような操作を連結して繰り返し処理の内容を組み立てる。最終的にcollectメソッドで、操作した要素をList等に変換する。よくわからなければ、SQLでイメージしてもらいたいかもしれません。
- 操作の内容はラムダ式で何をするのかを書く。
- なんとなく、例題の処理内容とラムダ式の内容が揃ってる感じがしませんか。
- あと変数も少ないですね。

関数の外だしについて説明する。
関数の種類について説明する。

```
private final Function<String, Predicate<MiddleCategory>> mKeyMach  
    = searchKey -> mc -> mc.getKey().equals(searchKey);
```

```
public void searchMacthConditon(String searchKey, List<MiddleCategory> middleCategoryList) {  
  
    middleCategoryList.parallelStream().filter(mKeyMach.apply(searchKey)).findFirst();  
}
```

もう少し応用を話す。AssetUtil

⇒ AssetUtilRefactored

標準の関数型インターフェース

| 種類 | 概要 | インターフェース名 |
|---------------|-----------------------------------|--|
| Function | 引数を1つまたは2つ受け取って結果を返却します。 | Function IntFunction LongFunction |
| Consumer | 引数を1つまたは2つ受け取って結果は返却しません。 | Consumer IntConsumer LongConsumer |
| Supplier | 引き数を受け取らず結果を返却します。 | Supplier IntSupplier LongSupplier |
| Predicate | 引数を1つまたは2つ受け取ってboolean型の結果を返却します。 | Predicate IntPredicate LongPredicate |
| UnaryOperator | 引数を1つ受け取って同じ型の結果を返却します。 | UnaryOperator IntUnaryOperator LongUnaryOperator |

以下にて、filter等の引数について、説明する。
<http://www.task-notes.com/entry/20150511/1431313200>

```
//Functionを使用して、戻り値としてfilterの引数であるPredicateを返す。  
private final Function<String, Predicate<MiddleCategory>> mKeyMach  
    = searchKey -> mc -> mc.getKey().equals(searchKey);
```

filter : 抽出

引数 : Predicate<T> / 戻り値 : Stream<T>

☑ 関数型インターフェース

filterメソッドはPredicate<T>を引数に取り、Streamのオブジェクトから条件に一致する（Predicateがtrueを返す）オブジェクトを抽出します。Predicateは判定を行うための関数型インターフェースです。実装が必要なメソッドは `boolean test(T t)` で引数を1つ受け取り、booleanを返します。

☑ filterの使い方

IntegerのStreamから数値が200以上のデータを抽出するサンプルです。

```
List<Integer> prices = Arrays.asList(100, 200, 300, 400, 500);  
// ラムダ式  
prices.stream().filter(pri -> pri > 200).forEach(System.out::println);  
  
// 匿名クラス  
prices.stream().filter(new Predicate<Integer>() {  
    @Override  
    public boolean test(Integer t) {  
        return t > 200;  
    }  
}).forEach(System.out::println);
```

Streamの戻りはOptionalであることを説明する。

```
Optional<MiddleCategory> middleCategory  
    = middleCategoryList.parallelStream().filter(mKeyMach.apply(searchKey)).findFirst();
```

以下のサイトでOptionalについて見て見る。

<http://www.task-notes.com/entry/20150708/1436324400>

Enumのサンプルを見せたりする。

toMapについて触れる。

```
//listのTOPにあるクラスの項目以外をkeyにする場合
Map<String, List<MiddleCategory>> res = cateList.stream().collect(
    Collectors.toMap(Category::getKey, k -> k.getMiddleCategoryList())
);

//listのTOPにあるクラスの項目をkeyにする場合
Map<Object, List<Category>> resGroupBy = cateList.stream().collect(
    Collectors.groupingBy(k->k.getKey()));
```

HashMapでnewしてる

```
*/
public static <T, K, U>
Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,
    Function<? super T, ? extends U> valueMapper) {
    return toMap(keyMapper, valueMapper, throwingMerger(), HashMap::new);
}
```

collectについて詳しい内容は、以下を参照。（詳しくは、後で見て）

http://www.atmarkit.co.jp/ait/articles/1407/28/news023_2.html

grouping,toMapなどどんな種類があるのか、javadocを見る。

<https://docs.oracle.com/javase/jp/8/docs/api/java/util/stream/Collectors.html>

joinについて説明。

joiningはStringBuilderを使用しているsrcを見せる。

以下のサンプルにて説明する。

code.collections.fpij.PrintList.java

// sortについて説明する

version8.TestLogic.javaにて説明する。

// joinについて説明する

joiningはStringBuilderを使用しているsrcを見せる。以下のサンプルにて説明する。
code.collections.fpij.PrintList.java

//reduceについて説明する

<http://d.hatena.ne.jp/gloryof/20140420/1397972939>時間が足りない場合は、割愛する。

// flatMapについて説明する

参考サイト

<http://www.task-notes.com/entry/20150513/1431486000>

時間が足りない場合は、割愛する。

一応、Limit offSetの例も準備

sample.logic.LimitOffSet.java

```
*/  
private List<Category> getTargetList(List<Category> cateList, final int offset, final int limit) {  
    return cateList.stream().skip(offset).limit(limit).collect(Collectors.toCollection(ArrayList::new));  
}
```

Mapの新機能（30分）

javadocにて追加機能の確認

<https://docs.oracle.com/javase/jp/8/docs/api/java/util/Map.html>

以下、サンプルコードを一緒に見る

<http://d.hatena.ne.jp/nowokay/20130523>