
Stream API

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
持ち帰っていただくことを目的としております。
そのため、駆け足での説明になってしまいます。

また、資料は、Webに存在するページを拝借し、AdobeのAcrobatで
繋ぎ合わせて作成している為、不要な文言も入っています。
その点、ご了承ください。

もくじ

1.Stream API の概要

2.簡単サンプルをもとにStreamのイメージを確認

3.各機能の詳細・仕組み

4.実用的なサンプルをもとにStream APIの機能について説明

Stream APIの概要

■ 出来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

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倍にする。

従来の方法

■ 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

■ 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る

```
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

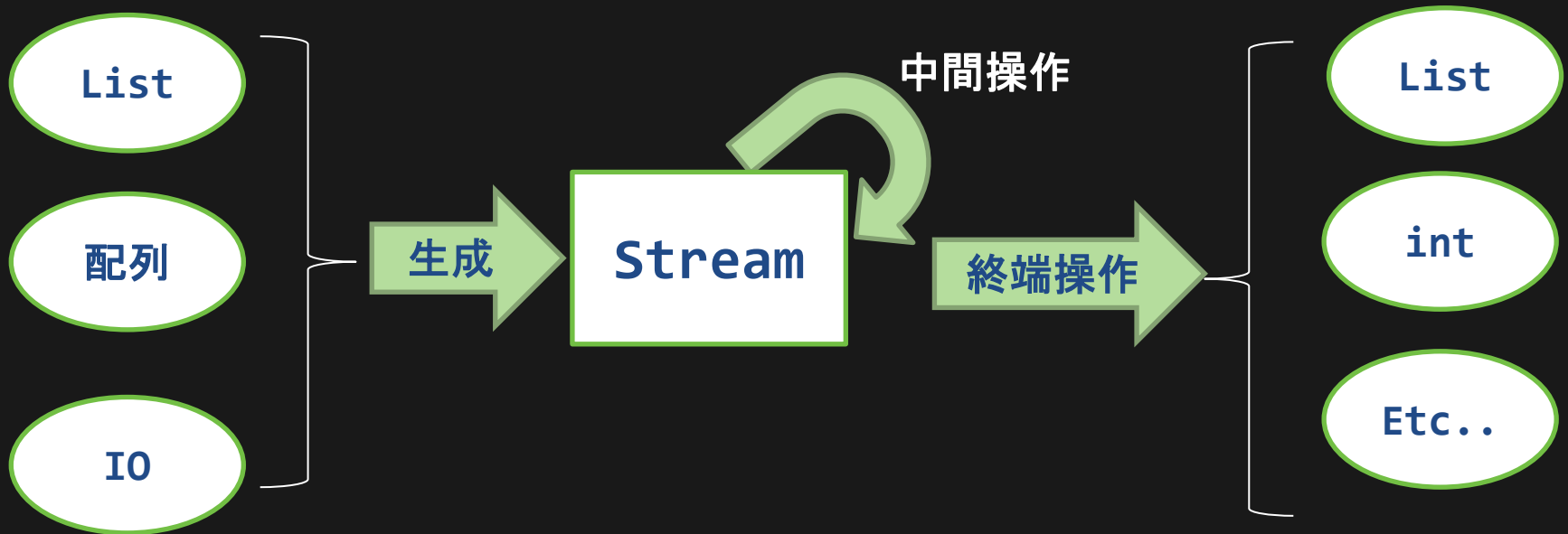
- ものすごいイテレーター。
 - 繰り返し処理に、条件抽出や変換、件数指定などの操作を幾らでも設定できる。
- 繰り返し処理の方法はStream APIの中に隠蔽される。
 - 今まで繰り返しは、for/while/拡張for文などを選んでいたが、その必要がない。
 - 繰り返し処理の最適化をStreamに任ず。
 - プログラマはStreamにどのような操作を行うか中心に書く。(宣言的なプログラム)

■ 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 – streamの結果をforEachにて任意のループ処理を行う。
- 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, 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。

Stream の終端操作（補足）

- `findFirst()` と `anyMatch()` の差は、Optional を生成するか否かのみ

- `findFirst()` と `findAny()`

`findAny` は、ストリームが保持する要素のうち、いずれかの要素を返す。

`findFirst` は最初の要素を返すという特徴がある為、

`parallelStrea()` を使用した場合、いずれかの中間処理に対する要素を返す `findAny` を使用したほうが、結果を早く受け取れる可能性がある。



ラムダ式の書き方について
みてみよう！

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1

■ 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 return を省略可能

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


ラムダ式の構文2

■ 引数なし、戻り値あり

```
() -> new ArrayList();
```

■ 引数1つ、戻り値なし

```
str -> System.out.println(str);
```

■ 引数1つ、戻り値あり (変換系)

```
s -> s.toUpperCase(s)
```

■ 引数1つ、戻り値あり (真偽値)

```
s -> "".equals(s)
```

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・ラムダ式の基礎



1. 標準の関数インターフェースについて
2. 中間処理の仕組み・機能について
3. ラムダ式を使用した共通機能化
4. Optionalの基礎
5. enumサンプルによるOptionalの使用例
6. Streamへの変換例
7. Collectの機能について
8. 既存のListへの追加方法について

JavaのSrcを見てみよう！



実はfilterやmapは、
Java8から追加された
関数型インターフェースを引数としている。

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

filter,mapの引数について

参考サイト:

<http://www.task-notes.com/entry/20150511/1431313200>

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と一致する(Predicateがtrueを返す)要素を抽出します。PredicateはPredicate<T>インターフェースで定義され、抽象メソッドはboolean test(T t)で定義されています。

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);
```

ラムダ式を使用した共通機能化について

サンプル:

- AssetUtil ⇒ AssetUtilRefactored
- Predicateを返すFunction

```
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();  
}
```

Optionalの基礎

FindFirstの戻り値はOptional

参考サイト:

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



enumサンプルによる Optionalの使用例

サンプル:

OtherObjectStream.java



Streamへの変換サンプル



Collectの機能について

- toMap : HashMapのインスタンスである

SampleCode : FromListCnvToMap.java

- MapのStream処理サンプル

SampleCode : FromListCnvToMap.java

- 参考サイト :

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

- javadoc :

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

既存のListへの追加方法について

サンプル:

DifferentListConvert.java

```
private void addExistingList(List<Category> cateList, List<Category> originList, String seachKey) {  
    cateList.stream().filter(category -> category.getKey().equals(seachKey))  
        .collect(Collectors.toCollection(() -> originList));  
}  
  
private void addExistingList2(List<Category> cateList, List<Category> originList, String seachKey) {  
    cateList.stream().filter(category -> category.getKey().equals(seachKey)).forEach(originList::add);  
}
```

実用サンプルによる説明



1. 複雑なソート処理
2. 文字列結合の進化 (joining)
3. 深い階層の処理をフラットに (flatMap)
4. ページングサンプル (limit, offset)

複雑なソート処理

サンプル：
version8.TestLogic.java

```
resultList = people.stream()
    .sorted(comparing(Person::getName, nullsLast(naturalOrder()))).thenComparing(Person::getAge))
    .collect(toList());

printPeople("Age,Name共に昇順: ", resultList);
// 完成型 (降順) さらに降順と昇順を混在させるには以下のようにする
resultList = people.stream()
    .sorted(comparing(Person::getName, nullsLast(reverseOrder()))).thenComparing(Person::getAge))
    .collect(toList());
```

```
private <E> List<E> sortList(List<E> list, Comparator<E> comparator) {
    return list.stream().sorted(comparator).collect(toList());
}
```

文字列結合の進化(joining)

サンプル:

`code.collections.fpij.PrintList.java`

深い階層の処理をフラットに (flatMap)

参考サイト:

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

サンプル:

NestListClassSearch.searchByFlatMap()

```
// もう一段階ネストしたクラスのリストを検索する。
final String searchKeyForLast = "BGK";
Optional<LastCategory> lastCategory = cateList.stream()
    .flatMap(c -> c.getMiddleCategoryList().stream())
    .flatMap(mc -> mc.getLastCategoryList().stream())
    .filter(lc -> lc.getKey().equals(searchKeyForLast))
    .findFirst();

// パフォーマンスを意識するとこうなることもある。
cateList.stream().flatMap(c -> c.getMiddleCategoryList().stream())
    .filter(mc -> mc.getKey().equals("BG"))
    .flatMap(mc -> mc.getLastCategoryList().stream())
    .filter(lc -> lc.getKey().equals(searchKeyForLast))
    .findFirst();
```


ページングサンプル (limit,offset)

サンプル:
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));  
}
```

その他



1. Streamのデバッグ方法について
2. Stream の流用について
3. java.util.Mapに追加された機能について
4. reduceについて

peekについて

Streamのデバック

Streamの流用について

参考サイト:

<http://d.hatena.ne.jp/nowokay/20130504>

```
27
28     Stream<String> namesStream = Arrays.asList("hoge hoge", "foo bar", "naoki", "kishida").stream();
29     System.out.println( namesStream.anyMatch(s -> s.length() > 7)); //true
30     System.out.println( namesStream.noneMatch(s -> s.length() > 7)); //true
31
```

Problems Javadoc Declaration Search Console Debug Call Hierarchy

<terminated> LimitOffSetTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/bin/java (2017/01/30 18:07:40)

true

Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
at java.util.stream.ReferencePipeline.noneMatch(ReferencePipeline.java:459)
at sample.logic.LimitOffSet.execute(LimitOffSet.java:38)
at test.exec.LimitOffSetTest.main(LimitOffSetTest.java:10)

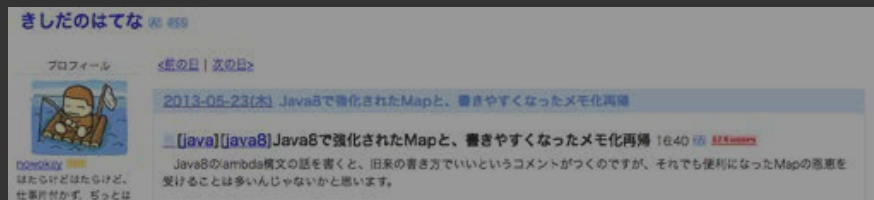
java.util.Mapに追加された機能

javadoc:

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

参考サイト:

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



reduceについて

サンプル:

<http://d.hatena.ne.jp/gloryof/20140420/1397972939>

ジェネリックメソッド活用のサンプル

```
/**
 * 指定のキーに対するKubunをimplementsしているenumの型を返す。
 *
 * @param clazz
 * @param keyValue
 * @return Optional<E> 区分値の型
 */
private <E extends Enum<E>> Optional<E> get(Class<E> clazz, String keyValue) {

    return Stream.of(clazz.getEnumConstants())
        .filter(k -> k instanceof kubun && ((kubun) k).getKey().equals(keyValue)).findFirst();
}
```

```
/**
 * Comparatorの再利用
 *
 * @param list
 * @param comparator
 * @return Sortedリスト
 */
private <E> List<E> sortList(List<E> list, Comparator<E> comparator) {
    return list.stream().sorted(comparator).collect(toList());
}
```

その他参考にしたサイト

- ・ Java8の追加機能の紹介スライド

[http://www.slideshare.net/minazou67/java-se-reintroduction?
qid=2630454e-e0ff-4f7b-
a643-287ec99bee90&v=&b=&from_search=1](http://www.slideshare.net/minazou67/java-se-reintroduction?qid=2630454e-e0ff-4f7b-a643-287ec99bee90&v=&b=&from_search=1)

- ・ Java8勉強会スライド

[http://www.slideshare.net/kentaromaeda581/java8-40752729?
qid=7ce924f2-
ca46-45b7-861d-46c3c2b5bc64&v=&b=&from_search=2](http://www.slideshare.net/kentaromaeda581/java8-40752729?qid=7ce924f2-ca46-45b7-861d-46c3c2b5bc64&v=&b=&from_search=2)

おすすめ本

<http://www.oreilly.co.jp/books/9784873117041/>



応用は、この本の後半
を読んでね。