Lambda 道場 問題編

Lambda 道場 問題編

1. はじめに

Java SE 8 で Project Lambda が導入されました。

Project Lambda の Lambda 式や Stream API を使用することで、関数型言語のアイデアを導入することができ、内部イテレータによって処理を行うことが可能になります。

Project Lambda の導入によって、今までの for 文や拡張 for 文といった外部イテレータから、内部イテレータで処理を記述することになり、Java のプログラミングスタイルが大きく変化します。

そこで、JJUG CCC のハンズオンでは Project Lambda の Lambda 式や Stream API を何度も書いていただきます。習うより慣れろで、繰り返し書くことにより内部イテレータの書き方も身についていくはずです。

本ハンズオンでは、プログラムの断片を提示していきます。それを Lambda 式や Stream で書き換えていってください。PC はなくても大丈夫。鉛筆で直接書き換えていってください。

もし、余裕があるのであれば、家に帰ってから、ぜひ PC で実行し直してみてください。使用した問題、解答は GitHub の LambdaDojo プロジェクトに置いておきます。

LambdaDojo https://github.com/skrb/LambdaDojo (https://github.com/skrb/LambdaDojo)

2. Lambda 式

Project Lambda で最も重要なのは内部イテレータを実現する Stream API です。しかし、Lambda 式が書けないと、Stream API も冗長な匿名クラスで書かざるをえません。Lambda 式で記述することによって、Stream API も簡潔に書けるようになります。

Lambda 式は関数型インタフェースを実装した匿名クラスの簡易的な記述法です。関数型インタフェースは実装すべきメソッドが単一の関数で @FunctionalInterface でインタフェースが修飾されています。

Lambda 式は (引数) -> { 処理 } の形式で表します。引数は関数型インタフェースの実装すべきメソッドの引数を表し、処理はそのメソッドの実体を表しています。

そこで、まず匿名クラスを Lambda 式を書き換えてみてましょう。

2-1. Lambda 式で書き換えてみましょう

```
        Comparator<Integer> comparator1 = new Comparator<Integer>() {
            @Override
            public int compare(Integer x, Integer y) {
                return x - y;
            }
        };
```

2-2. Lambda 式で書き換えてみましょう

```
        Callable<Date> callable1 = new Callable<Date>() {
            @Override
            public Date call() throws Exception {
                return new Date();
            }
        };
```

2-3. Lambda 式で書き換えてみましょう

```
        Runnable runnable1 = new Runnable() {
            @Override
            public void run() {
                doSomething();
            }
        };
```

2-4. Lambda 式で書き換えてみましょう

```
    @FunctionalInterface
    public interface Doubler<T extends Number> {
        T doDouble(T x);
    }

    ...

        Doubler doubler = new Doubler() {
            @Override
            public Double doDouble(Double x) {
                return 2.0 * x;
            }
        };
```

2-5. Lambda 式で書き換えてみましょう

次のプログラムは JavaFX のプログラムです。ボタンをクリックするとカウンタが増加するというだけの単純なプログラムですが、ここで使われている匿名クラスを Lambda 式で書き換えてみましょう。

```
public class LambdaForJavaFX extends Application {
    private int count;

    @Override
    public void start(Stage stage) {
        stage.setTitle("JavaFX Lambda");

        // カウンタを表示するラベル
        final Label counterLabel = new Label(String.valueOf(count));

        Button button = new Button("Count");

        // ボタンをクリックすると、カウントアップする
        button.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                count++;
                counterLabel.setText(String.valueOf(count));
            }
        });

        // 垂直方向にラベルとボタンを配置するコンテナ
        VBox root = new VBox(10);
        root.setPadding(new Insets(10));
        root.getChildren().addAll(counterLabel, button);

        Scene scene = new Scene(root, 100, 100);
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String... args) {
        launch(args);
    }
}
```

3. for 文の変換 (Iterable)

Lambda 文を書くことに慣れたら、外部イテレータを内部イテレータで書き直してみましょう。

最も単純な方法は、Iterable インタフェースに追加された forEach メソッドです。forEach メソッドを使用することで、拡張 for 文を内部イテレータに変換することが可能です。

3-1. forEach メソッドで書き換えてみましょう

```
        List<String> strings = Arrays.asList("a", "b", "c", "d", "e");

        StringBuilder builder = new StringBuilder();
        for (String s: strings) {
            builder.append(s);
        }
        System.out.println(builder.toString());
```

3-2. forEach メソッドで書き換えてみましょう

```
        List<Integer> numbers = Arrays.asList(10, 5, 2, 20, 12, 15);

        int sum = 0;
        for (Integer number: numbers) {
            sum += number;
        }
        System.out.println(sum);
```

3-3. forEach メソッドで書き換えてみましょう

拡張 for 文でない、単純な for 文も forEach メソッドで書き換えることができます。

ただし、ループカウンターを作成しなければなりません。0 から 9 までの数値の入った配列やリストを作成してもよいのですが、それよりもプリミティブ型に対応したストリームを使う方がスマートです。

ここではループカウンタが int なので、IntStream インタフェースを使用することができます。IntStream インタフェースには、範囲を指定して IntStream オブジェクトを生成する range メソッドが定義されていますので、これを利用します。

Stream インタフェースや IntStream インタフェースなどにも forEach メソッドがあるので、それを使用してみましょう。

```
        for (int i = 0; i < 10; i++) {
            System.out.print(i);
        }
        System.out.println();
```

4. for 文の変換 (Stream)

最後は、ストリームです。

ストリームは様々な処理を行うことができるイテレータです。Iterable インタフェースのように、自身ではオブジェクトを保持せずに、元となるコレクションなどの要素をイテレートしていきます。

ストリームにはオブジェクトを対象とした Stream インタフェース、プリミティブ型を対象にした IntStream インタフェース、LongStream インタフェース、DoubleStream インタフェースの 4 種類があります。

基本的なメソッドは同じですが、sum メソッド、max メソッドなどプリミティブ型が対象のストリームにしかないメソッドもあります。

はじめから様々な処理を行っている for 文を変換するのは難しので、簡単な例から徐々に慣れていきましょう。

ストリームの生成には複数の方法があります。コレクションから生成するには、Collection#stream メソッドを使用します。また、IntStream オブジェクトを生成するには、前述した range メソッドを使用するのが手軽です。

ストリームはメソッドを多段に連ねて、処理を行います。これをストリームのパイプラインと呼びます。

パイプラインの最後に使用できるメソッドが終端操作、それ以外を中間操作と呼びます。中間操作の戻り値は必ずストリームになります。終端操作の戻り値がストリームパイプラインの処理結果になるわけです。

今回、使用するストリームの主なメソッドは終端操作が 3 種類、中間操作が 3 種類です。

終端操作

- forEach 各要素に対して、何らかの処理を行う
- reduce 前回値を用いて、集約処理を行う
- collect 要素の集約処理を行う

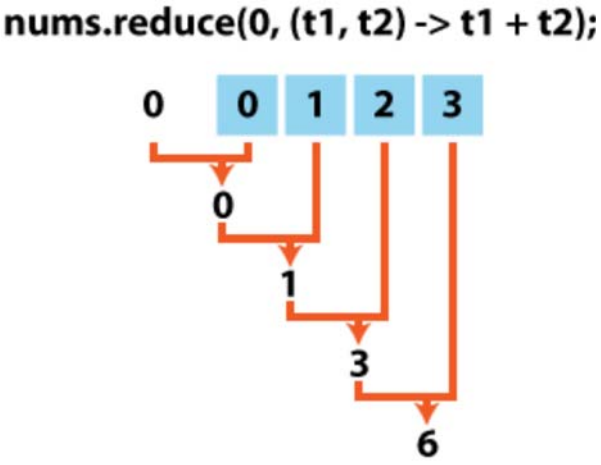
中間操作

- filter 条件に合致した要素だけをフィルタリングする
- map/mapToInt/mapToDouble/mapToLong/mapToObj 要素を、他の値に変換する
- flatMap 要素をストリームに変換し、それを連結して 1 つのストリームを生成する

この 6 種類のメソッドが使いこなせれば、ストリームは攻略できるはずです。

以下に reduce メソッド、filter メソッド、map メソッドの概念図をあげておきます。

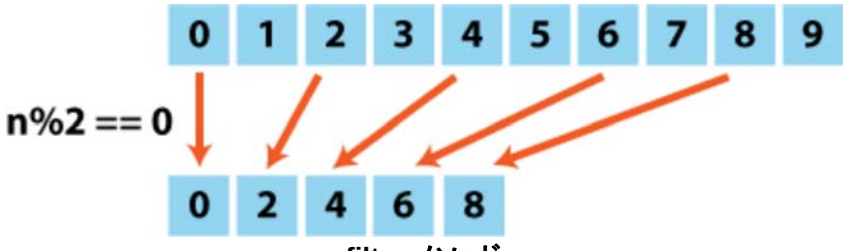
reduce メソッドは初期値 (単位元) があり、初期値と初めの要素で何らかの処理を行い、値を返します。その値が次のイテレートのラムダ式の第 1 引数として使用されます。このようにして、最後までイテレートを行い、その戻り値が最終的な結果になります。



reduce メソッド

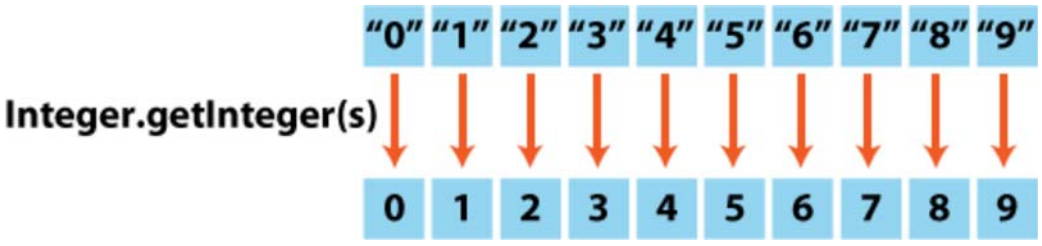
filter メソッドと map メソッドは中間操作なので、戻り値はストリームになります。

filter メソッドは条件に合致した要素だけをフィルタリングし、新たなストリームを生成するメソッドです。



filter メソッド

一方の map メソッドは各要素に対し、何らかの処理を行い、新たな値を作成することで、新たなストリームを作成します。以下の図では文字列のストリームの個々の要素を、Integer オブジェクトに変換しています。最終的に Stream<Integer> オブジェクトを生成します。



map メソッド

mapToInt メソッドなどの mapToX メソッド群は、map メソッドの一種で、Stream オブジェクトからプリミティブ型に対応した InstStream オブジェクトなど、もしくはプリミティブ型に対応したストリームから Stream オブジェクトへの変換を行うメソッドです。

また、flatMap メソッドも map メソッドの一種ですが、個々の要素をストリームに変換します。通常の map メソッドだと、個々の要素をストリームにすると、ストリームの要素にストリームという入れ子のストリームになります。これに対し、flatMap メソッドでは個々の要素を変換してできたストリームを展開し、最終的に 1 つのストリームに変換するところが map メソッドと異なります。

では、これらのメソッドを使って、for 文をストリームに書き換えて見ましょう。

4-1. Stream で書き換えてみましょう

はじめに行うのが、条件文を含む拡張 for 文です。

最終的に行う処理は標準出力への出力なので、これは forEach メソッドが使用できます。まず、すべてを forEach メソッドで記述してみましょう。

次に条件文を filter メソッドで書き換えることに挑戦してみましょう。

```
        List<Integer> numbers = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        for (Integer x: numbers) {
            if (x % 2 == 0) {
                System.out.print(x);
            }
        }
        System.out.println();
```

4-2. Stream で書き換えてみましょう

乱数から生成したリストの平均と分散を求める処理をストリームにしてみましょう。

まず、乱数をリストに追加する処理です。

この for 文ももちろんストリームで記述できます。

この処理でキーになるのはリストに追加しているのが、ループカウンタではないということです。イテレーションしている値を他の値に変えるにはどうすればよいでしょうか。

さらに、リストの生成についてもストリームの処理で行えないかどうかを考えてみましょう。

ストリームの処理の結果として値を返すには reduce メソッドか、collect メソッドを使用します。reduce メソッドはイテレートする要素の型と同じ型を戻り値とします。これに対して、collect メソッドは任意の型を戻り値にとることができます。

ということは....

collect メソッドの引数には Collectors クラスのユーティリティメソッドの戻り値を私用する場合がほとんどです。この場合、何が使用できるかチェックしてみましょう。

```
        Random random = new Random();
        List<Double> numbers = new ArrayList<>();
        for (int i = 0; i < 100; i++) {
            numbers.add(random.nextDouble());
        }
```

さて、生成したリストの平均を算出しましょう。

この場合、中間操作は必要ないので、いきなり終端操作を記述します。この場合のように、すべての要素をまとめて 1 つの値にまとめるには、reduce メソッドか collect メソッドを使用します。

ここでは、reduce メソッドを使用してみましょう。collect メソッドを使用した場合も、reduce メソッドの使い方が分からないと記述できないので、reduce メソッドをまず攻略してみましょう。

reduce メソッドで処理が記述できたら、このコードをもう少し効率よく記述できないか考えてみましょう。キーになるのはオートボクシングです。

```
        double ave = 0.0;
        for (Double x: numbers) {
            ave += x;
        }
        ave /= numbers.size();
```

最後に分散処を求める処理です。平均が記述できれば、分散も簡単に書けるはずですよ。

```
        double div = 0.0;
        for (Double x: numbers) {
            div += (x - ave) * (x -ave);
        }
        div /= numbers.size();
```

4-3. Stream で書き換えてみましょう

文字列を要素に持つリストを、個々の単語に分割して、新たにリストに保持させます。

1つの要素を分割して、それをまとめるメソッドがありましたよね。

```
        List<String> sentences = ...;

        List<String> words = new ArrayList<>();
        for (String sentence: sentences) {
            String[] splitedSentence  = sentence.split(" ");
            for (String word: splitedSentence) {
                words.add(word);
            }
        }
```

4-4. Stream で書き換えてみましょう

次に行うのはファイルを読み込んで単語数を数えるプログラムの断片です。これも Stream で書き換えてみましょう。

ヒント: BufferReader クラスに Java SE 8 で追加されたメソッドがありますよ。

```
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            int wordCount = 0;

            for (;;) {
                String line = reader.readLine();
                if (line == null) {
                    break;
                }

                String[] words = line.split(" ");
                wordCount += words.length;
            }
            System.out.println(wordCount);
        } catch (IOException ex) {
            // 例外処理
        }
```

4-5. Stream で書き換えてみましょう

単語の切り出し、ワードカウントときたら、次は単語ごとの使用回数を数えてみましょう。結果はキーが単語、値が使用回数の Map にすることにします。

途中までは単語の切り出しと同じですね。単語を切り出した後に、どうやって集計をするかが肝になります。

これはなかなか難しいですよ。

```
        List<String> sentences = ...;

        Map<String, Integer> result = new HashMap<>();

        for (String sentence: sentences) {
            String[] splitedSentence  = sentence.split(" ");

            for (String word: splitedSentence) {
                String lowerWord = word.toLowerCase();
                int count = result.getOrDefault(lowerWord, 0);
                result.put(lowerWord, count+1);
            }
        }
```

いかがでしたでしょうか。

ちなみに、個々の問題は正解が 1 つとは限りません。いろいろな書き方があり、それに応じて使用するメソッドも変化します。いろいろな書き方をできるようにしておくと、応用範囲が広がると思います。