Java 8 のイディオム: 関数合成およびコレクション・パイプライン・パターン

Java でコレクションを繰り返し処理するための関数型のパターン

Venkat Subramaniam
Founder
Agile Developer, Inc.

2017年 7月 06日

コード内でコレクションを繰り返し処理する方法は、無限ループだけではありません。関数合成とコレクション・パイプラインは、ステートメントではなく式を使ってコレクションをソートできる、Java での 2 つのパターンです。

このシリーズの他の記事を見る

関数型スタイルのプログラミングを導入するようになると、作成するプログラムに特定の関数型設計パターンが現れてきます。これらのパターンが自然に現れてくるとしても、実際に使いこなせるようになるための努力は必要です。この記事では、関数型スタイルのパターンのうち、関数合成とコレクション・パイプラインという 2 つのパターンを取り上げます。この 2 つを組み合わせることで、コード内でコレクションを繰り返し処理できます。この 2 つのパターンの仕組みを理解しておくと、高階関数とラムダ式を最大限利用するように Java プログラムを構成する上で役立ちます。

このシリーズについて
Java 始まって以来の最も大々的な更新となっている Java 8 には、どこから手を付けてよいのか戸惑ってしまうほど新しい機能が満載されています。このシリーズでは、教育者である著者の Venkat Subramaniam がイディオムを考慮した Java 8 手法を簡潔に紹介し、当たり前のように思ってきた Java の慣例を見直して、プログラムに新しい手法と構文を徐々に取り込んでいくよう読者を導きます。

ステートメントと式

コード・ベース内で `for` を grep 検索するだけで、自分がコード内でいかに頻繁に `for` ループを使っているかを知って驚くことでしょう。私はこれを「for ハンマー (`for` hammer)」と呼んでいます。つまり、ループ処理が必要なときは、いつでもその手段として `for` の一撃に頼りがちであるという意味です。

Java で使用する for とwhile は、どちらもステートメントです。ステートメントはアクションを実行しますが、結果は生成しません。本質的に、何らかの有用なアクションを実行するステートメ

ントによって導かれる結果は、データの状態変化 (ミューテーション) です。ステートメントはその影響を、データの状態変化という形でしか伝えることができません。それとは対照的に、式はデータの状態を変化させることなく、結果を生成することができます。

コード内でステートメントを使用するということは、例えて言うと、チームとして共同作業を行っていながらも、チームのメンバー間で直接成果物を受け渡すことができない状態を表します。成果物を他のメンバーと共有するには、成果物をテーブルや棚に置いて、そのメンバーが引き取れるようにするしか方法はありません。一方、式は連鎖した形で機能します。あるメンバーがタスクを完了すると、その成果物を次のメンバーに渡し、そのメンバーがタスクを完了すると、その成果物をまた次のメンバーに渡していくといった形です。

コレクション・パイプライン・パターンは、式を使用することで形成しやすくなります。コレクション・パイプライン・パターンとは、Martin Fowler 氏が、ある処理で収集された出力が次の処理に入力される処理のシーケンスとして説明しているパターンです。コレクション・パイプラインはオブジェクト指向プログラミングで使用されるパターンですが (おそらく、オブジェクト・ビルダーを使用するコード内でこのパターンを目にしたことがあるはずです)、関数型プログラミングでのほうが目立っています。

関数合成パターンとコレクション・パイプライン・パターンの 2 つは連携して機能します。次のセクションでは、お馴染みの `for` ステートメントを使用して問題を解決します。その上で、関数合成とコレクション・パイプラインという 2 つの連動するパターンを使用して、同じ問題をより効率的に解決する方法を説明します。

ステートメントを使用した繰り返し処理とソート

まず始めに、メーカー、モデル、製造年をプロパティーとして持つ、以下の `Car` クラスを見てください。

リスト 1. Car クラス

```
package agiledeveloper;

public class Car {
  private String make;
  private String model;
  private int year;

  public Car(String theMake, String theModel, int yearOfMake) {
    make = theMake;
    model = theModel;
    year = yearOfMake;
  }

  public String getMake() { return make; }
  public String getModel() { return model; }
  public int getYear() { return year; }
}
```

`Car` インスタンスのコレクションは、以下のようにして追加できます。

リスト 2. Car インスタンスのコレクション

```
package agiledeveloper;
```

```
import java.util.Collections;
import java.util.Comparator;
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;

public class Iterating {
  public static List<Car> createCars() {
    return Arrays.asList(
      new Car("Jeep", "Wrangler", 2011),
      new Car("Jeep", "Comanche", 1990),
      new Car("Dodge", "Avenger", 2010),
      new Car("Buick", "Cascada", 2016),
      new Car("Ford", "Focus", 2012),
      new Car("Chevrolet", "Geo Metro", 1992)
    );
  }
```

リスト 3 では、命令型スタイルのプログラミングに従って、リストを繰り返し処理し、製造年が 2000 年より後の自動車の名前を取得しています。名前を取得した後は、製造年を基準に昇順でモデルをソートします。

リスト 3. 「for」を使用して製造年でソートされるモデル

```
public static List<String> getModelsAfter2000UsingFor(List<Car> cars) {
    List<Car> carsSortedByYear = new ArrayList<>();

    for(Car car : cars) {
      if(car.getYear() > 2000) {
        carsSortedByYear.add(car);
      }
    }

    Collections.sort(carsSortedByYear, new Comparator<Car>() {
      public int compare(Car car1, Car car2) {
        return new Integer(car1.getYear()).compareTo(car2.getYear());
      }
    });

    List<String> models = new ArrayList<>();
    for(Car car : carsSortedByYear) {
      models.add(car.getModel());
    }

    return models;
  }
```

ご覧のように、このコードでは大量のループ処理が行われます。まず、`getModelsAfter2000UsingFor` メソッドがそのパラメーターとして自動車のリストを取ります。次に、製造年が 2000 年より後の自動車を抽出またはフィルタリングして、それらの自動車を `carsSortedByYear` という名前の新しいリストに挿入します。続いて、このリストを、製造年を基準に昇順でソートします。そして最後に `carsSortedByYear` リストを繰り返し処理してモデル名を取得し、それらの名前をリストに挿入して返します。

これよりも効率的なコードにするにはどうすればよいでしょうか。例えば、すべての自動車を同時にソートすれば、`for` ループのうちの 1 つを排除することができます。けれども、その場合、ソートするオブジェクト・リストのサイズが大きくなってしまいます。

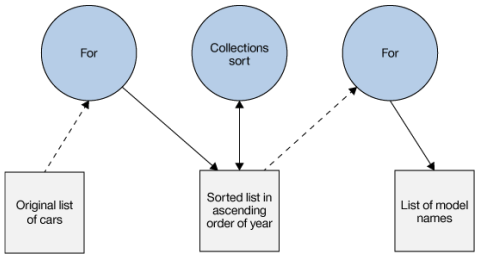
以下に、自動車リストの出力例を示します。

リスト 4. getModelsAfter2000UsingFor の出力

```
Avenger, Wrangler, Focus, Cascada
```

この簡単な例は、私が「ステートメント・エフェクト (effect of statements)」と呼んでいる状況を具体化するものです。概して関数とメソッドは式として使用できますが、`Collectionssort` メソッドは結果を返しません。それは、このメソッドがステートメントとして使用されているためです。したがって、引数として渡されるリストの状態を変化させるだけで、結果を返すことはしません。2 つの `for` ループも、繰り返し処理によってリストの状態を変化させます。これらの要素はステートメントとしてしか機能しないため、このコードには不要なガーベッジ変数が含まれる結果となっています (図 1 を参照)。

図 1. ステートメント・エフェクト



コレクション・パイプラインを使用した繰り返し処理とソート

関数型プログラミングでは、一連の小さなモジュール関数または処理を順序付けて複雑な処理を組み立てるという方法が一般的な慣例となっています。この順序付けは、関数の合成、または関数合成と呼ばれています。データのコレクションが関数合成を通過することで、コレクション・パイプラインになります。関数合成とコレクション・パイプラインの 2 つは、関数型スタイルのプログラミングで頻繁に用いられている設計パターンです。

関数合成とコレクション・パイプラインを利用することで、サイズの大きい 1 つの `for` ハンマーに頼るのではなく、目前の問題に応じて複数の特化されたツールを使用できるようになります。命令型スタイルのプログラミングでは、あらゆるものにステートメントを使用するのが一般的ですが、関数型スタイルのプログラミングでは式の使用が促されます。式には、ステートメントのようにオブジェクトの状態変化といった副次作用がありません。`filter` や `map` のような式も結果を返すため、その結果を別の関数に渡すことができます。コレクション・パイプラインは、このようにして作成します。一例として、リスト 5 のコードを見てください。

リスト 5. コレクション・パイプライン・パターンでの関数合成

```
public static List<String> getModelsAfter2000UsingPipeline(
   List<Car> cars) {
   return
     cars.stream()
         .filter(car -> car.getYear() > 2000)
         .sorted(Comparator.comparing(Car::getYear))
         .map(Car::getModel)
         .collect(toList());
 }
```

上記の `getModelsAfter2000UsingPipeline` メソッドは、リスト 3 の `getModelsAfter2000UsingFor` メソッドと同じ結果をもたらしますが、このコードには以下の違いがあることに注目してください。

- 関数型スタイルのコードは、命令型スタイルのコードよりも簡潔です。
- 関数型スタイルのコードは明示的に状態を変化させることはなく、使用するガーベッジ変数は命令型スタイルのコードよりも少なくなります。
- 2 番目のメソッドで使用されている関数/メソッドは、いずれも値を返す式です。この点を、リスト 3 に示されている Collections.sort メソッドと見比べてみてください。
- コレクション・パイプライン・パターンを使用した getModelsAfter2000UsingPipeline は、非常にわかりやすく表現されています。

このコードの目的は、ほんの数行で明らかにされています。自動車のコレクションから、製造年が 2000 年より後の自動車だけをフィルタリングまたは抽出すること、製造年を基準にソートすること、オブジェクトをモデル名にマッピングまたは変換すること、そして結果をリストに収集することが、それぞれの行でわかります。

リスト 5 のコードが簡潔で表現力豊かなものになっている背景のいくぶんかは、メソッド参照を使用していることにあります。ラムダ式を `filter` に渡すのは道理にかなっています。この場合、ラムダ式で、特定のオブジェクトから製造年のプロパティーを取得して、その値を 2000 と比較するという処理に対応できるためです。ただし、ラムダ式を `map` に渡してもそれほど有効ではありません。`map` メソッドに渡される式は `car -> car.getModel()` で、この式は比較的単純な式であるためです。ラムダ式は特定のオブジェクトのプロパティーを返すだけで、実際の計算や処理は行いません。したがって、メソッド参照で置き換えるのが賢明です。

`map` メソッドにはラムダ式を渡すのではなく、メソッド参照 `Car::getModel` を渡しています。同様に、比較メソッドにも、ラムダ式 `car -> car.getYear()` を渡すのではなく、メソッド参照 `Car::getYear` を渡しています。メソッド参照は短く簡潔で、表現力豊かです。可能な限り、メソッド参照を使用することが推奨されます。

リスト 5 のコレクション・パイプラインは、図 2 のようになります。

図 2. コレクション・パイプラインの魅力

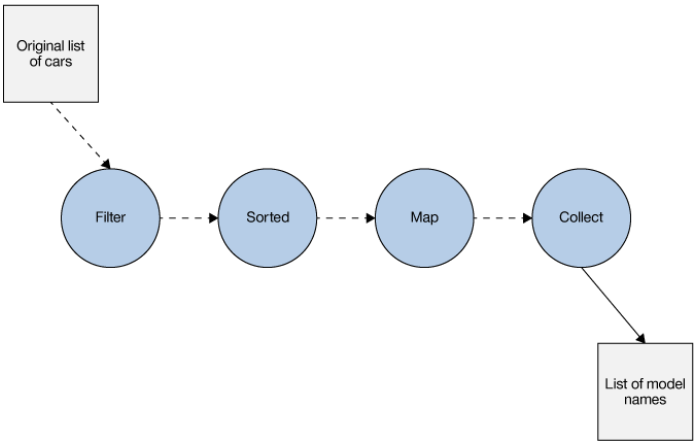


図 2 を見ると、`getModelsAfter2000UsingPipeline` 関数がコレクション・パイプラインを実行して、渡された入力を一連の関数によって処理し、形を変えていく仕組みがわかります。場合によっては、データに各関数が適用される間、Java 8 の遅延評価と関数結合機能 (このリンク先のページで紹介されている『Javaによる関数型プログラミング』(2014年) を参照) によって、中間オブジェクトが生成されないようにすることもできます。関数自体も、パイプラインによるデータ遷移中に、中間オブジェクトを可視または使用可能にすることはありません。

まとめ

命令型スタイルのプログラミングでは、ほとんどの類のデータ処理に `for` および `while` ループを使用するのが一般的です。この記事では、その代わりとなる手法として、関数型スタイルのプログラミングで非常によく使われている関数合成とコレクション・パイプラインによる手法を説明しました。関数合成は、一連のモジュール関数を順序付けて複雑な処理を作成するという単純な

手法です。この順序付けられた一連の関数をデータに適用することで、コレクション・パイプラインになります。関数合成とコレクション・パイプラインという 2 つのパターンにより、アップストリームからダウンストリームへと向かうフローの中で、データの状態を徐々に変えていくという高度なプログラムを作成することが可能になります。

関連トピック： Martin Fowler 氏によるコレクション・パイプライン・パターン Java 8 でのラムダ式およびメソッド参照 関数型のデザイン・パターン、第 1 回 - 関数型の世界でのパターンの現れ方 Javaによる関数型プログラミング (オライリージャパン、2014年)