Java 8 のイディオム: パススルーに代わる手段

Java コード内でパススルー・ラムダ式を認識し、メソッド参照で置き換える方法を学ぶ

Venkat Subramaniam
Founder
Agile Developer, Inc.

2017年 10月 09日

パススルー・ラムダ式のバリエーションの数は限られていますが、ほぼ常に、これらのラムダ式が引き起こす面倒はラムダ式の価値を打ち消してしまいます。一般的なパススルーのバリエーションをコード内で識別する方法を学び、そのそれぞれを単純で表現豊かなメソッド参照で置き換えた場合の結果を確かめてください。

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。

関数型スタイルのプログラミングでは、ラムダ式が広範囲にわたって使用されますが、ラムダ式を読んで理解するのは難しい場合があります。ラムダ式が 1 つ以上のパラメーターを渡すためだけに使われているとしたら、大抵はメソッド参照で置き換えたほうが理解しやすいコードになります。この記事で、コード内で使用されているパススルー・ラムダ式を認識する方法、そしてそのようなラムダ式を同等のメソッド参照で置き換える方法を学んでください。メソッド参照を使用するには学習曲線がありますが、すぐに長期的な利益がもたらされて、最初の努力が報われるはずです。

パススルー・ラムダ式とは何か？

関数型スタイルのプログラミングでは、ラムダ式を匿名関数として渡し、ラムダを高階関数の引数として使用するのが一般的な方法となっています。例えばリスト 1 では、`filter` メソッドにラムダ式を渡しています。

リスト 1. パススルー・ラムダ式

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

numbers.stream()
  .filter(e -> e % 2 == 0)
  .forEach(e -> System.out.println(e));
```

上記のコードでは、ラムダ式を `forEach` メソッドにも渡しています。これらのラムダ式の目的は明らかに異なりますが、この 2 つのラムダ式の間に重要な、しかも捉えにくい別の違いがあるとしたら、それは最初のラムダ式は実際に何らかの処理を行う一方、2 番目のラムダ式はそうではないという点です。

`forEach` メソッドに渡されるラムダ式が、パススルー・ラムダ式と呼ばれるものです。式 `e -> System.out.println(e)` はそのパラメーターを、`PrintStream` クラスの `println` メソッド (`System.out` のインスタンス) に引数として渡します。

リスト 1 の 2 番目のラムダ式には何も問題はありませんが、このタスクで必要とされない複雑な構文になっています。「(パラメーター) -> 本体」の目的を理解するためには、本体 (`->` の右辺) にステップインし、そのパラメーターによってどのような処理が行われるのかを調べなければなりません。その作業は、ラムダ式が渡すパラメーターによって実際の処理が行われないとしたら無駄なものになります。

パラメーターで実際の作業を行わないパススルー・ラムダ式は、メソッド参照で置き換えるほうが有益です。メソッドの呼び出しとは異なり、メソッド参照はどのメソッドにパラメーターを渡すかをメソッド名で指定します。メソッド参照を使用すると、パラメーターを渡す方法にさまざまな選択肢が開けることにもなります。

前のコードを以下のように作成し直してみてください。

リスト 2. メソッド参照を使用してパラメーターを渡す

```
numbers.stream()
  .filter(e -> e % 2 == 0)
  .forEach(System.out::println);
```

このように、メソッド参照を使用すると、理解しやすいコードになります。これは、最初は取るに足らないメリットに思えるかもしれませんが、作成して理解するコードの量が増えてくると、そのメリットは倍増することになります。

パラメーターを引数として渡す

以降のセクションでは、パススルー・ラムダ式のバリエーションを探り、それぞれのパススルー・ラムダ式をメソッド参照で置き換える方法を説明します。

インスタンス・メソッドへの引数

ラムダ式は、パラメーターをインスタンス・メソッドに引数として渡すためにかなり一般的に使われています。リスト 1 では、パラメーター `e` が引数として `println` メソッドに渡されました。このメソッドそのものは、`System.out` のインスタンス・メソッドです。

リスト 2 ではリスト 1 のラムダ式を、referenceToInstance::methodName という形式のメソッド参照 `System.out::println` で置き換えました。

図 1 に示されているラムダ式の構造を見るとわかるように、パラメーターはインスタンス・メソッドに引数として渡されます。

図 1. パラメーターからインスタンス・メソッドの引数へ

reference::instanceMethod

メソッド参照に馴染みがないとしたら、上記のようなラムダ式を見ることで、その構造と、パラメーターがどこに渡されるのか理解するのに役立つはずです。ラムダ式をメソッド参照に変更するには、共通の部分、パラメーター、および引数を削除し、メソッド呼び出しでドットをコロンで置き換えればよいだけです。

`this` 上のメソッドの引数

メソッド参照で置き換える前のパススルーの特殊な例は、インスタンス・メソッドが現在のメソッドのコンテキスト・インスタンスに対して呼び出される場合です。

例えば、`Example` という名前のクラスに `increment` のインスタンス・メソッドがあるとします。

リスト 3. インスタンス・メソッドを持つクラス

```
public class Example {
  public int increment(int number) {
    return number + 1;
  }

  //...
}
```

このクラスにはさらに別のインスタンス・メソッドがあります。そのインスタンス・メソッドでは以下のように、ラムダ式を作成して、それを `Stream` の `map` メソッドに渡しています。

リスト 4. パラメーターをインスタンス・メソッドに渡すラムダ式

```
.map(e -> increment(e))
```

すぐにはわからないかもしれませんが、このコードは構造の点で前の例と非常によく似ています。つまり、どちらの例でも、パタメーターをインスタンス・メソッドに引数として渡しています。このコードを作成し直すと、この類似点がもう少しわかりやすくなります。

リスト 5. 明らかになったパススルー

```
.map(e -> this.increment(e))
```

`increment` を呼び出すターゲットとして冗長な `this` を導入すると、パススルーの構造が明らかになります。これがわかれば、メソッド参照によって、この冗長性を簡単に解消できます。

リスト 6. メソッド参照による冗長性の解消

```
.map(this::increment)
```

`e -> System.out.println(e)` を `System.out::println` で置き換えたのと同じように、ラムダ式 `e -> increment(e)` (正確には、`e -> this.increment(e)`) は `this::increment` で置き換えることができます。いずれの場合も、コードがより理解しやすくなります。

静的メソッドの引数

これまでに紹介した 2 つの例では、パラメーターをインスタンス・メソッドに引数として渡すラムダ式を置き換えました。パラメーターを静的メソッドに渡すラムダ式も置き換えることができます。

リスト 7. パラメーターを静的メソッドに渡す

```
.map(e -> Integer.valueOf(e))
```

上記のラムダ式は、`Integer` クラスの `valueOf` メソッドに、引数としてパラメーターを渡しています。このコードの構造は図 1 に示した構造と同じですが、唯一の違いとして、上記の例で呼び出されるメソッドはインスタンス・メソッドではなく静的メソッドです。前の 2 つの例と同じように、このラムダ式をメソッド参照で置き換えるには、メソッド参照の基準をインスタンスに置くのではなく、クラスに置きます (リスト 8 を参照)。

リスト 8. 静的メソッドへのメソッド参照

```
.map(Integer::valueOf)
```

要約すると、ラムダ式の目的がパラメーターをインスタンス・メソッドに渡すことだけであれば、そのラムダ式をインスタンスに基づくメソッド参照で置き換えることができます。パラメーターを静的メソッドに渡すのであれば、ラムダ式をクラスに基づくメソッド参照で置き換えることができます。

パラメーターをターゲットに渡す

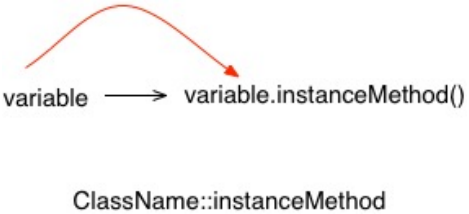
ClassName::methodName の形式を使用できるシナリオには 2 つの種類があります。一方のシナリオは、これまで見てきたように、パラメーターを静的メソッドに引数として渡すというものです。もう一方のシナリオとして、パラメーターがメソッド呼び出しのターゲットである場合のバリエーションを見ていきましょう。

リスト 9. パラメーターをターゲットとして使用する

```
.map(e -> e.doubleValue())
```

上記の例で、パタメーター `e` は `doubleValue` メソッドのターゲットです。このパラメーターに推測される型は、`Integer` であると前提します。表 2 に、このタイプのパススルー・ラムダ式の構造を示します。

図 2. パラメーターからターゲットへ



以上の 2 つのラムダ式に含まれるパラメーターはそれぞれ異なる方法で渡されるとは言え (一方は静的メソッドの引数として、もう一方はインスタンス・メソッド呼び出しのターゲットとして渡されます)、メソッド参照の形式はまったく変わらず、ClassName::methodName となります。

曖昧さとメソッド参照

メソッド参照を調べても、パラメーターが静的メソッドに渡されるのか、あるいはターゲットとして使用されているのかを簡単に判断することはできません。この違いを把握するには、該当するメソッドが静的メソッドであるか、インスタンス・メソッドであるかを知る必要があります。コードの読みやすさという点ではそれほど大きな問題ではありませんが、コードが正常にコンパイルされるようにするためには、この違いを把握することが不可欠です。

クラスに静的メソッドと、同じ名前の両立するインスタンス・メソッドがある場合、メソッド参照を使用すると、コンパイラーは呼び出しが曖昧であることからエラーを返します。したがって、例えば `Integer` クラスには静的メソッド `public static String toString(int i)` とインスタンス・メソッド `public String toString()` の両方があるため、ラムダ式 `(Integer e) -> e.toString()` をメソッド参照 `Integer::toString` で置き換えることはできません。

この特定の問題の場合、`Object` には `statictoString` がないため、開発者または IDE が `Object::toString` を使用するよう提案することで解決できます。この解決法によってコンパイルは成功するとしても、一般に、このような知恵は助けになりません。それよりも、メソッド参照が目的のメソッドを呼び出すことを確認するスキルが必要です。疑わしいときは、混乱やエラーを回避するためにラムダ式を使用するのが最善の策となります。

コンストラクター呼び出しを渡す

メソッド参照は、静的メソッドとインスタンス・メソッドを表すためだけでなく、コンストラクターの呼び出しを表すためにも使用できます。以下に示す、`Supplier` 内から `toCollection` に対してコンストラクターを呼び出す例を見てください。

リスト 10. コンストラクター呼び出し

```
.collect(toCollection(() -> new LinkedList<Double>()));
```

リスト 10 のコードは、データの `Stream` を取って、それを `LinkedList` 内に集約または収集することを目的としています。`toCollection` メソッドは引数として `Supplier` を取ります。Supplier が取るパラメーターはないため、空の `()` になっています。この呼び出しによって `Collection` のインスタンス (この例の場合は `LinkedList`) が返されます。

パラメーター・リストは空ですが、一般的なコードの構造は以下のようになります。

図 3. パラメーターからコンストラクターの引数へ

ClassName::new

受け取ったパラメーター (空の場合もあります) が、コンストラクターの引数として渡されます。この場合、ラムダ式を `new` へのメソッド参照で置き換えることができます。その方法は以下のとおりです。

リスト 11. コンストラクター呼び出しをメソッド参照で置き換える

```
.collect(toCollection(LinkedList::new));
```

メソッド参照を使用したコードのほうが、ラムダ式を使った元のコードよりも大幅にノイズが少なくなるため、読んで理解するのが簡単になります。

複数の引数を渡す

これまでのところ、単一のパラメーターや空のパラメーターを渡す例を見てきました。けれども、メソッド参照はゼロまたは 1 つのパラメーターでしか機能しないわけではありません。複数の引数を使用する場合でも有効に機能します。以下の `reduce` 処理を見てください。

リスト 12. ラムダ式を使用した reduce()

```
.reduce(0, (total, e) -> Integer.sum(total, e)));
```

`reduce` メソッドは `Stream<Integer>` に対して呼び出され、`Integer` の `sum` メソッドを使ってストリームに含まれる値を合計します。この例でのラムダ式は 2 つのパラメーターを取り、それらのパラメーターを (まったく同じ順序で) `sum` メソッドに引数として渡します。図 4 に、このラムダ式の構造を示します。

図 4. 2 つのパラメーターを引数として渡す

ClassName::methodName

このラムダ式は、以下のようにメソッド参照で置き換えることができます。

リスト 13. 2 つのパラメーターを取るラムダ式を置き換える

```
.reduce(0, Integer::sum));
```

`static` メソッド呼び出しがその引数として、ラムダ式に指定された複数のパラメーターをパラメーター・リストに示されている通りの順序で取るとしたら、`static` メソッドを指すメソッド参照でラムダ式を置き換えることができます。

ターゲットおよび引数としてのパススルー

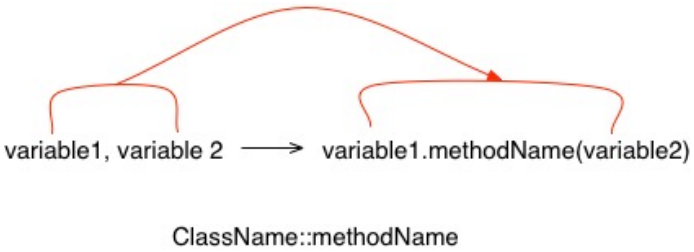
すべてのパラメーターを引数として `static` メソッドに渡すのではなく、ラムダ式ではパラメーターのうちの 1 つをインスタンス・メソッド呼び出しのターゲットとして使用できる場合もあります。最初のパラメーターがターゲットとして使用される場合、ラムダ式をメソッド参照で置き換えることができます。リスト 14 の例を見てください。

リスト 14. パラメーターをターゲットとして使用するラムダ式を使った reduce()

```
.reduce("", (result, letter) -> result.concat(letter)));
```

この例では、`reduce` メソッドが `Stream<String>` に対して呼び出されます。ラムダ式は `String` の `concat` インスタンス・メソッドを使用してストリングを連結します。このラムダ式内でのパススルーの構造は、前の `reduce` の例で見た構造とは異なります。

図 5. 呼び出しのターゲットとして渡される最初のパラメーター



ラムダ式の最初のパラメーターが、インスタンス・メソッド呼び出しのターゲットとして使用され、2 番目のパラメーターが、そのメソッドの引数として使用されます。この順序を前提として、リスト 15 に示すようにラムダ式をメソッド参照で置き換えることができます。

リスト 15. 最初のパラメーターをターゲットとして使用するメソッド参照

```
.reduce("", String::concat));
```

ラムダ式がインスタンス・メソッドを呼び出すとしても、同じくクラス名を使用しなければならないことに注意してください。つまり、静的メソッドを呼び出す場合でも、最初のパラメーターをターゲットとして使用してインスタンス・メソッドを呼び出す場合でも、メソッド参照は同じ形になりますが、曖昧さがない限り、問題はありません。

メソッド参照を使用するメリット

パススルー・ラムダ式のバリエーションと構造、そしてそれらのラムダ式を置き換えるメソッド参照を理解するには、ある程度の時間と努力が必要です。私の場合、この概念と構文を十分に理解するまでに数週間かかりました。けれどもその後は、パススルーではなくメソッド参照を使用するほうがより自然に感じられるようになりました。

ラムダ式はコードを簡潔にしますが、メソッド参照はそれ以上にコードを簡潔かつ表現力豊かにするため、コードを理解するのに必要な努力が大幅に減ります。この点を強調するために、最後の例を見てください。

リスト 16. ラムダ式を使用した例

```
List<String> nonNullNamesInUpperCase =
    names.stream()
      .filter(name -> Objects.nonNull(name))
      .map(name -> name.toUpperCase())
      .collect(collectingAndThen(toList(), list -> Collections.unmodifiableList(list)));
```

`List<String> names` を基に、上記のコードはこのリストからすべての `null` 値を削除し、残った名前のそれぞれを大文字に変換して、その結果を変更不可能なリスト内に収集します。

ここで、メソッド参照を使用して上記のコードを作成し直しましょう。この例の場合、それぞれのラムダ式は静的メソッドまたはインスタンス・メソッドにパラメーターを渡すパススルー・ラムダ式です。したがって、各ラムダ式をメソッド参照で置き換えます。

リスト 17. メソッド参照を使用した例

```
List<String> nonNullNamesInUpperCase =
    names.stream()
      .filter(Objects::nonNull)
      .map(String::toUpperCase)
      .collect(collectingAndThen(toList(), Collections::unmodifiableList));
```

2 つのリストを見比べると、メソッド参照を使用したコードのほうがより流ちょうで理解しやすいことが一目瞭然です。このコードからは、指定された名前のうち、null ではない名前をフィルタリングし、大文字にマッピングし、変更不可能なリストに収集するという内容をすぐに理解できます。

まとめ

パラメーターを 1 つ以上の他の関数に渡すことだけを目的とするラムダ式を見つけたときは必ず、そのラムダ式をメソッド参照で置き換えたほうがよいかどうかを考えてください。メソッド参照で置き換える決め手となるのは、ラムダ式の内部で実際の処理が行われていないことです。その場合、ラムダ式はパラメーターを渡すだけのパススルー・ラムダ式であるため、そのタスクには、ラムダ式の構文はおそらく不必要に複雑でしょう。

ほとんどの開発者にとって、メソッド参照を使用する方法を習得するには多少の努力が必要になります。けれどもいったん理解してしまえば、メソッド参照を使用したコードのほうが、ラムダ式を使用した同じコードに比べ、流ちょうで表現豊かなコードになることがわかるはずです。

関連トピック： Java 8 Method Reference: How to Use it Java programming with lambda expressions Java 8 言語での変更内容 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)