Java.next: Java.next としての Java 8

これまでの Java を置き換えるのにふさわしい言語として Java 8 を評価する

Neal Ford

2014年 7月 10日

連載「Java.next」の今回の記事では、皆さんが次に使用するプログラミング言語の有力な候補として Java 8 リリースを取り上げます。ラムダ式ブロックと Stream API がどのような形で Java を最新の言語にアップグレードするかを理解してください。

このシリーズの他の記事を見る

この連載について

Java の遺産となるのは、プラットフォームであって、言語ではないでしょう。200 を超える言語が JVM 上で実行されている今、最終的にこれらの言語の 1 つが JVM のプログラミングに最適な方法として Java 言語に取って代わることは避けられません。この連載では、Java 開発者が自分たちの近い将来を垣間見ることができるように、3 つの次世代 JVM 言語 — Groovy、Scala、Clojure — について、新しい機能やパラダイムを比較対照することで、詳しく探ります。

この連載の元々の使命は、3 つの新しい JVM 言語を比較対照して、どの言語が Java 言語の後継になり得るかを読者の皆さんが評価できるようにすることでした。ところが連載開始から現在に至るまでに、Java 言語は Generics を追加して以来、最も著しい変化を遂げました。現在は Java 自体が、Groovy、Scala、および Clojure が持つ望ましい特徴の多くを見せるようになっています。今回の記事では、Java 8 を Java.next 言語とみなし、この言語のプログラミング・パラダイムがいかに効果的に強化されているかを示す例を記載します。

ついに導入された高階関数

高階関数とは、引数として他の関数を取るか、または他の関数を結果として返す関数のことです。Java は (よく使われる言語のなかではおそらく最後まで高階関数の導入に抵抗していた言語ですが)、ラムダ式ブロックという形でついに高階関数を導入しました。Java 8 のエンジニアたちは高階関数をただ単に言語に追加するのではなく、それよりも賢い選択として、古いインターフェースで関数型の機能を利用できるようにしました。

私は「Java.next: 関数型のコーディング・スタイル」の記事で、従来であれば命令型で解決していたような問題を Java.next 言語で処理する場合の実装例を説明しました。例として取り上げた問題は、名前の入力リストから 1 文字だけの名前を除外して、それぞれの名前を先頭が大文字で始ま

るようにし、コンマで区切った 1 つのリストにして返すというものです。リスト 1 に、命令型の Java によるソリューションを示します。

リスト 1. 命令型の名前変換

```
public String cleanNames(List<String> listOfNames) {
    StringBuilder result = new StringBuilder();
    for(int i = 0; i < listOfNames.size(); i++) {
        if (listOfNames.get(i).length() > 1) {
            result.append(capitalizeString(listOfNames.get(i))).append(",");
        }
    }
    return result.substring(0, result.length() - 1).toString();
}

public String capitalizeString(String s) {
    return s.substring(0, 1).toUpperCase() + s.substring(1, s.length());
}
```

以前のバージョンの Java を使用した場合、繰り返し処理を行うのが標準的ですが、Java 8 では、Stream を使用してこのジョブを簡潔に処理します。Stream とは、コレクションと UNIX のパイプを合わせたような動作をする抽象化です。リスト 2 では、Stream を使用しています。

リスト 2. Java 8 での名前変換

```
public String cleanNames(List<String> names) {
    return names
            .stream()
            .filter(name -> name.length() > 1)
            .map(name -> capitalize(name))
            .collect(Collectors.joining(","));
}

private String capitalize(String e) {
    return e.substring(0, 1).toUpperCase() + e.substring(1, e.length());
}
```

リスト 1 の繰り返し処理を行うバージョンでは、フィルタリング、変換、連結のタスクごとにコレクションをループ処理するのは効率的ではないため、これらのタスクをすべて 1 つの `for` ループの中で処理する必要があります。Java 8 の Stream を使用すれば、出力を生成する関数を呼び出す (「終了処理」と呼ばれます) までの関数 (`collect()` や `forEach()` など) の呼び出しをチェーニングして合成することができます。

> リスト 2 では、ラムダ式ブロックを作成するために構文糖を少し使用しています。具体的には、`filter(name -> name.length() > 1)` は、`filter((name) -> name.length() > 1)` の省略形です。引数が 1 つの場合、必要以上の括弧の対はオプションとなります。

リスト 2 の `filter()` メソッドは、関数型言語でよく使われている filter メソッドと同様のメソッドです (「Java.next: シノニムのややこしさを克服する」を参照してください)。この `filter()` メソッドが引数に取る高階関数は、フィルタリング基準として使用するブール値を返します。このブール値が `true` の場合はフィルタリングされたコレクションに含まれることを意味し、`false` の場合はそのコレクションに含まれないことを意味します。

`filter()` メソッドは、ブール値を返すメソッド `Predicate<T>` を型として持つ引数を取ります。述部インスタンスは、リスト 3 に示すように、必要に応じて明示的に作成することができます。

リスト 3. ハンド・コーディングによる述部の作成

```
Predicate<String> p = (name) -> name.startsWith("Mr");
List<String> l = List.of("Mr Rogers", "Ms Robinson", "Mr Ed");
l.stream().filter(p).forEach(i -> System.out.println(i));
```

リスト 3 で作成している述部には、フィルタリング・ラムダ式ブロックを使用しています。この述部は、3 行目で `filter()` メソッドを呼び出すときに、要求される引数として渡されます。

リスト 2 の `map()` メソッドは期待通りの動作をし、コレクションに含まれる各要素に対して `capitalize()` メソッドを適用します。最後に終了処理として、Stream から値を生成する `collect()` メソッドを呼び出します。`collect()` メソッドはお馴染みの reduce 処理を実行します。つまり、要素を結合し、(通常は) より小さいサイズの結果を生成します。場合によっては、結果が単一の値になることもあります (例えば、「合計」演算の場合)。Java 8 にも `reduce()` メソッドはありますが、この例では、`StringBuilder` などの可変コンテナーと効率的に連動するという理由から collect() を使用したほうが望ましいです。

既存のクラスやコレクションに map や reduce などの関数型の構成体を認識させるようにすることによって、Java はコレクションの効率的な更新という問題に直面します。例えば、`ArrayList` といった典型的な Java コレクションで使用できなければ、reduce 処理の有用性は大幅に下がります。Scala と Clojure のコレクション・ライブラリーの多くはデフォルトで不変であるため、ランタイムは効率的な処理を生成することができます。一方、Java 8 は開発者にコレクションを変更するように強制することはできず、Java の既存のコレクション・クラスの大多数は可変です。そのことから、Java 8 には、`ArrayList` や `StringBuilder` などのコレクションに対して可変の reduce 処理を実行するメソッドが組み込まれています。これらのメソッドは、結果を毎回置き換えるのではなく、既存の要素を更新します。リスト 2 では `reduce()` を使用することもできますが、このインスタンスで返されるコレクションには、`collect()` のほうが効率的に機能します。

記事「Java.next: 並行性を比較対照する」で説明している関数型言語の利点の 1 つは、関数型言語の場合、通常は修飾子を 1 つ追加するだけで、簡単にコレクションを並列化できることです。これと同じ利点が、Java 8 にも備わっています (リスト 4 を参照)。

リスト 4. Java 8 での並列処理

```
public String cleanNamesP(List<String> names) {
    return names
            .parallelStream()
            .filter(n -> n.length() > 1)
            .map(e -> capitalize(e))
            .collect(Collectors.joining(","));
}
```

Scala での場合と同様に、リスト 4 では `parallelStream()` 修飾子を追加することによって、Stream の処理が並列に実行されるようにしています。関数型プログラミングでは実装の詳細をランタイムに任せるため、開発者はより上位の抽象化レベルで作業することができます。スレッド化を簡単にコレクションに適用できることが、この利点を実証しています。

Java 8 のリデューサーのタイプ間にある違いは、関数型プログラミングのような奥の深いパラダイムを既存の言語構成体に追加する難しさを表しています。Java 8 チームは、ほとんどシームレスに

関数型の構成体を追加するという偉業を成し遂げました。そのような統合の好例の 1 つが、関数型インターフェースの追加です。

関数型インターフェース

一般的な Java イディオムは、`Runnable` や `Callable` などのメソッドが 1 つだけあるインターフェースです。これらのインターフェースは、SAM (Single Abstract Method) インターフェースと呼ばれています。多くの場合、SAM は主に移植可能なコードのトランスポート・メカニズムとして使用されます。Java 8 で移植可能なコードを実装するのに最善の方法は、ラムダ式ブロックを使用することであり、「関数型インターフェース」と呼ばれる賢いメカニズムが、ラムダ式と SAM を有用な方法で相互作用できるようにします。関数型インターフェースは、抽象メソッドが 1 つだけ定義されるインターフェースです (抽象メソッドの他に、複数のデフォルト・メソッドを定義することができます)。関数型インターフェースは既存の SAM インターフェースを増補し、従来の匿名内部クラスをラムダ式ブロックで置き換えられるようにします。例えば、`Runnable` インターフェースには現在、`@FunctionalInterface` アノテーションでフラグを立てることができるようになっています。このオプションのアノテーションは、コンパイラーに対し、`Runnable` がインターフェースであること (クラスや列挙ではないこと)、そしてアノテーションを付けられた型が関数型インターフェースの要件を満たすことを確認するように指示します。

ラムダ式ブロックが持つ、置換できる性質を示す一例として、Java 8 では以下のように、`Runnable` 匿名内部クラスの代わりにラムダ式ブロックを渡すことによって新しいスレッドを作成することができます。

```
new Thread(() -> System.out.println("Inside thread")).start();
```

関数型インターフェースをシームレスにラムダ式ブロックと統合するのに有効な箇所は、山ほどあります。関数型インターフェースが注目に値する革新となっている理由は、すでに確立されている Java イディオムと連動するところにあります。

デフォルト・メソッド

Java 8 では、インターフェースにデフォルト・メソッドを宣言することもできます。デフォルト・メソッドとは、インターフェース・タイプの中で宣言され、`default` キーワードでマークした、抽象化されていない (本体のある) 非静的パブリック・メソッドです。インターフェースの各デフォルト・メソッドは、そのインターフェースを実装するクラスに自動的に追加されるため、クラスをデフォルトの機能で修飾する便利な方法となります。例えば、`Comparator` インターフェースには現在、10 を超えるデフォルト・メソッドが組み込まれています。ラムダ式ブロックを使用してコンパレーターを作成する場合、逆コンパレーターは簡単に作成することができます (リスト 5 を参照)。

リスト 5. `Comparator` のデフォルト・メソッド

```
List<Integer> n = List.of(1, 4, 45, 12, 5, 6, 9, 101);
Comparator<Integer> c1 = (x, y) -> x - y;
Comparator<Integer> c2 = c1.reversed();
System.out.println("Smallest = " + n.stream().min(c1).get());
System.out.println("Largest = " + n.stream().min(c2).get());
```

リスト 5 では、ラムダ式ブロックの中にラップされた `Comparator` インスタンスを作成しています。こうすることで、`reversed()` デフォルト・メソッドを呼び出すことによって逆コンパレーターを作成することができます。デフォルト・メソッドをインターフェースにアタッチできるという機能は、ミックスインの一般的な使い方を模倣したものであり (記事「Java.next: ミックスインとトレイト」を参照)、Java 言語に追加された素晴らしい機能の 1 つです。

Optional

リスト 5 の終了処理の呼び出しでは、呼び出しのチェーニングによって `get()` が呼び出されていることに注意してください。`min()` などの組み込みメソッドを呼び出すと、値ではなく `Optional` が返されます。この動作は、「Java.next: Groovy、Scala、Clojure の共通点、第 3 回」で説明した Java.next の option 機能を模倣しています。`Optional` はメソッドの戻り値として、エラーとしての `null` を、正当な値としての `null` と混同させないためのものです。例えば、Java 8 の終了処理では、`fPresent()` メソッドを使用することで、正当な結果が存在する場合にだけコード・ブロックを実行することができます。その一例として、以下のコードは、値が存在する場合にのみ結果を出力します。

```
n.stream()
    .min((x, y) -> x - y)
    .ifPresent(z -> System.out.println("smallest is " + z));
```

追加のアクションを行う必要がある場合に使用できるメソッドには、`orElse()` もあります。デフォルト・メソッドによってどれだけの能力が追加されるかは、Java 8 の `Comparator` インターフェースを見ればはっきりとわかります。

Stream に関するその他の詳細

> “ *Java 8 の Stream インターフェースと関連機能は、Java 言語に新しい命を吹き込む、十分に考え抜かれた機能拡張です。* ”

Java 8 の「Stream」抽象化によって可能にされている高度な関数型の機能はさまざまにあります。Stream は多くの点でコレクションのように機能しますが、コレクションとは異なる主な点としては以下の内容が挙げられます。

- Stream は値を保管するのではなく、終了処理を介した入力ソースから宛先までのパイプラインとして機能します。
- Stream は、ステートフルではなく、関数型になるように意図されています。例えば、`filter()` 処理は、ベースとなるコレクションを変更することなく、フィルタリングされた値の Stream を返します。
- Stream 処理は、可能な限り遅延を試みます (「Java.next: メモ化、そして関数型での相乗効果」および「関数型の考え方: 遅延処理、第 1 回」を参照)。遅延コレクションは、値を取得しなければならない場合にだけ処理を行います。
- Stream は無制限 (または無限) にすることができます。例えば、すべての数値を返す Stream を作成し、`limit()` や `findFirst()` などのメソッドを使用してサブセットを収集することができます。

- `Iterator` と同じように、Stream は使用時に取り込まれます。それ以降に Stream を再び使用するには、その前に Stream を再生成する必要があります。

Stream の処理は、「中間」処理または「終了」処理のいずれかです。中間処理の場合は、新しい Stream を返し、常に遅延処理を行います。例えば、Stream に対して `filter()` 処理を使用する場合、これは実際に Stream をフィルタリングするのではなく、終了処理によってトラバースされるときにだけフィルタリングされた値を返す Stream を作成することになります。終了処理は Stream をトラバースして、値を生成するか、(推奨されませんが、副次的影響を生み出す関数を作成した場合には) 副次的影響を生み出します。

Stream には、すでに多くの有用な終了処理が組み込まれています。連載「関数型の考え方」で取り上げた数値分類子 (前の 2 回の「Java.next」の連載記事でも取り上げています) を例に用いると、Java 8 ではリスト 6 のように実装することになります。

リスト 6. Java 8 の数値分類子

```
public class NumberClassifier {

    public static IntStream factorsOf(int number) {
        return range(1, number + 1)
                .filter(potential -> number % potential == 0);
    }

    public static boolean isPerfect(int number) {
        return factorsOf(number).sum() == number * 2;
    }

    public static boolean isAbundant(int number) {
        return factorsOf(number).sum() > number * 2;
    }

    public static boolean isDeficient(int number) {
        return factorsOf(number).sum() < number * 2;
    }

}
```

他の言語での数値分類子のバージョンを十分に理解していれば (「関数型の考え方: 関数型の観点で考える、第 1 回」を参照)、リスト 6 には `sum()` メソッドの宣言がないことに気付くはずです。他の言語では例外なく、このコードを実装するのに `sum()` メソッドを自分で作成しなければなりませんでした。Java 8 には `sum()` が終了処理として組み込まれているため、これを自ら作成する必要はありません。関数型プログラミングでは、可変の構成要素を隠すことによって、開発者によるエラーの可能性を削減します。`sum()` を実装する必要がなければ、そもそも実装で間違いを犯すことすらできません。Java 8 の Stream インターフェースと関連機能は、Java 言語に新しい命を吹き込む、十分に考え抜かれた機能拡張です。

数値分類子の他のバージョンでは、`factors()` メソッドの最適化バージョンを紹介しました。この最適化バージョンでは、平方根までの約数の候補をトラバースして、ペアの約数を生成します。Java 8 での `factors()` メソッドの最適化バージョンは、リスト 7 に記載するとおりです。

リスト 7. Java 8 の最適化された数値分類子

```
    public static List fastFactorsOf(int number) {
        List<Integer> factors = range(1, (int) (sqrt(number) + 1))
                .filter(potential -> number % potential == 0)
                .boxed()
                .collect(Collectors.toList());
        List factorsAboveSqrt = factors
                .stream()
                .map(e -> number / e).collect(toList());
        factors.addAll(factorsAboveSqrt);
        return factors.stream().distinct().collect(toList());
    }
```

リスト 7 の `factorsOf()` メソッドは、2 つの Stream を単純に 1 つの結果に合成することはできませんが、Stream を連結することはできます。ただし、Stream はトラバースが完了した時点で使い果たされるため (`Iterator` と同様です)、再び使用するには、その前に再生成しなければなりません。リスト 7 では、Stream を使用した 2 つのコレクションを作成し、結果を連結し、整数平方根による重複のエッジ・ケースを処理するために `distinct()` の呼び出しを追加しています。Java 8 の Stream には、Stream を作成する機能を含め、圧倒的な能力があります。

まとめ

今回の記事では、Java.next 言語としての Java 8 を調査しました。その結果は、かなりの高得点になりました。優れた設計の Stream ライブラリー、そしてデフォルト・メソッドなどの賢い拡張メカニズムにより、既存の Java コードの大部分で、ほとんど手間をかけずに新機能のメリットを享受することができます。

次回の記事では、言語の選択に関するいくつかの考えをまとめて、この連載を締めくくります。

関連トピック

- Lambda Expressions: このチュートリアルで、新しい Java 8 の機能について詳しく学んでください。
- 「Java 8 言語での変更内容」(Dennis Sosnoski 著、developerWorks、2014年): Java 言語でのラムダ式と interface の変更についての批判的な見方を読んでください。
- 『Functional Programming in Java』(Venkat Subramaniam 著、Pragmatic Bookshelf、2014年): この素晴らしいリソースを調べてください。
- 『Functional Thinking』(Neal Ford 著、O'Reilly Media、2014 年): この連載の著者が書いた本で、関数型プログラミングについて詳しく学んでください。
- 「Java マルチテナンシーの紹介」(Graeme Johnson、Michael Dawson 共著、developerWorks、2014年6月): IBM Java 8 ベータ版における、クラウド・システムの新しい機能について学んでください。
- Scala: Scala は JVM 上で実行される最近の関数型言語です。
- Groovy は、Java 言語の動的バージョンとして Java の構文と機能が更新されたものです。
- Clojure: Clojure は JVM 上で実行される最近の関数型 Lisp です。
- developerWorks Java technology ゾーン: Java プログラミングのあらゆる側面を網羅した豊富な記事を調べてください。
- JDK 8 Project: Java 8 をダウンロードしてください。
- 「IBM SDK, Java Technology Edition Version 8」: IBM SDK for Java 8.0 Beta Program に参加してください。