Java.next: Java.next 言語

プログラミング言語の多言語化が進む世界で Groovy、Scala、Clojure を活用する

Neal Ford
Director / Software Architect / Meme Wrangler
ThoughtWorks Inc.

2013年 5月 30日

この記事は、Groovy、Scala、Clojure という 3 つの次世代 JVM 言語の詳細な比較を行うために、Neal Ford によって新たに開始される developerWorks の連載です。この第 1 回では、皆さんが現時点でメインのプログラミング言語として Java を使い続けることにしているかどうかに関わらず、この 3 つの言語の類似点と相違点を理解することによって何が得られるのかを説明します。
2013年 4月 16日 — 「参考文献」に「Common ground in Groovy, Scala, and Clojure, Part 1」と「Common ground in Groovy, Scala, and Clojure, Part 2」へのリンクを追加しました。

2013年 5月 14日 — 「参考文献」に「Common ground in Groovy, Scala, and Clojure, Part 3」へのリンクを追加しました。

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、それぞれの言語は Java 言語の機能を超える新たな興味深い機能をもたらしています。この連載では、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。この連載の目的は、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した上で、新しい言語の学習にどれだけの時間をかけるかの選択を十分な知識に基づいて行えるようにすることです。

かつて私が Martin Fowler 氏と共同で行った基調講演の中で、彼は次のような鋭い意見を述べました。

> 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。

Java 技術を最初に設計した技術者達は、言語をランタイムから分離するという素晴らしい決断を下しました。その結果、Java プラットフォーム上で実行できる言語は 200 を超えました。コンピューター・プログラミング言語の寿命は短いことが一般的であるため、このアーキテクチャー

は Java プラットフォームが長く存続する上で極めて重要です。2008年以降、Oracle の主催により毎年開催されている JVM Language Summit は、JVM 上で実行される代替言語を実装する人達に Java プラットフォーム技術者とのコラボレーションをオープンに行える機会を提供してきました。

Java.next の連載記事に興味をお持ちいただき、ありがとうございます。この連載では、パラダイム、設計上の選択、快適さが異なっていて興味深い、Groovy、Scala、Clojure という 3 つの最新の JVM 言語についての概要を説明します。これらの言語の詳細は、それぞれの言語の Web サイトに書かれているため、ここでは説明しません (「参考文献」を参照)。一方、言語コミュニティーの Web サイトは、伝道活動を主な目的としているため、客観的な情報や、それぞれの言語に不向きな処理の例が不足しています。この連載では内容のある比較を行い、そうした不足を補います。今回の記事ではその準備段階として、Java.next 言語の概要と、これらの言語を学ぶメリットについて説明します。

Java を超える

Java 言語は、Bruce Tate 氏が彼の著書『Beyond Java』(「参考文献」を参照) で「perfect storm (パーフェクト・ストーム)」と呼んでいる事態 — つまり Web の台頭、さまざまな理由で当時の Web 技術が不適当であったこと、そして企業による多層アプリケーション開発の増加といった要素が重なったこと — を通じて大いに注目されるようになりました。Tate 氏はさらにその著書の中で、このパーフェクト・ストームは極めてまれな出来事が重なったものであったと述べており、今後他のどのような言語も Java と同じような形で Java に匹敵するほど注目されるようにはならないだろう、とも述べています。

Java 言語は、機能が非常に柔軟であることが実証されていますが、その構文と本質的なパラダイムに制限があることが長年知られています。有望な変更が追加される予定であるにもかかわらず、Java 言語の構文は将来の重要な目標のいくつか (例えば、関数型プログラミングの要素など) をサポートすることができません。しかし 1 つの新しい言語で Java に取って代わるようなものを探そうとすると、それは決して見つからないものを探していることになってしまいます。

多言語プログラミング

多言語プログラミング — 私が 2006年のブログ記事 (「参考文献」を参照) で復活させ、再び広めた言葉 — は、1 つのプログラミング言語ではすべての問題を解決することはできない、という理解に基づいています。一部の言語には、特定の問題によく適した考え方や機能が組み込まれています。例えば、Swing ほどに洗練されていても、Swing UI を Java で作成しようとすると極めて厄介です。なぜなら、型宣言が必要であったり、使いにくい匿名内部クラスが動作に必要であったり、そのほかにも面倒な作業が必要になるからです。UI の作成によく適した言語 (例えば Groovy (`SwingBuilder` 機能 (「参考文献」を参照)) など) を使用すると、はるかに容易に Swing アプリケーションを作成することができます。

JVM 上で実行される言語が非常に増えたことから、ベースになるさまざまな言語のバイト・コードやライブラリーを保守する一方で組み合わせて使用できるため、多言語プログラミングの概念が一層魅力的なものになっています。例えば、`SwingBuilder` は Swing を置き換えるのではなく、既存の Swing API の上のレイヤーで機能します。もちろん、開発者達は長年、JVM の外部でさまざまな言語を取り混ぜて使用 (例えば、特定の目的で SQL と JavaScript を使用するなど) してきまし

たが、今やそれを JVM の内部で行うことがより一般的になりつつあります。ThoughtWorks では多くのプロジェクトに複数の言語を使用しており、ThoughtWorks Studios によって開発されたすべてのツールが複数の言語を取り混ぜて使用しています。

皆さんにとってメインの開発言語が Java のままであっても、Java に代わる他の言語がどのように動作するかを学ぶことにより、それらの言語を戦略的に採用できるようになります。Java は今後も JVM エコシステムの重要な一部でありつづけると思いますが、最終的にはプラットフォームのアセンブリー言語として — つまり純粋にパフォーマンス上の理由から、あるいは特殊な要件を満たすことのみを目的として — JVM エコシステムにとどまり続けることになると思います。

進化

1980年代初め、私が大学生だった頃、私達は Pecan Pascal と呼ばれる開発環境を使用していました。この環境にはユニークな機能があり、同じ Pascal コードを Apple II 上でも IBM PC 上でも実行することができました。Pecan の技術者達は「バイト・コード」と呼ばれる謎めいたものを使用することにより、この機能を実現していました。開発者は作成した Pascal コードをこの「バイト・コード」にコンパイルし、その「バイト・コード」をそれぞれのプラットフォーム用にネイティブに作成された「仮想マシン」上で実行しました。しかしそれは散々な体験でした。単純なクラスの割り当ての場合でさえ、コードは非常に低速で実行されました。当時のハードウェアは、とてもそうした難題に対応できなかったのです。

Pecan Pascal から 10 年後、Sun が Pecan Pascal と同じアーキテクチャーを使用した Java をリリースしました。それは非常に負荷が重いものでしたが、1990年代半ばのハードウェア環境では成功を収めました。また Java には自動ガーベッジ・コレクションなど、開発者にとって使い勝手のよい機能も追加されていました。C++ のような言語を扱ってきた経験から、私はもう一度ガーベッジ・コレクション機能のない言語でコーディングしたいとは決して思いません。私は、メモリー管理のような下位レベルの複雑な問題を解決する方法の検討に時間をかけるのではなく、上位レベルの抽象化によってビジネス上の複雑な問題を解決する方法を検討することに時間をかけたいと思います。

コンピューター言語の寿命は一般に短いものですが、その理由の 1 つは言語やプラットフォームの設計の革新が非常に速いためです。プラットフォームが強力になるにつれ、プラットフォームはより多くの作業を処理できるようになります。例えば (2010年に追加された) Groovy のメモ化 (memoization) 機能は、関数呼び出しの結果をキャッシュに入れます。キャッシング・コードをハンド・コーディングすることで、バグが発生する可能性を招くのではなく、リスト 1 に示すように単純に `memoize()` メソッドを呼び出す必要があります。

リスト 1. Groovy での関数のメモ化

```
def static sum = { number ->
  factorsOf(number).inject(0, {i, j -> i + j})
}
def static sumOfFactors = sum.memoize()
```

リスト 1 では、`sumOfFactors` メソッドを呼び出した結果は自動的にキャッシュに入れられます。また、`memoizeAtLeast()` と `memoizeAtMost()` という別のメソッドでキャッシング動作をカスタマイズすることもできます。Clojure にもメモ化の機能があり、Scala でメモ化を実装するのも簡単です。次世代言語 (そして一部の Java フレームワーク) に備わるメモ化などの高度な機能は、次第

に Java 言語にも追加されるでしょう。Java の次期リリースでは高階関数が追加され、メモ化の実装がはるかに容易になるはずです。次世代の Java 言語を学ぶことで、将来のJava の機能を垣間見ることができます。

Groovy、Scala、Clojure

Groovy は 21 世紀の Java 構文 — つまり普通のコーヒーではなくエスプレッソ — です。Groovy の設計の目標は、Java 言語の基本的なパラダイムをサポートしつつ Java 構文を更新して面倒な部分を減らすことです。つまり、Groovy は JavaBeans などを認識し、プロパティーへのアクセスを単純化しています。Groovy は、この連載の今後の記事で取り上げる重要な関数型機能などの、新しい機能を急速に取り入れています。それでもなお、Groovy は基本的にオブジェクト指向の命令型言語です。Java と Groovy との基本的な違いは、Groovy は静的型付け言語ではなく動的型付け言語であること、そして Groovy のメタプログラミング機能は Java よりもはるかに優れていることの 2 点です。

Scala は JVM を活用するためにゼロから設計された言語ですが、構文は完全に再設計されています。Scala は強い静的型付けの言語 — Java よりも型付けは厳格でありながら、型の扱いは Java よりも容易です — であり、オブジェクト指向と関数型の両方のパラダイムをサポートしています (ただし、関数型のパラダイムの方が推奨されます)。例えば Scala では、よりなじみのある可変変数を作成する `var` よりも、(Java で変数を `final` と修飾する場合と同様に) 不変変数を生成する `val` 宣言の方が推奨されます。Scala は、オブジェクト指向と関数型の両方のパラダイムを十分にサポートすることにより、おそらく皆さんの現在の姿である、オブジェクト指向の命令型プログラマーから、おそらく皆さんが将来なるはずの姿である、より関数型のプログラマーへの橋渡しをしてくれます。

Clojure は Lisp の方言であり、その構文は他の言語とはかなり異なっています。Groovy と同じく強い動的型付けの言語である Clojure には、設計に関する独特の決定が反映されています。Clojure にはレガシー Java との完全かつ詳細な相互運用性がありますが、古いパラダイムとの橋渡しになろうとはしていません。例えば、明らかに関数型である Clojure は、レガシー Java との相互運用性のためにオブジェクト指向をサポートしており、オブジェクト指向プログラマーが使い慣れた機能 (例えば、ポリモーフィズムなど) をすべてサポートしていますが、それらはオブジェクト指向の方法ではなく、関数型の方法によるサポートです。Clojure は、新しい機能を優先して古いパラダイムを打ち破る、いくつかのコアとなるエンジニアリングの原則 (ソフトウェア・トランザクショナル・メモリーなど) を基に設計されています。

パラダイム

構文を除くと、Groovy、Scala、Clojure という 3 つの言語間の最も興味深い違いは、型付けの違い、そして根底にある基本的なパラダイム (つまり、関数型か命令型か) の違いです。

静的型付けと動的型付けの比較

プログラミング言語の分類において、静的型付け言語とは Java の `int x;` 宣言のように明示的な型宣言をする言語を指し、動的型付け言語とは宣言に型情報を必要としない言語を指します。また、記事で取り上げる言語はすべて強い型付けの言語であるため、これらの言語で書かれたコードでは、型が割り当てられると、その型に基づいた検討が可能になります。

Java の型システムは、静的型付けの不便さが多い割にメリットが少ないとして、多くの批判にさらされてきました。例えば、現在のように限定的ながら型推論が利用できるようになるまでは、Java を使用する開発者は代入文の両辺で型宣言を繰り返す必要がありました。Scala は Java よりも型付けが静的ですが、型推論が多用されるため、日常的に使用する上では Java よりもはるかに手軽です。

Groovy には、一見、静的型付けと動的型付けを橋渡ししているような 1 つの動作があります。リスト 2 に示す単純なコレクション・ファクトリーの例を考えてみてください。

リスト 2. Groovy のコレクション・ファクトリー

```
class CollectionFactory {
  def List getCollection(description) {
    if (description == "Array-like")
      new ArrayList()
    else if (description == "Stack-like")
      new Stack()
  }
}
```

リスト 2 のクラスはファクトリーとして動作し、渡された `description` パラメーターに基づいて、`List` インターフェースを実装する 2 つ — `ArrayList` または `Stack` — のいずれかを返します。これを Java 開発者が見ると、どちらが返されても確実に条件を満たしているように思えます。しかし、リスト 3 に示す 2 つのユニット・テストは、そう簡単ではないことを明らかにします。

リスト 3. Groovy におけるコレクションの型のテスト

```
@Test
void test_search() {
  List l = f.getCollection("Stack-like")
  assertTrue l instanceof java.util.Stack
  l.push("foo")
  assertThat l.size(), is(1)
  def r = l.search("foo")
}

@Test(expected=groovy.lang.MissingMethodException.class)
void verify_that_typing_does_not_help() {
  List l = f.getCollection("Array-like")
  assertTrue l instanceof java.util.ArrayList
  l.add("foo")
  assertThat l.size(), is(1)
  def r = l.search("foo")
}
```

リスト 3 の最初のユニット・テストでは、ファクトリーを介して `Stack` を取得し、それが実際に `Stack` であることを検証した後、`push()`、`size()`、`search()` といった Stack らしい処理を実行しています。しかし 2 番目のユニット・テストでは、テストをパスするためには `MissingMethodException` という期待される例外でテストを保護しなければなりません。`Array-like` コレクションを取得して `List` 型の変数に格納すると、戻り値として実際にリストを受け取ることを検証することができます。しかし `search()` メソッドを呼び出そうとすると、`ArrayList` には `search()` メソッドが含まれていないため、例外がトリガーされます。つまりこの宣言では、メソッドの呼び出しが適切であることを保証する、コンパイル時の保護が提供されなかったのです。

これはバグのように見えますが、適切な動作なのです。Groovy では、型が保証するのは代入文の妥当性のみです。例えば、リスト 3 で `List` インターフェースを実装していない何かを返したとすると、ランタイム例外 (`GroovyCastException`) がトリガーされます。このことから、Groovy は確実に、Clojure とともに強い動的型付け言語のグループに含まれます。

しかし最近 Groovy に加えられた変更により、Groovy での静的型付けと動的型付けの区別は一層あいまいになりました。Groovy 2.0 で `@TypeChecked` アノテーションが追加された結果、型チェックの厳密さをクラス・レベルまたはメソッド・レベルで随時決定できるようになりました。リスト 4 は @TypeChecked アノテーションを説明したものです。

リスト 4. アノテーションによる型チェック

```
@TypeChecked
@Test void type_checking() {
    def f = new CollectionFactory()
    List l = f.getCollection("Stack-like")
    l.add("foo")
    def r = l.pop()
    assertEquals r, "foo"
}
```

リスト 4 で追加した `@TypeChecked` アノテーションでは、代入文とそれに続くメソッド呼び出しの両方を検証しています。例えば、リスト 5 のコードはコンパイルされないはずです。

リスト 5. 無効なメソッド呼び出しを防ぐ型チェック

```
@TypeChecked
@Test void invalid_type() {
    def f = new CollectionFactory()
    Stack s = (Stack) f.getCollection("Stack-like")
    s.add("foo")
    def result = s.search("foo")
}
```

リスト 5 では、ファクトリーからの戻りに対して型キャストを追加し、`Stack` の `search()` メソッドを呼び出せるようにする必要があります。ただし、この機能には制約があり、静的型付けが有効になっている場合、Groovy の動的機能の多くは動作しません。それでもこの例は、静的型付けと動的型付けとの橋渡しのために Groovy が進化を続けていることを示しています。

これらの言語はどれも、非常に強力なメタプログラミング機能を持っています。そのため、より厳格な型付けを後から追加することができます。例えば、選択的型付けを Clojure に追加するためのサイド・プロジェクトがいくつか存在しています。ただし、選択的型付けがオプションの場合は、選択的型付けは型システムの一部ではなく、検証メカニズムにすぎないことが一般的です。

命令型と関数型の比較

Groovy、Scala、Clojure を比較する上でのもう 1 つの基準は、命令型か関数型かという点です。命令型プログラミングは、ステップバイステップの命令にフォーカスしており、多くの場合、かつての下位レベルのハードウェアの簡便性を模倣しています。関数型プログラミングは、第一級構成体としての関数にフォーカスしており、状態遷移と可変性を最小限にしようとします。

Groovy は Java の影響を大きく受けているため、基本的には命令型言語です。しかしその一方で、当初から関数型言語の機能が数多く含まれており、これまでさらに多くの関数型の機能が追加されてきました。

Scala はこれら 2 つのパラダイムにまたがっており、両方をサポートしています。Scala は関数型プログラミングを優先 (そして推奨) していますが、それでもオブジェクト指向プログラミングと命令型プログラミングをサポートしています。つまり Scala を適切に使用するためには、やみくもに 2 つのパラダイムを混在させることがないように、訓練されたチームが必要です。マルチパラダイム言語には、不適切にパラダイムを混在させる危険が常に付きまといます。

Clojure は明らかに関数型です。Clojure は他の JVM 言語とのやり取りを容易にするためにオブジェクト指向をサポートしていますが、命令型と関数型を橋渡ししようとはしていません。むしろ、Clojure の独特の設計には、設計者がエンジニアリングの適切なプラクティスであると考える内容が表れています。設計に関するこうした決定は、実にさまざまな結果をもたらすこととなり、Clojure では Java の世界を悩ませ続けてきたいくつかの問題 (例えば、並行性など) を画期的な方法で解決できるようになっています。

これらの新しい言語を学ぶには、考え方のシフトが必要です。そして必要なシフトの多くは、命令型と関数型の違いによるものです。これについてはさまざまな内容をこの連載で取り上げていく予定です。

まとめ

開発者達は、プログラミング言語の多言語化が進む世界に身を置いています。こうした世界では、さまざまな問題を解決するために複数のプログラミング言語が使われることから、問題の解決にどの方法が適しているかを判断する上で実質的に効果があるのは、新しい言語の活用方法を学ぶことです。たとえ皆さんが Java を使い続けるとしても、次世代の JVM 言語の機能は次第に Java にも組み込まれていくはずです。それらの機能を調べてみると、Java 言語の将来を垣間見ることができます。

連載「Java.next」の次回の記事では、Groovy、Scala、Clojure のすべてに共通する点を調べることで、この 3 つの言語の比較を始めることにします。

著者について

Neal Ford

Neal Ford は世界的な IT コンサルティング企業 ThoughtWorks のディレクターであり、ソフトウェア・アーキテクトであり、Meme Wrangler でもあります。また彼は、アプリケーション、教育資料、雑誌記事、コースウェア、ビデオや DVD によるプレゼンテーションなどの設計と開発も行っています。さまざまな技術に関する本の著者、編集者でもあり、最新の著作は『Presentation Patterns』です。彼は大規模なエンタープライズ・アプリケーションの設計や構築を専門にしています。また彼は世界各地で開催される開発者会議での講演者としても国際的に有名です。彼の Web サイトをご覧ください。