Java.next: Groovy、Scala、Clojure の共通点、第 2 回

Java.next 言語では定型的な処理や複雑さがいかに減少しているかを学ぶ

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

2013年 6月 20日

Java 言語に関する共通の不満は、単純なタスクやデフォルトにも定型的な処理がありすぎて、混乱を招きがちであることに関連しています。3 つの Java.next 言語ではいずれも、これらの領域でより賢明な手法を取っています。連載「Java.next」の今回の記事では、Groovy、Scala、Clojure が Java 言語で扱いにくかった部分をどのように解決しているかを紹介します。
2013年 5月 06日 — 著者の要望により、リスト 9 の構文糖とその説明文に情報を追加しました。

2013年 5月 14日 — 「参考文献」に「Groovy、Scala、Clojure の共通点 第 3 回」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。

Java プログラミング言語は、現在私たちが直面している状況とは異なる制約のなかで誕生しました。特に、Java 言語にプリミティブ型が存在する理由は、1990年代中頃のハードウェアにはパフォーマンスとメモリーに制約があったためです。その頃に比べると、Java 言語は Autoboxing によって煩雑さをかなり解消するまでに進化していますが、Java.next 言語 — Groovy、Scala、Clojure — はさらに一歩先を行き、言語全体にわたって整合性に欠ける部分や問題を生じがちな部分をなくす方向へと向かっています。

今回の記事では、構文およびデフォルトの振る舞いの両面で、Java.next 言語が Java 言語の一般的な制約をどのようにして排除しているかを説明します。最初に取り上げる制約は、プリミティブ・データ型の存在です。

プリミティブ型の消滅

Java 言語は、プリミティブ型とそれに対応する型ラッパー・クラスの 8 つのペアから始まりました。これは、元々はパフォーマンスとメモリーの制約に対処するために考えられたものでした。その後、Autoboxing によってプリミティブ型とラッパー・クラスの区別は次第に曖昧になってきましたが、Java.next 言語はそれよりも大幅に進んでいます。開発者にとって、プリミティブ型とラッパー・クラスの違いはもはや存在しないかのように思えるほどです。

Groovy では、プリミティブ型の存在が完全に開発者から隠されています。例えば、`int` は常に `Integer` を表しています。また、Groovy では数値のオーバーフローによるエラーを防ぐために、数値型を自動的に上位の型へ変換します。リスト 1 に示す Groovy シェルでの対話の一例を見てください。

リスト 1. Groovy でのプリミティブ型の自動処理

```
groovy:000> 1.class
===> class java.lang.Integer
groovy:000> 1e12.class
===> class java.math.BigDecimal
```

リスト 1 の Groovy シェルを見ると、定数でさえも、そのベースにあるクラスによって表されています。すべての数値 (およびその他のプリミティブ型に見えるもの) は実際にはクラスであることから、メタプログラミング手法を使用できます。これらの手法には、ドメイン特化言語 (DSL) を作成する際によく使われる、数値にメソッドを追加するという手法も含まれます。これにより、`3.cm` といった表現を使用できるようになります。この機能については、拡張性について話題にする今後の記事で詳しく説明します。

Groovy と同じく、Clojure でもプリミティブ型とラッパーの間の違いを隠し、あらゆる型に対してメソッドを呼び出せるようになっています。容量節約のための型変換は、自動的に行われます。Clojure が表面下でカプセル化する膨大な数の最適化については、この言語のドキュメントで詳しく説明されています (「参考文献」を参照)。Clojure では多くの場合に、開発者が型のヒントを提供することによって、コンパイラーにより高速なコードを生成させることができます。クリティカル・セクションには、例えば `(defn sum[x] ... )` とメソッドを定義する代わりに `(defn sum[^float x] ... )` のようにして型のヒントを追加すると、より処理効率に優れたコードが生成されます。

Scala も同じくプリミティブ型とラッパーの違いを隠していて、通常、コードのタイム・クリティカルな部分には表面下でプリミティブ型を使用します。さらに、例えば `2.toString` などのように、定数でメソッドを呼び出すこともできます。プリミティブ型と `Integer` などのラッパーを臨機応変に組み合わせる機能が備わった Scala は、Java の Autoboxing よりも表面化でさまざまなことを行っています。例えば、Scala の `==` 演算子は、Java の `==` 演算子とは異なり、プリミティブ型でもオブジェクト参照でも適切に機能します (つまり、参照を比較するのではなく、値を比較します)。さらに、Scala に組み込まれている `eq` メソッド (およびその対称となる `ne` メソッド) は、常にそのベースとなる参照型が等しいかどうか (または等しくないかどうか) を比較します。基本的

に、Scala はデフォルトの振る舞いを賢く切り替えているということです。Java 言語では、`==` が参照を比較しますが、参照を比較するケースはほとんどありません。一方、値を比較するのは、それほど直観的ではない `equals()` です。Scala では、そのベースとなる実装が何であるかに関わらず、`==` は常に適切な処理 (値の比較) を行い、めったに使用されない参照の等価性のチェックにはメソッドで対応します。

こうした Scala の特徴は、Java.next 言語を使用することでもたらされる主なメリットの 1 つが、開発者は下位レベルの詳細を言語とランタイムに任せて上位レベルの問題に集中できる点にあることを示しています。

デフォルトの簡素化

ほとんどの Java 開発者たちの間で一致しているほぼ普遍的な意見として、Java 言語には一般的な処理に必要な構文があまりにも多すぎるというものがあります。例えば、プロパティーの定義やその他のボイラープレート・コードによって、クラス定義があふれかえり、重要なメソッドがわかりにくくなる結果となります。一方、すべての Java.next 言語には、コードの作成および使用を簡素化する手段があります。

Scala でのクラスと case クラス

Scala では、アクセサー、ミューテーター、およびコンストラクターを自動的に作成することによって、クラス定義をすでに簡素化しています。リスト 2 に、Java クラスの例を記載します。

リスト 2. Jave での簡潔な `Person` クラス

```
class Person {
    private String name;
    private int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return name + " is " + age + " years old.";
    }
}
```

リスト 2 でボイラープレート・コードでないのは、オーバーライドされた `toString()` メソッドだけです。コンストラクターとすべてのメソッドは、IDE によって生成されています。けれども、コードを素早く作成することよりも大きな価値があるのは、コードを後で読んで簡単に理解で

きることです。不必要な構文は、その根底にある意味を理解する前に、使用しなければならないコードの量を増やすことになります。

Scala の `Person` クラス

驚くことに、Scala で作成されたリスト 3 のコードは、わずか 3 行の単純な定義で前の例と同じクラスを作成しています。

リスト 3. Scala での同じクラス

```
class Person(val name: String, var age: Int) {
  override def toString = name + " is " + age + " years old."
}
```

リスト 3 の `Person` クラスは、可変の `age` プロパティー、不変の `name` プロパティー、2 つのパラメーターからなるコンストラクター、そしてオーバーライドされた `toString()` メソッドに要約されています。肝心な部分が構文の中に埋もれていないため、このクラスの特徴がすぐにわかるはずです。

Scala の設計で重視しているのは、余分な構文を最小限にしてコードを作成できるようにすることです。そのため、多くの構文はオプションになっています。リスト 4 に、ストリングを大文字に変更するクラスを冗長に記述したコードを記載します。

リスト 4. 冗長なクラス

```
class UpperVerbose {
  def upper(strings: String*) : Seq[String] = {
    strings.map((s:String) => s.toUpperCase())
  }
}
```

リスト 4 のコードの大部分はオプションです。リスト 5 に同じコードを示しますが、今度は `class` ではなく、`object` になっています。

リスト 5. 大文字に変換する簡潔なオブジェクト

```
object Up {
  def upper(strings: String*) = strings.map(_.toUpperCase())
}
```

Scala では Java の静的メソッドに相当するものとして、クラスではなく、`object` を作成します。`object` は、Scala に組み込みのシングルトン・インスタンスです。リスト 4 にはあった、メソッドの戻り型、単一行のメソッド本体を区切る山括弧、そして不要な `s` パラメーターは、リスト 5 ではいずれも取り除かれています。この「折り畳み式構文」は、Scala の長所でもあり、短所でもあります。折り畳み式構文を使用すると、極めて慣用的なコードを記述することが可能ですが、その反面、初心者にとっては難解なコードになる可能性があります。

case クラス

オブジェクト指向のシステムでは、オブジェクト指向ではないシステムとやりとりしなければならない場合は尚更のこと、データ・ホルダーとしての役割を果たす単純なクラスを使用すること

がよくあります。このようなタイプのクラスが普及していることから、Scala プロジェクトではさらに踏み込んで、case クラスを作成するという結果に至りました。case クラスは自動的に、以下のような構文上の便宜を図ります。

- クラスの名前に基づくファクトリー・メソッドを作成することができます。例えば、新しいインスタンスを作成するには、`new` キーワードに煩わされることなく、`val bob = Person("Bob", 42)` のようなコードにすることができます。
- クラスのパラメーター・リストに含まれるすべての引数には、自動的に `val` が付けられます (つまり、不変の内部フィールドになります)。
- コンパイラーがクラスに応じて適切なデフォルトの `equals()`、`hashCode()`、および `toString()` メソッドを生成します。
- コンパイラーはクラスに `copy()` メソッドを追加するため、新しいコピーを返すことによって、変更を加えたバージョンを作成することができます。

Java.next 言語は構文上の欠点を修正するだけでなく、最近のソフトウェアが機能する仕組みを十分に理解した上で、その仕組みに合わせて機能を形にしています。

Groovy の自動生成プロパティー

Java.next 言語のうち、Java 構文に最も準拠している Groovy では、一般的なケースに対して構文糖によるコードを生成します。リスト 6 に、Groovy で作成された単純な `Person` クラスを記載します。

リスト 6. Groovy での `Person` クラス

```
class Person {
  private name
  def age

  def getName() {
    name
  }

  @Override
  String toString() {
    "${name} is ${age} years old."
  }
}

def bob = new Person(name: "Bob", age:42)

println(bob.name)
```

リスト 6 の Groovy コードでは、フィールド `def` を定義することによって、アクセサーとミューテーターの両方が生成されます。いずれか一方だけにしたい場合は、上記の `name` プロパティーと同じように自分で定義することもできます。このメソッドには `getName()` という名前が付けられていますが、より直観的な `bob.name` という構文でこのメソッドにアクセスできることに変わりはありません。

Groovy に `equals()` メソッドと `hashCode()` メソッドのペアを自動的に生成させるには、クラスに `@EqualsAndHashCode` アノテーションを追加します。このアノテーションは Groovy の AST (Abstract Syntax Tree: 抽象構文木) 変換を使用して、指定されたプロパティーに基づくメソッドを

生成します (「参考文献」を参照)。デフォルトでは、このアノテーションはプロパティーだけを考慮します (フィールドは考慮されません)。フィールドも考慮させるには、`includeFields=true` 修飾子を追加します。

Clojure のマップ風レコード

Clojure でも他の言語と同じく `Person` クラスを作成できますが、Clojure で作成すると、イディオムによる慣用的なコードにはなりません。従来、Clojure のような言語は、このような情報を保持するためにマップ (名前と値のペア) データ構造を利用しており、関数を使用してこの構造を扱います。マップで構造化データをモデル化することもできますが、最近の一般的なケースではレコードを使用するようになっています。Clojure でのレコードとは、(多くの場合にネストされた) プロパティーを持つ型名をより形式的にカプセル化したものであり、これらのプロパティーのどれもがインスタンスごとに同じセマンティックな意味を持ちます (Clojure でのレコードは、C 系の言語での `struct` のようなものです)。

一例として、以下に個人の定義を記載します。

```
(def mario {:fname "Mario"
            :age "18"})
```

この構造を前提とすると、`age` には `(get mario :age)` でアクセスすることができます。単純なアクセスは、マップでは一般的な処理です。Clojure の場合、マップではキーがアクセサー関数の役割をするという構文糖を利用して、さらに短縮された `(:age mario)` を使用できます。Clojure ではマップに対する操作が求められるため、マップの操作を簡単に行えるようにする構文糖がふんだんに用意されています。

Clojure には、リスト 7 に示すように、ネストされたマップ要素にアクセスするための構文糖もあります。

リスト 7. Clojure での簡潔なアクセス

```
(def hal {:fname "hal"
          :age "17"
          :address {:street "Enfield Tennis Academy"
                    :city "Boston"
                    :state "MA"}})

(println (:fname hal))
(println (:city (:address hal)))
(println (-> hal :address :city))
```

リスト 7 で定義しているのは、`hal` という名前のネストされたデータ構造です。外側の要素にアクセスするには、当然 `(:fname hal)` を使用します。リスト 7 の最後から 2 番目の行に示されているように、Lisp 構文は、「インサイド・アウト (内側から外側への方向)」で評価します。したがって、最初に `hal` から `address` レコードを取得してから、`city` フィールドにアクセスする必要があります。Clojure では「インサイド・アウト (内側から外側への方向)」での評価が一般的な記述方法であるため、特殊な演算子、`->` スレッド演算子が用意されています。この演算子は、`(-> hal :address :city)` のように式をひっくり返すことでより自然に読めるようにします。

レコードを使用して、上記に相当する構造を作成することができます (リスト 8 を参照)。

リスト 8. レコードを使用した構造の作成

```
(defrecord Person [fname lname address])
(defrecord Address [street city state])
(def don (Person. "Don" "Gately"
           (Address. "Ennet House" "Boston", "MA")))

(println (:fname don))
(println (-> don :address :city))
```

`defrecord` を使用して作成したリスト 8 の構造は、従来のクラス構造により近いものになっています。Clojure では、お馴染みのマップ処理とイディオムを使用して、同じように簡単にレコード構造の内部にアクセスすることができます。

Clojure 1.2 では、レコード定義の一般的な処理に関する構文糖が以下の 2 つのファクトリー関数として追加されました。

- `->TypeName`: フィールドの位置関連のパラメーターを受け入れます。
- `map->TypeName`: キーワードをフィールド値にマッピングします。

イディオムのような関数を使用した場合、リスト 8 のコードはリスト 9 のようなコードになります。

リスト 9. さらに簡潔な Clojure の構文糖

```
(def don (->Person "Don" "Gately"
  (->Address "Ennet House" "Boston", "MA")))
```

マップや単純な構造よりもレコードが優先される状況は多々あります。第一に、`defrecord` は Java クラスを作成して、マルチメソッド定義の中で簡単に使用できるようにします。第二に、`defrecord` はより多くの処理をこなし、レコードを定義する際にフィールドの検証やその他の細かい作業を可能にします。第三に、レコードは処理速度がはるかに高速です。よく知られたキーの一定のセットを使用する場合は、特に高速です。

Clojure ではレコードとプロトコルを一緒に使用してコードを構造化します。この 2 つの関係については、今後の記事で詳しく探ります。

まとめ

3 つすべての Java.next 言語には、Java 言語と比べ、構文上の便宜が図られています。Groovy と Scala では、どちらもクラスと共通ケースを簡単に作成できるようになっています。Clojure では、マップ、レコード、およびクラスをシームレスに使用することができます。すべての Java.next 言語に共通のテーマは、不要なボイラープレート・コードの排除です。次回の記事では、引き続きこのテーマを取り上げ、例外について説明します。

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。