Java.next: Groovy、Scala、Clojure の共通点、第 1 回

3 つの次世代 JVM 言語が演算子の多重定義にどう対処しているかを探る

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

2013年 6月 13日

Java.next 言語 (Groovy、Scala、Clojure) の間には、相違点よりも共通点の方が多く、機能や便利さの多くの点で共通する方向に向かっています。今回の記事では、演算子を多重定義 (オーバーロード) できないという Java 言語の昔からの問題に、3 つの言語がそれぞれどのように対処しているかを探ります。また、関連する概念である、演算子の結合規則と優先順位についても説明します。

2013年 4月 16日 — 「参考文献」に「Java.next: Java.next 言語」と「Java.next: Groovy、Scala、Clojure の共通点、第 2 回」へのリンクを追加しました。

2013年 5月 14日 — 「参考文献」に「Java.next: 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。

プログラミング言語における優れた概念は、長く存続するとともに、他の言語にも波及して次第に浸透していきます。従って、Java.next 言語 — Groovy、Scala、Clojure — に共通する機能が数多くあることは驚くに当たりません。連載「Java.next」の今回およびそれに続く記事では、各言語の関数が共通する方向へ向かっていることが、それぞれの言語の構文にどのように表れているかを探ります。まずは、Java 言語の昔からの問題を補完する機能 — 演算子を多重定義 (オーバーロード) する機能 — から始めます。

演算子の多重定義

Java の `BigDecimal` クラスをいじったことのある人であれば、リスト 1 のようなコードを見たことがあるはずです。

リスト 1. Java コードでの `BigDecimal` の貧弱なサポート

```
BigDecimal op1 = new BigDecimal(1e12);
BigDecimal op2 = new BigDecimal(2.2e9);
// (op1 + (op2 * 2)) / (op1/(op1 + (op2 * 1.5e2))
BigDecimal lhs = op1.add(op2.multiply(BigDecimal.valueOf(2)));
BigDecimal rhs = op1.divide(
        op1.add(op2.multiply(BigDecimal.valueOf(1.5e2))),
            RoundingMode.HALF_UP);
BigDecimal result = lhs.divide(rhs);
System.out.println(String.format("%,.2f", result));
```

リスト 1 は、コメントに記載した式を実現しようとしています。Java プログラミングでは、算術演算子を多重定義することはできないため、メソッドの呼び出しを使用せざるを得ません。静的インポートが役に立つ場合もありますが、特定のコンテキストに適した演算子を多重定義する、という明確なニーズが存在しています。最初に Java を設計した技術者達は、演算子を多重定義できるようにすると複雑になりすぎると考え、意図的に Java 言語では演算子を多重定義できないようにしました。しかしこれまで Java が使われてきた中で明らかなのは、この機能がないことで開発者に強いられる複雑さの方が、この機能が乱用された場合の複雑さを上回っていることです。

方法は少しずつ異なりますが、3 つの Java.next 言語はいずれも演算子の多重定義を実装しています。

Scala の演算子

Scala では、演算子とメソッドの区別をなくすことによって、演算子を多重定義できるようにしています。Scala での演算子は、特殊な名前を持つメソッドにすぎません。例えば、乗算演算子をオーバーライドするには、`*` メソッドをオーバーライドします (Scala では、* は有効なメソッド名であり、このことがインポート命令に Java のアスタリスク (*) 文字ではなく、アンダーバー (_) 文字を使用する 1 つの理由です)。

ここでは多重定義の説明をするために、複素数を使用します。複素数は、実部と虚部を含む数学的表記であり、通常は例えば `3 + 4i` のように書きます (「参考文献」を参照)。工学、物理学、電磁気学、カオス理論などの多くの科学分野で、複素数はよく使われています。リスト 2 は Scala で複素数を実装したものです。

リスト 2. Scala で実装した複素数

```
final class Complex(val real: Int, val imaginary: Int) {
  require (real != 0 || imaginary != 0)

  def +(operand: Complex) =
      new Complex(real + operand.real, imaginary + operand.imaginary)

  def +(operand: Int) =
    new Complex(real + operand, imaginary)

  def -(operand: Complex) =
    new Complex(real - operand.real, imaginary - operand.imaginary)
```

```
  def -(operand: Int) =
    new Complex(real - operand, imaginary)

  def *(operand: Complex) =
      new Complex(real * operand.real - imaginary * operand.imaginary,
          real * operand.imaginary + imaginary * operand.real)

  override def toString() =
      real + (if (imaginary < 0) "" else "+") + imaginary + "i"

  override def equals(that: Any) = that match {
    case other : Complex => (real == other.real) && (imaginary == other.imaginary)
    case _ => false
  }

  override def hashCode(): Int =
    41 * ((41 + real) + imaginary)
}
```

equals() と match キーワード

リスト 2 で、もう 1 つの興味深い特徴は、equals() メソッドの中でパターン・マッチングが使われていることです。Scala では型キャストもできますが、型を突き合わせる方が一般的です。that パラメーターは、Scala の継承階層の最上位である Any と宣言されています。equals() メソッドの本体は match の呼び出しで構成されており、この呼び出しは渡された型が一致するとフィールドの値を確認し、型が一致しない場合はデフォルトで false を返します。

Scala では、Java 言語における冗長な部分のほとんどを簡素化するために、不要な土台の部分をなくしています。例えば、リスト 2 のクラスでは、コンストラクターのパラメーターとフィールドはクラス定義のところに記述されています。この場合、クラスの本体はコンストラクターとして動作するため、`require()` メソッドを呼び出すと、値が存在するかどうかの検証が最初のインスタンス化アクションとして実行されます。Scala では自動的にフィールドが提供されるため、クラスの残りの部分にはメソッド定義が含まれています。`+`、`-`、`*` 演算子に関しては、引数として複素数 `Complex` を取るメソッドを演算子そのままの名前で宣言しています。複素数の乗算は加算や減算ほど単純ではありません。リスト 2 の多重定義された `*` メソッドは以下の公式を実行します。

```
(x + yi)(u + vi) = (xu - yv) + (xv + yu)i
```

リスト 2 の `toString()` メソッドは、Java.next 言語の間のちょっとした共通点、つまり「文の代わりに式を使用する」という共通点を例示しています。`toString()` メソッドでは、虚部が正の場合はプラス (+) 符号を付ける必要がありますが、それ以外の場合は虚部の暗黙的なマイナス符号をそのまま使えば十分です。Scala では、`if` は文ではなく式であるため、Java の場合のような三項演算子 (`?:`) は必要ありません。

実際には、追加された `+`、`-`、`*` メソッドは標準の演算子と見分けがつきません。それを示すものがリスト 3 のユニット・テストです。

リスト 3. Scala で定義した複素数をテストする

```
class ComplexTest extends FunSuite {
  test("addition") {
    val c1 = new Complex(1, 3)
    val c2 = new Complex(4, 5)
```

```
    assert(c1 + c2 === new Complex(1+4, 3+5))
  }

  test("subtraction") {
    val c1 = new Complex(1, 3)
    val c2 = new Complex(4, 5)
    assert(c1 - c2 === new Complex(1-4, 3-5))
  }

  test("multiplication") {
    val c1 = new Complex(1, 3)
    val c2 = new Complex(4, 5)
    assert(c1 * c2 === new Complex(
        c1.real * c2.real - c1.imaginary * c2.imaginary,
        c1.real * c2.imaginary + c1.imaginary * c2.real))
  }
}
```

リスト 3 のテストでは、興味深い不整合があることを明らかにすることができません。その問題については、このすぐ後で「結合規則」について説明する際に明らかにし、解決方法を示すことにします。その前に、まずは Groovy と Clojure が演算子をどのように多重定義するかについて説明します。

Groovy のマッピング

Groovy では、オーバーライド可能なマッピング・メソッドを提供することにより、すべての Java 演算子を多重定義します (例えば `+` 演算子をオーバーライドするには、`Integer` クラスの `plus()` メソッドをオーバーライドします)。Groovy における演算子の多重定義についての詳細は、私が寄稿した関数型言語の拡張性に関する連載「関数型の考え方」の記事「関数型のデザイン・パターン、第 3 回」で、今回の記事と同じく複素数の例を用いて説明しています。

Groovy では、もちろん新しいメソッドを作成することはできますが、新しい演算子を作成することはできません。Spock テスティング・フレームワーク (「参考文献」を参照) などの一部のフレームワークでは、難解ながらも存在している演算子 (`>>>` など) を多重定義しています。Scala と Clojure では、方法はまったく異なりますが、演算子とメソッドを一様に扱います。

Groovy には、新しい便利な演算子もいくつか導入されています。例えば、「セーフ・ナビゲーション演算子」(`?.`) は、呼び出し側がどれもヌルではないことを保証します。「エルビス (Elvis) 演算子」(`?:`) は、Java の三項演算子を短縮したものであり、デフォルト値を簡単に提供したい場合に便利です。Groovy には、これらの新しい演算子のための拡張メソッドはなく、新しい演算子を多重定義できないようにしています。それに、開発者達がこれらの演算子を多重定義したいと思う明白な理由があるわけでもありません。演算子を多重定義する場合の典型的な理由は、演算子を使用した以前の経験を活かしてコードを読みやすくすることにあります。Groovy 以外では、これらの演算子を使用する可能性は低そうです。便利さのために演算子を使用しても読みにくいコードになるようであれば、演算子の多重定義は危険なものとなります。

Clojure の演算子

Scala の場合と同じように、Clojure の演算子はシンボル名を持つメソッドにすぎません。そのため、例えばカスタム型の `+` メソッドを簡単に作成することができます。ただし Clojure で適切に演算子をオーバーライドするには、共通のコアから一連のメソッドを生成するためのプロトコルと手法を理解する必要があります。それについては今後の記事で説明することにします。

結合規則

演算子の結合規則とは、演算子が式の左側のメソッドなのか、それとも右側のメソッドなのかについて言及するものです。Scala では、基本的にすべてのメソッドが演算子として機能できるため、他のほとんどの言語とは異なる方法でホワイト・スペースを使用します。例えば `x + y` という式は、実際には `x.+(y)` というメソッド呼び出しになります。それを Scala REPL (インタープリター) セッションで示したものがリスト 4 です。

リスト 4. Scala でのホワイト・スペースの変換

```
scala> val sum1 = x.+(y)
sum1: Int = 22

scala> val sum2 = (12).+(10)
sum2: Int = 22
```

リスト 4 を見ると、ホワイト・スペースの変換が定数に対しても機能することがわかります。Scala では、お望みであればすべてのメソッドを演算子として扱うことができます。例えば `String` クラスの `indexOf()` メソッドは、引数として渡された文字が、文字列内でどの位置にあるかを示すインデックス位置を返します。Scala では、このメソッドを昔ながらの方法で `s.indexOf('a')` のように呼び出すことも、`s indexOf 'a'` のように演算子として呼び出すこともできます。(このメソッドには興味深いことに、検索を開始するインデックス位置を指定するための追加パラメーターを受け付けるように、多重定義されたメソッドがあります。そのメソッドを呼び出す場合にも、演算子の表記を使用することができます。ただし、その際は `s indexOf('a', 3)` のようにパラメーターを括弧に入れる必要があります。)

Groovy は Java の結合規則に従っており、特定の演算子に対する規則は Java 言語の場合と同じです。Clojure には、結合規則に関する懸念はまったくありません。すべてのステートメントが一義に表されるため、Clojure の Lisp 構文が結合規則を必要とすることはありません。

Scala の場合、目標の 1 つがすべてを演算子として使用できるようにすることなので、任意の結合規則に従うというわけにはいきません。その場限りの演算子を許容しながらも、規則を確立するには、言語としてどうすればよいのでしょう。Scala はこの問題を、開発者が最大限の自由を発揮できる革新的な方法、つまり演算子に命名規則を採用するという方法で解決しています。Scala の演算子はデフォルトで左結合であるため、式は左オペランドでのメソッド呼び出しを行うこととなり、例えば `x + y` という式は `x.+(y)` と解釈されます。一方でメソッド名の最後が `:` の場合には、演算子は右結合となり、例えば `i +: j` を呼び出すと `j.+:(i)` と解釈されます。

リスト 3 のテストでは全容が明らかにされない理由は、結合規則を考えるとわかります。リスト 2 における Scala の `Complex` クラスの定義では、私が実装している `+` 演算子と `-` 演算子は、`Complex` 型を引数に取るバージョンと、`Int` 型を引数に取るバージョンがあります。このように型を柔軟にすることで、複素数と通常の整数 (虚部がゼロの複素数) との間で演算をすることができます。リスト 5 は、複素数と整数との演算が可能であるかどうかをユニット・テストで確認するコードです。

リスト 5. 型が混在する場合のテスト

```
test("mixed addition from Complex") {
  val c1 = new Complex(1, 3)
  assert(new Complex(7, 3) == c1 + 6)
}

test("mixed subtraction from Complex") {
  val c1 = new Complex(10, 3)
  assert(new Complex(5, 3) == c1 - 5)
}
```

リスト 5 のテストの結果は、両方とも問題なく合格で、`Int` バージョンの演算子メソッドが呼び出されます。一方、以下のテストを実行すると、結果は不合格となります。

```
test("mixed subtraction from Int") {
  val c1 = new Complex(10, 3)
  assert(new Complex(15, 3) == 5 + c1)
}
```

この 2 つのテストの間の微妙な違いは、結合規則に関するものです。Scala では左側の演算子のメソッドを呼び出すことを思い出してください。つまりこのテストは、複素数を扱う方法を認識している、`Int` に対して定義されたメソッドを呼び出そうとしているのです。

この問題を解決するには、`Int` と `Complex` との間に暗黙的なキャストを定義します。この型変換には、いくつかの方法がありますが、それについては今後の記事で詳細に説明します。この例の場合には、以下のようにコンパニオン・オブジェクト (Java 言語では `static` と宣言されるメソッドが格納される場所) を作成し、`Complex` と呼ぶことにします。

```
final object Complex {
  implicit def intToComplex(x: Int) = new Complex(x, 0)
}
```

この定義には、`Int` を受け付け、それを `Complex` として返す 1 つのメソッドが含まれています。ここでは、この宣言を `Complex` クラスと同じソース・ファイルに配置し、`import nealford.javaNext.complexnumbers.Complex.intToComplex` コマンドを実行することで、このメソッドをテスト・ケースにインポートして、暗黙的な型変換を実現します。この型変換をスコープ内に用意すると、演算子によって行われるメソッド呼び出しの処理方法を、テストが認識するため、このテスト・ケースの実行結果は合格となります。

優先順位

演算子の優先順位 (つまり演算の順序) とは、演算の実行順序に曖昧さがあり得る状況での実行順序を定めた、言語の規則を指しています。Groovy では、一般的な演算子に関しては Java の優先順位の規則に従っており、カスタム演算子に関しては独自の規則を定義します。Clojure には、優先順位の規則はなく、規則を定める必要もありません。すべてのコードは括弧を使用した形式で作成され、中置記法に特有の曖昧さは決して生じないからです。

Scala では、演算子の名前の先頭文字で演算の順序を判断します。その優先順位の階層は以下のとおりです。

- 他のすべての特殊文字

- `/ %`
- `+ -`
- `:`
- `= !`
- `< >`
- `&`
- `^`
- `|`
- すべての文字
- すべての代入演算子

ランクの高い文字で始まる演算子は優先順位が高くなっています。例えば `x *** y ||| z` という式は `(x.***(y)).|||(z)` と解釈されます。この規則の唯一の例外は、代入文 (等号 (`=`) で終わるすべての演算子) に関するものであり、それらの演算子は自動的に優先順位が最も低くなります。

まとめ

Java.next 言語の共通の目標は、Java 言語に影響を与えている厄介な制約を緩和することです。Java.next の各言語がこの問題にどのように取り組んでいるかを示す格好の例が、演算子の多重定義です。Java.next の 3 つの言語は、どれも演算子の多重定義を許容していますが、それをどのように実装しているかはそれぞれに異なっています。結合規則や優先順位などの問題の処理方法に関する微妙な違いを見ると、言語の各部分がどのように関係し合っているのかを理解することができます。Clojure の興味深い側面の 1 つは構文です。すべての式は最初から括弧で囲まれているため、優先順位や結合規則に関する曖昧さが排除されています。

次回の記事では、「すべてのものがオブジェクト」という考え方が、どれほど深く Java.next 言語に根付いているかを探ります。

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。