関数型の考え方: 関数型のデザイン・パターン、第 3 回

Interpreter パターンと言語の拡張

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2012年 6月 14日

Gang of Four (GoF) の Interpreter デザイン・パターンは、ある言語から新しい言語を作成することによって、その言語を拡張するように促すパターンです。ほとんどの関数型言語では、演算子の多重定義やパターン・マッチングなど、さまざまな方法で言語を拡張できるようになっています。Java ではこのような拡張手法を一切使用できませんが、次世代の JVM 言語では、実装の詳細は異なるとはいえ、言語の拡張が可能になっています。この記事では Neal Ford が、Groovy、Scala、および Clojure で Interpreter デザイン・パターンの目的を実現するために、Java では使用できない方法で関数型言語を拡張する方法を調査します。

このシリーズの他の記事を見る

この連載について

この連載の目的は、読者の皆さんの考え方を関数型の発想へと方向転換し、よくある問題を新たな考え方で検討することによって、日常的なコーディングの改善方法を見つけるお手伝いをすることです。そのために、関数型プログラミングに特徴的な概念、関数型プログラミングを Java 言語で行えるようにするフレームワーク、JVM 上で動作する関数型プログラミング言語、そして今後、言語設計を学習する上での方向性などについて詳しく探ります。この連載の対象読者は、Java および Java の抽象化がどのように機能するかは知っていても、関数型言語を使用した経験がほとんど、あるいはまったくない開発者です。

連載「関数型の考え方」では今回も引き続き、Gang of Four (GoF) のデザイン・パターン (「参考文献」を参照) 手法に代わる、関数型のソリューションを調査します。今回の記事で取り上げるのは、GoF のデザイン・パターンのなかで最も強力ながらも、最も理解されていない Interpreter パターンです。

Interpreter パターンの定義は以下のとおりです。

> ある特定の言語に対し、文法の表現を定義し、さらにはその表現を使用して文を解釈するインタープリターを定義すること。

別の言葉で言い換えると、現在使用している言語が特定の問題に適切でない場合、その言語を使って新しい言語を作ってしまうということです。この手法の好例は、Grails や Ruby on Rails な

どの Web フレームワークにあります (「参考文献」を参照)。Grails および Ruby on Rails は、それぞれの基本言語 (Groovy、Ruby) を拡張して Web アプリケーションを作成しやすくしています。

このパターンが「最も理解されていない」理由は、新しい言語を作成するのはまれなことで、言語を新たに作成するには特殊なスキルとイディオムが必要だからです。その一方、このパターンは解決しようとしている問題に合わせて、使用しているプログラミング言語を拡張するように促すことから、デザイン・パターンのなかで「最も強力な」パターンとなっています。このように言語を拡張することは Lisp の世界では (したがって Clojure の世界でも) 一般的な考え方ですが、主流の言語ではそれほど普及していません。

言語自体の拡張が許可されていない言語 (例えば、Java) を使用している開発者は、言語の構文の枠にはまった考え方をしがちです。それは、そうするしかないからです。けれども簡単に拡張できる言語を使い慣れてくると、問題の解決を言語に適応させるのではなく、問題を解決できるように、言語を適応させるようになってきます。

アスペクト指向プログラミングを使用するのでない限り、Java には直接的な言語拡張メカニズムがありません。けれども、次世代の JVM 言語である Groovy、Scala、および Clojure (「参考文献」を参照) では例外なく、さまざまな方法で言語を拡張できるようになっています。これにより、これらの言語は Interpreter デザイン・パターンの目的を実現します。この記事ではまず、この 3 つの JVM 言語で演算子の多重定義を実装する方法を説明し、次に Groovy と Scala で既存のクラスを継承する方法を説明します。

演算子の多重定義

関数型言語に共通の機能は、演算子の多重定義です。つまり、新しい型を処理して新しい動作を行わせるように演算子 (例えば、`+`、`-`、`*` など) を再定義することができます。Java の形成期には、意図的に演算子の多重定義が排除されていましたが、Java を自然に受け継ぐ JVM 言語を含め、最近のほとんどすべての言語では演算子を多重定義できるようになっています。

Groovy

Groovy の目的は、Java 本来のセマンティクスを維持しつつも、Java の構文を最近の言語で使われているような構文へと変えることです。したがって、Groovy では演算子をメソッド名に自動的にマッピングするという方法で、演算子を多重定義できるようにしています。例えば、`Integer` の `+` 演算子を多重定義する場合には、`Integer` クラスの `plus()` メソッドをオーバーライドするといった具合です。マッピングの全リストはインターネットで調べることができます (「参考文献」を参照)。表 1 に、マッピングの一部を記載します。

表 1. Groovy の演算子/メソッドのマッピングの抜粋

演算子	メソッド
`x + y`	`x.plus(y)`
`x * y`	`x.multiply(y)`
`x / y`	`x.div(y)`
`x ** y`	`x.power(y)`

演算子を多重定義する一例として、`ComplexNumber` クラス (複素数クラス) を Groovy と Scala の両方で作成してみます。複素数とは、実数の部分 (実部) と虚数の部分 (虚部) を持つ数学的概念で、通常は `3 + 4i` のように表現されます。複素数は多くの科学分野 (エンジニアリング、物理学、電磁気学、カオス理論など) で一般的に使用されています。これらの分野のアプリケーションを作成している開発者にとって、対象とする問題領域を反映する演算子を作成できるということは大いに役立ちます (複素数についての詳細は、「参考文献」を参照してください)。

リスト 1 に、Groovy で作成した `ComplexNumber` クラスを記載します。

リスト 1. Groovy での `ComplexNumber`

```
package complexnums

class ComplexNumber {
   def real, imaginary

  public ComplexNumber(real, imaginary) {
    this.real = real
    this.imaginary = imaginary
  }

  def plus(rhs) {
    new ComplexNumber(this.real + rhs.real, this.imaginary + rhs.imaginary)
  }

  def multiply(rhs) {
    new ComplexNumber(
        real * rhs.real - imaginary * rhs.imaginary,
        real * rhs.imaginary + imaginary * rhs.real)
  }

  String toString() {
    real.toString() + ((imaginary < 0 ? "" : "+") + imaginary + "i").toString()
  }
}
```

リスト 1 では、実部と虚部の両方を格納するクラスを作成し、多重定義した `plus()` 演算子と `multiply()` 演算子を作成しています。2 つの複素数を加算するのは簡単です。つまり `plus()` 演算子では、2 つの複素数の実部と虚部をそれぞれ加算することで、結果が生成されます。一方、2 つの複素数を乗算する場合には、以下の式を使用する必要があります。

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

リスト 1 の `multiply()` 演算子は、上記の式を再現したもので、2 つの複素数の実部同士の積から虚部同士の積を減算したものを実部とし、複素数の互いの実部と虚部を乗算して得られる 2 つの積を加算したものを虚部とします。

リスト 2 で、複素数演算子をテストします。

リスト 2. 複素数演算子のテスト

```
package complexnums

import org.junit.Test
import static org.junit.Assert.assertTrue
import org.junit.Before
```

```
class ComplexNumberTest {
  def x, y

  @Before void setup() {
    x = new ComplexNumber(3, 2)
    y = new ComplexNumber(1, 4)
  }

  @Test void plus_test() {
    def z = x + y;
    assertTrue 3 + 1 == z.real
    assertTrue 2 + 4 == z.imaginary
  }

  @Test void multiply_test() {
    def z = x * y
    assertTrue(-5  == z.real)
    assertTrue 14 == z.imaginary
  }
}
```

リスト 2 の `plus_test()` メソッドと `multiply_test()` メソッドで多重定義された演算子 (どちらも、ドメイン・エキスパートが使用する記号で表現されています) を使用している方法は、組み込み型を同じように使用する際の方法と見分けがつかないほどです。

Scala (および Clojure)

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。つまり、演算子は特殊な名前の付いた単なるメソッドに過ぎません。したがって、Scala で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。リスト 3 では、Scala で複素数を作成しています。

リスト 3. Scala での複素数

```
class ComplexNumber(val real:Int, val imaginary:Int) {
    def +(operand:ComplexNumber):ComplexNumber = {
        new ComplexNumber(real + operand.real, imaginary + operand.imaginary)
    }

    def *(operand:ComplexNumber):ComplexNumber = {
        new ComplexNumber(real * operand.real - imaginary * operand.imaginary,
            real * operand.imaginary + imaginary * operand.real)
    }

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

リスト 3 のクラスには、お馴染みの `real` メンバーと `imaginary` メンバーの他、`+` および `*` 演算子/メソッドが含まれています。リスト 4 を見るとわかるように、`ComplexNumbers` は自然な形で使用することができます。

リスト 4. Scala で複素数を使用する例

```
val c1 = new ComplexNumber(3, 2)
val c2 = new ComplexNumber(1, 4)
val c3 = c1 + c2
assert(c3.real == 4)
assert(c3.imaginary == 6)

val res = c1 + c2 * c3

printf("(%s) + (%s) * (%s) = %s\n", c1, c2, c3, res)
assert(res.real == -17)
assert(res.imaginary == 24)
```

演算子とメソッドを一体化することによって、Scala では演算子を容易に多重定義できるようにしています。Clojure で演算子を多重定義する場合も、これと同じメカニズムを使用します。以下は、Clojure コードで `**` 演算子を多重定義する例です。

```
(defn ** [x y] (Math/pow x y))
```

クラスの継承

次世代の JVM 言語では、演算子の多重定義と同様に、Java 言語自体では不可能な方法でクラス (Java のコアとなるクラスを含む) を継承することができます。これらの継承機能は、ドメイン特化言語 (Domain-Specific Language: DSL) を作成するときによく使用されます。DSL は GoF の視野にまったく入っていませんでしたが (当時よく使われていた言語では DSL が普及していなかったためです)、DSL は Interpreter デザイン・パターン本来の目的を示す良い例です。

例えば `Integer` などのコア・クラスに単位やその他の修飾子を追加すれば、(演算子を追加する場合と同じく) 実際の問題をより綿密にモデル化することができます。このような追加は、Groovy でも Scala でも可能です。ただし、使用するメカニズムは異なります。

Groovy の `Expando` およびカテゴリー・クラス

Groovy には、既存のクラスにメソッドを追加するためのメカニズムとして、`ExpandoMetaClass` とカテゴリー・クラスの 2 つがあります (前回の記事では、Adapter パターンのコンテキストで `ExpandoMetaClass` について詳しく説明しました)。

例えば、ある企業では一風変わった昔ながらの理由により、速度を表記する方法として、1 時間あたりのマイル数 (Miles Per Hour: MPH) ではなく、2 週間あたりのハロン数 (Furlong per Fortnight: FF) を使用することになっているとします。開発者は MPH から FF への変換を度々行っていることに気付きました。Groovy の `ExpandoMetaClass` を使用すれば、`FF` プロパティーを `Integer` に追加することで、この変換に対処することができます (リスト 5 を参照)。

リスト 5. `ExpandoMetaClass` を使用して Integer に FF 単位を追加する

```
static {
  Integer.metaClass.getFF { ->
    delegate * 2688
  }
}

@Test void test_conversion_with_expando() {
  assertTrue 1.FF == 2688
}
```

ExpandoMetaClass の代わりに、カテゴリー・ラッパー・クラスという Objective-C から採り入れた概念を使用することもできます。リスト 6 では、この概念を使って (今度は小文字の) ffプロパティーを Integer に追加しています。

リスト 6. カテゴリー・クラスを使用して単位を追加する

```
class FFCategory {
  static Integer getFf(Integer self) {
    self * 2688
  }
}

@Test void test_conversion_with_category() {
  use(FFCategory) {
    assertTrue 1.ff == 2688
  }
}
```

カテゴリー・クラスは、静的メソッドの集合を持つ通常のクラスです。各メソッドは、少なくとも 1 つの引数を取り、1 つ目の引数が、このメソッドで増補する型です。例えば、リスト 6 の FFCategory クラスには getFf() メソッドがあり、このメソッドは Integer 型の引数を取ります。このカテゴリー・クラスを use キーワードと一緒に使用すると、コード・ブロック内の該当するすべての型が増補されます。ユニット・テストでは、リスト 6 の一番下に示されているようにすれば、コード・ブロック内の ff プロパティーを参照することができます (Groovy は get メソッドを自動的にプロパティー参照に変換することを思い出してください)。

2 つのメカニズムから選択することで、増補の範囲をより正確に制御することができます。例えば、システム全体で MPH がデフォルトの速度単位として使用されている一方、ff への変換が度々必要になるとしたら、ExpandoMetaClass を使用してグローバルに変更したほうが適切です。

読者のなかには、コアとなる JVM クラスを再オープンすることの有用性には懐疑的な人もいるかもしれません。それは、JVM クラスを再オープンする影響が広範囲に及ぶことを懸念しているためですが、カテゴリー・クラスを使用すれば、危険を伴う可能性のある拡張による影響の範囲を制限することができます。ここで、このメカニズムを上手に使用している実在のオープンソース・プロジェクトの例を紹介します。

easyb プロジェクト (「参考文献」を参照) では、テスト対象のクラスが持つ側面を検証するテストを作成することができます。一例として、リスト 7 に記載する easyb テストのスニペットを見てください。

リスト 7. easyb による queue クラスのテスト

```
it "should dequeue items in same order enqueued", {
    [1..5].each {val ->
        queue.enqueue(val)
    }
    [1..5].each {val ->
        queue.dequeue().shouldBe(val)
    }
}
```

テストの検証フェーズで呼び出す `shouldBe()` メソッドは `queue` クラスに含まれていませんが、このメソッドは、easyb フレームワークによって自動的に追加されています。その仕組みは、easyb のソースに含まれる `it()` メソッド定義を見るとわかります (リスト 8 を参照)。

リスト 8. easyb の `it()` メソッド定義

```
def it(spec, closure) {
  stepStack.startStep(BehaviorStepType.IT, spec)
  closure.delegate = new EnsuringDelegate()
  try {
    if (beforeIt != null) {
      beforeIt()
    }
    listener.gotResult(new Result(Result.SUCCEEDED))
    use(categories) {
      closure()
    }
    if (afterIt != null) {
      afterIt()
    }
  } catch (Throwable ex) {
    listener.gotResult(new Result(ex))
  } finally {
    stepStack.stopStep()
  }
}

class BehaviorCategory {
  // ...

  static void shouldBe(Object self, value) {
    shouldBe(self, value, null)
  }

  //...
}
```

リスト 8 の `it()` メソッドは、spec (テストを記述する文字列) と、closure (テスト本体を表現するクロージャー・ブロック) を引数に取ります。メソッドの中間地点で、このクロージャーが `BehaviorCategory` ブロック (リストの一番下にあるブロック) 内で実行されます。この `BehaviorCategory` が `Object` を増補して、Java のあらゆるインスタンスがそのインスタンス自体の値を検証できるようにするという仕組みです。

Groovy のオープン・クラスのメカニズムは、階層の最上位にある `Object` を選択的に増補できるようにすることで、あらゆるインスタンスで簡単に結果を検証できるようにすると同時に、その変更を `use` ブロックの本体に制限します。

Scala での暗黙のキャスト

Scala では暗黙のキャストを使用して、既存のクラスの増補をシミュレートします。暗黙のキャストはメソッドをクラスに追加するわけではありません。暗黙のキャストによって、Scala 言語がオブジェクトを、目的のメソッドを持つ適切な型へと自動的に変換できるようにします。例えば、`isBlank()` メソッドを `String` クラスに追加することはできませんが、暗黙のキャストを作成することで、`String` を、このメソッドを持つクラスに自動的に変換することができます。

リスト 9 は、`Person` インスタンスを適切な型の配列に簡単に追加できるように、`append()` メソッドを `Array` に追加する例です。

リスト 9. 人を追加するためのメソッドを `Array` に追加する

```
case class Person (firstName: String, lastName: String) {}

class PersonWrapper(a: Array[Person]) {
  def append(other: Person) = {
    a ++ Array(other)
  }
  def +(other: Person) = {
    a ++ Array(other)
  }
}

implicit def listWrapper(a: Array[Person]) = new PersonWrapper(a)
```

リスト 9 では、いくつかのプロパティーを持つ単純な `Person` クラスを作成しました。`Array[Person]` (Scala の Generics は、`< >` ではなく `[ ]` を区切り文字として使用します) を `Person` 対応にするために、`PersonWrapper` クラスを作成し、そこに目的の `append()` メソッドを組み込みます。リストの最後で作成している暗黙のキャストが、配列で `append()` メソッドが呼び出されると、自動的に `Array[Person]` を `PersonWrapper` に変換します。リスト 10 では、この変換をテストしています。

リスト 10. 既存のクラスへの自然な拡張のテスト

```
val p1 = new Person("John", "Doe")
var people = Array[Person]()
people = people.append(p1)
```

リスト 9 では、`+` メソッドも `PersonWrapper` クラスに追加しています。リスト 11 に、この見事に直観的な演算子バージョンを使用する方法を示します。

リスト 11. さらに読みやすくなるように言語を変更する

```
people = people + new Person("Fred", "Smith")
for (p <- people)
  printf("%s, %s\n", p.lastName, p.firstName)
```

Scala では、実際に元のクラスにメソッドを追加しているわけではなく、自動的に適切な型に変換することで、メソッドを追加しているように見せかけています。暗黙のキャストを多用し過ぎて、相互接続されたクラスが複雑に入り組む結果とならないようにするためには、Groovy のような言語でメタプログラミングを行う際に必要な配慮が Scala でも必要です。けれども、正しく使用すれば、暗黙のキャストは非常に表現豊かなコードの作成に役立ちます。

まとめ

GoF によるオリジナルの Interpreter デザイン・パターンは新しい言語の作成を提言していますが、彼らが前提とした言語では、現在私たちが意のままに使えるしなやかな拡張メカニズムがサポートされていませんでした。次世代の Java 言語は例外なく、さまざまな手法によって言語レベルで拡張性をサポートします。今回の記事では、演算子の多重定義が Groovy、Scala、および Clojure でどのように機能するかを説明し、Groovy と Scala でのクラスの継承を検討しました。

今後の記事では、Scala スタイルのパターン・マッチングと Generics の組み合わせによって、従来のいくつかのデザイン・パターンを置き換える方法を紹介します。その話題に欠かせない概念は、関数型のエラー処理でも活躍します。それが、次回の記事のトピックです。

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である ThoughtWorks のソフトウェア・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、最新の著書は『プロダクティブ・プログラマ — プログラマのための生産性向上術』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。