関数型の考え方: ディスパッチの再考

次世代の JVM 言語でメソッド・ディスパッチに追加された、Java とは微妙に異なる内容

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2012年 9月 20日

Java プラットフォームのための次世代の言語には、Java 言語よりも柔軟なメソッド・ディスパッチ・メカニズムがあります。連載「関数型の考え方」の今回の記事では、Neal Ford が Scala や Clojure などの関数型言語のディスパッチ・メカニズムを探り、コードの実行に対する新しい考え方を紹介します。

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。

前回の記事では、Java の Generics を使用して Scala のパターン・マッチングを模倣することで、簡潔で読みやすい条件文を作成できることを説明しました。Scala のパターン・マッチングは、ディスパッチ・メカニズムとして選択可能な方法の 1 つの例です。ここで私は「ディスパッチ・メカニズム」という言葉を「言語が振る舞いを動的に選択するさまざまな仕組み」のことを表す広義の言葉として使っています。今回の記事ではこの話題を各種の関数型 JVM 言語に広げ、それぞれの言語のディスパッチ・メカニズムが Java 言語でのディスパッチ・メカニズムをさらに簡潔かつ柔軟にする仕組みを明らかにします。

Groovy によるディスパッチの改善

Java での条件付き実行は、`switch` 文が適用される極めて限られたケースを除き、結局は `if` 文を使用することになります。`if` 文が長々と連なると読みにくくなってしまうため、Java 開発者は GoF

(Gang of Four) の Factory (または Abstract Factory) パターン (「参考文献」を参照) を利用しています。より柔軟な決定式を持つ言語を使用すれば、コードのかなりの部分をさらに簡潔にすることができます。

Groovy には、Java の `switch` 文の構文を模倣する (ただし、振る舞いは模倣しない) 強力な `switch` 文があります。リスト 1 を見てください。

リスト 1. 大幅に改善された Groovy の `switch` 文

```
class LetterGrade {
  def gradeFromScore(score) {
    switch (score) {
      case 90..100 : return "A"
      case 80..<90 : return "B"
      case 70..<80 : return "C"
      case 60..<70 : return "D"
      case 0..<60  : return "F"
      case ~"[ABCDFabcdf]" : return score.toUpperCase()
      default: throw new IllegalArgumentException("Invalid score: ${score}")
    }
  }
}
```

Groovy の `switch` 文は、多種多様な動的型を受け入れます。例えば、リスト 1 の `score` パラメーターは、0 から 100 までの数値、あるいは文字で表現された等級のいずれかになります。Groovy では、Java の場合と同様に、それぞれの `case` は `return` または `break` で終了しなければならず、これらが指定されない場合はフォールスルー動作をしますが、その一方で Java とは異なり、範囲 (`90..100`) を指定すること、終了値を含めない範囲 (`80..<90`) を指定すること、正規表現 (`~"[ABCDFabcdf]"`) を指定すること、デフォルトの条件を指定することができます。

また、Groovy では動的型付けができるため、異なる型のパラメーターを送信しても適切な応答が返ってくるようにすることができます。リスト 2 のユニット・テストに、その一例を示します。

リスト 2. Groovy の文字等級のテスト

```
@Test
public void test_letter_grades() {
  def lg = new LetterGrade()
  assertEquals("A", lg.gradeFromScore(92))
  assertEquals("B", lg.gradeFromScore(85))
  assertEquals("D", lg.gradeFromScore(65))
  assertEquals("F", lg.gradeFromScore("f"))
}
```

より強力な `switch` 文は、一連の `if` 文と Factory デザイン・パターンの間の有益な妥協点を提供します。Groovy の `switch` 文を使用すれば、範囲やその他の複雑な型とのマッチングが可能になります。このマッチングは Scala のパターン・マッチングと似たような趣旨のものです。

Scala のパターン・マッチング

Scala のパターン・マッチングでは、マッチング・ケースと、そのそれぞれのケースに応じた振る舞いを指定することができます。前回の記事で取り上げた、文字等級の例を検討してみましょう (リスト 3 を参照)。

リスト 3. Scala での文字等級

```
val VALID_GRADES = Set("A", "B", "C", "D", "F")

def letterGrade(value: Any) : String = value match {
  case x:Int if (90 to 100).contains(x) => "A"
  case x:Int if (80 to 90).contains(x) => "B"
  case x:Int if (70 to 80).contains(x) => "C"
  case x:Int if (60 to 70).contains(x) => "D"
  case x:Int if (0 to 60).contains(x) => "F"
  case x:String if VALID_GRADES(x.toUpperCase) => x.toUpperCase
}
```

Scala では、パラメーターの型を `Any` として宣言することで、動的な入力を許可することができます。上記で適用する演算子は `match` です。この演算子は、マッチする最初の true 条件を見つけると、結果を返します。リスト 3 に示されているように、ケースごとに条件を指定するガード条件を含めることができます。

リスト 4 に、いくつかの文字等級の選択肢を実行した結果を記載します。

リスト 4. Scala での文字等級のテスト

```
printf("Amy scores %d and receives %s\n", 91, letterGrade(91))
printf("Bob scores %d and receives %s\n", 72, letterGrade(72))
printf("Sam never showed for class, scored %d, and received %s\n", 44, letterGrade(44))
printf("Roy transfered and already had %s, which translated as %s\n",
    "B", letterGrade("B"))
```

Scala でのパターン・マッチングは、多くの場合、Scala の case クラスと一緒に使用されます。case クラスは、代数やその他の構造化データ型を表現するように意図されています。

融通の効く Clojure 言語

Clojure も、Java プラットフォームのための次世代の関数型言語の 1 つです (「参考文献」を参照)。JVM での Lisp の実装である Clojure は、最近のほとんどの言語とは著しく異なる構文を使用します。開発者がこの構文に順応するのは簡単ですが、主流の Java 開発者には奇異な印象を与えます。Lisp 言語ファミリーが持つ最も優れた特徴の 1 つは、同図像性です。同図像性とは、言語がその独自のデータ構造を使用して実装されることを意味します。したがって、他の言語では到底不可能なまでの拡張が可能になります。

Java および Java に似た言語には、「キーワード」、つまりその言語の構文上の基礎を成すものがあります。開発者が言語の新しいキーワードを作成することは不可能であり (ただし、Java に似た言語のなかには、メタプログラミングによる拡張が可能な言語もあります)、キーワードには開発者が使用できないセマンティクスがあります。その一例は、Java の `if` 文です。Java の if 文は、ブール式の短絡評価を行います。Java でメソッドやクラスを作成することはできても、基本的なビルディング・ブロックを作成することはできないため、「問題をこのプログラミング言語の構文に変換する」必要があります (実際、多くの開発者は自分たちの仕事を、この変換を行うことだと思っています)。Clojure のような Lisp の変形では、開発者が「問題に合わせて言語を変更する」ことができます。そのため、言語の設計者が作成できるものと、その言語を使って開発者が作成できるものとがはっきりと区別されていません。同図像性が持つ意味については、今後の記事で詳しく探ることとして、ここで理解しなければならない重要な特性は、Clojure (およびその他の Lisp 関連の言語) の背後にある設計思想です。

Clojure では、開発者がこの言語を使用して、読みやすい (Lisp) コードを作成します。リスト 5 に、Clojure で作成した場合の文字等級の例を記載します。

リスト 5. Clojure の文字等級

```
(defn letter-grade [score]
  (cond
    (in score 90 100) "A"
    (in score 80 90)  "B"
    (in score 70 80)  "C"
    (in score 60 70)  "D"
    (in score 0 60)   "F"
    (re-find #"[ABCDFabcdf]" score) (.toUpperCase score)))

(defn in [score low high]
  (and (number? score) (<= low score high)))
```

リスト 5 ではまず、`letter-grade` という理解しやすいメソッドを作成し、次に、このメソッドを機能させるための `in` メソッドを実装しました。このコードで、`in` メソッドによって処理される一連のテストを評価するために使用しているのは、`cond` 関数です。これまでに記載した例と同じく、数値と既存の等級文字列の両方を処理します。最終的な戻り値は大文字にする必要があるので、入力が小文字になっている場合には、返された文字列で `toUpperCase` メソッドを呼び出します。Clojure では、メソッドはクラスではなく第一級オブジェクトなので、メソッド呼び出しは「逆順」になります。具体的に説明すると、Java での `score.toUpperCase()` 呼び出しは、Clojure では (`(.toUpperCase score)`) に相当します。

リスト 6 で、Clojure の文字等級の実装をテストします。

リスト 6. Clojure の文字等級のテスト

```
(ns nealford-test
  (:use clojure.test)
  (:use lettergrades))


(deftest numeric-letter-grades
  (dorun (map #(is (= "A" (letter-grade %))) (range 90 100)))
  (dorun (map #(is (= "B" (letter-grade %))) (range 80 89)))
  (dorun (map #(is (= "C" (letter-grade %))) (range 70 79)))
  (dorun (map #(is (= "D" (letter-grade %))) (range 60 69)))
  (dorun (map #(is (= "F" (letter-grade %))) (range 0 59))))

(deftest string-letter-grades
  (dorun (map #(is (= (.toUpperCase %)
           (letter-grade %))) ["A" "B" "C" "D" "F" "a" "b" "c" "d" "f"])))

(run-all-tests)
```

この例でのテスト・コードは、テスト対象の実装よりも複雑です！けれどもこのことが、Clojure コードがいかに簡潔になり得るかを示しています。

`numeric-letter-grades` テストの目的は、すべての値が適切な範囲に収まっているかどうかを確認することです。Lisp を使い慣れていないとしたら、コードを内側から外側の方向で読むことが、最も簡単にコードを理解する方法となります。まず、コード `#(is (= "A" (letter-grade %)))` が新しい匿名関数を作成します。この匿名関数は引数を 1 つだけ取り (引数を 1 つだけ取る

匿名関数では、その引数を関数本体の中で % としてすることが表現できます)、正しい文字等級が返される場合には true を返します。map 関数はこの匿名関数を、その 2 番目の引数のコレクションに対してマッピングします。このコレクションは、適切な範囲内にある数値のリストです。つまり、map はコレクションに含まれる項目ごとに匿名関数を呼び出し、変更後の値のコレクションを返します (ここではこの戻り値を無視しています)。dorun 関数は、副次効果が発生するのを容認し、テスト・フレームワークではこの副次効果を利用します。リスト 6 に示されている各 range で map を呼び出すと、すべて true の値からなるリストが返されます。clojure.test 名前空間の is メソッドは、副次効果としての値を検証します。dorun 内でマッピング関数を呼び出すことにより、この副次効果が適切に発生し、テスト結果が返されるというわけです。

Clojure のマルチメソッド

いくつもの if 文が長々と連なっていると、読んで理解するのも、デバッグするのも困難です。けれども Java には、言語レベルで特に優れた代替手段があるわけではありません。この問題を解決するには、一般に GoF による Factory デザイン・パターンか Abstract Factory デザイン・パターンが用いられます。Factory パターンは、Java では有効に機能します。これは、クラス・ベースのポリモーフィズムにより、親クラスまたはインターフェースに汎用メソッドのシグニチャーを定義した上で、実行する実装を動的に選択できるためです。

ファクトリーとポリモーフィズム

Groovy の構文は Java よりも簡潔で読みやすいため、これから紹介するいくつかのサンプル・コードでは、Java の代わりに Groovy を使用しますが、ポリモーフィズムはこの両方の言語で同じように機能します。まずは、リスト 7 に記載する、Product ファクトリーを定義するインターフェースとクラスの組み合わせを見てください。

リスト 7. Groovy で積のファクトリーを作成する

```
interface Product {
  public int evaluate(int op1, int op2)
}

class Multiply implements Product {
  @Override
  int evaluate(int op1, int op2) {
    op1 * op2
  }
}

class Incrementation implements Product {
  @Override
  int evaluate(int op1, int op2) {
    def sum = 0
    op2.times {
      sum += op1
    }
    sum
  }
}

class ProductFactory {
  static Product getProduct(int maxNumber) {
    if (maxNumber > 10000)
      return new Multiply()
    else
```

```
      return new Incrementation()
  }
}
```

リスト 7 では、2 つの数値の積を求める方法の動作を定義するインターフェースを作成し、このアルゴリズムを 2 つの異なるバージョンで実装しています。ファクトリーからどちらの実装を返すかに関するルールは、`ProductFactory` で決定します。

このファクトリーは、何らかの決定基準によって導き出される具体的な実装のための抽象的なプレースホルダーとして使用します。一例として、リスト 8 のコードを見てください。

リスト 8. 動的に実装を選択する

```
@Test
public void decisionTest() {
  def p = ProductFactory.getProduct(10010)
  assertTrue p.getClass() == Multiply.class
  assertEquals(2*10010, p.evaluate(2, 10010))
  p = ProductFactory.getProduct(9000)
  assertTrue p.getClass() == Incrementation.class
  assertEquals(3*3000, p.evaluate(3, 3000))
}
```

リスト 8 では `Product` 実装の 2 つのバージョンを作成して、正しいほうの実装がファクトリーから返されることを検証しています。

Java では、継承とポリモーフィズムは密接に関係する概念であり、ポリモーフィズムによってオブジェクトのクラスが作成されます。他の言語では、この 2 つの関係は緩くなっています。

任意の選択肢から選べる Clojure のポリモーフィズム

多くの開発者は、オブジェクト指向言語が権力の頂点であると信じ、Clojure がオブジェクト指向言語ではないという理由で、この言語を却下しています。けれども、それは間違いです。Clojure には、オブジェクト指向の言語に備わっているすべての機能が、他の機能とは独立して実装されています。例えば、Clojure はポリモーフィズムをサポートしていますが、それは、クラスを評価してディスパッチを決定することだけに限られません。Clojure でサポートされるポリモーフィックなマルチメソッドでは、開発者が望むあらゆる特性 (またはその組み合わせ) によってディスパッチがトリガーされます。

一例を紹介しましょう。Clojure では、データは通常、クラスのデータ部分を模倣した struct 内に格納されています。リスト 9 の Clojure コードを見てください。

リスト 9. Clojure において色を表すデータ構造を定義する

```
(defstruct color :red :green :blue)

(defn red [v]
  (struct color v 0 0))

(defn green [v]
  (struct color 0 v 0))

(defn blue [v]
  (struct color 0 0 v))
```

リスト 9 では、それぞれ色の値に対応する 3 つの値を格納するデータ構造を定義しました。また、単一の色で満たされた構造を返す 3 つのメソッドも作成しています。

Clojure でのマルチメソッドとは、ディスパッチ関数を受け入れて、決定基準を返すメソッド定義のことです。それに続く以降の定義によって、さまざまなバージョンのメソッドを具体化することができます。

リスト 10 にマルチメソッド定義の例を記載します。

リスト 10. マルチメソッドを定義する

```
(defn basic-colors-in [color]
  (for [[k v] color :when (not= v 0)] k))

(defmulti color-string basic-colors-in)

(defmethod color-string [:red] [color]
  (str "Red: " (:red color)))

(defmethod color-string [:green] [color]
  (str "Green: " (:green color)))

(defmethod color-string [:blue] [color]
  (str "Blue: " (:blue color)))

(defmethod color-string :default [color]
  (str "Red:" (:red color) ", Green: " (:green color) ", Blue: " (:blue color)))
```

リスト 10 で定義している `basic-colors-in` という名前のディスパッチ関数は、ゼロ以外のすべての色の値のベクトルを返します。メソッドのバリエーションとして、ディスパッチ関数が単一の色を返した場合の動作を指定します。この例での動作は、その色の文字列を返すことです。最後のケースには、オプションの `:default` キーワードを含めました。これが、残りのケースを処理します。この最後のケースについては、単一の色を受け取ったと見なすことはできないため、すべての色の値のリストを返します。

リスト 11 に、上記のマルチメソッドを演習するためのテストを記載します。

リスト 11. Clojure での色をテストする

```
(ns colors-test
  (:use clojure.test)
  (:use colors))

(deftest pure-colors
  (is (= "Red: 5" (color-string (struct color 5 0 0))))
  (is (= "Green: 12" (color-string (struct color 0 12 0))))
  (is (= "Blue: 40" (color-string (struct color 0 0 40)))))

(deftest varied-colors
  (is (= "Red:5, Green: 40, Blue: 6" (color-string (struct color 5 40 6)))))
```

リスト 11 では、単一の色を指定してメソッドを呼び出すと、単一色バージョンのマルチメソッドが実行されます。色の複合プロファイルを指定してメソッドを呼び出すと、デフォルトのメソッドがすべての色を返します。

ポリモーフィズムを継承から切り離すと、コンテキストに応じた強力なディスパッチ・メカニズムになります。例えば、それぞれに異なる特性のセットでタイプを定義する画像ファイルのフォーマットの問題を考えてみてください。Clojure ではディスパッチ関数を使用することで、Java のポリモーフィズムと同じくコンテキストに応じた (ただし制約は Java よりも少ない) 強力なディスパッチを実現することができます。

まとめ

この記事では、次世代の JVM 言語に登場するさまざまなディスパッチ・メカニズムを駆け足で紹介しました。ディスパッチが制限された言語で作業すると、デザイン・パターンのような外部の次善策によって、コードが煩雑になりがちです。今までに存在していなかった新しい言語での代替策を選択するのは、勇気が要ります。なぜなら、パラダイム・シフトが必要になるためです。けれども、関数型の考え方を習得する過程で、パラダイム・シフトを避けて通ることはできません。

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。