関数型の考え方: 関数型の観点で考える、第 3 回

フィルタリング、ユニット・テスト、そしてコードを再利用するための手法

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2011年 7月 29日

連載「関数型の考え方」の今回の記事では、著者の Neal Ford が引き続き関数型プログラミングの構成体とパラダイムについて説明します。まずは Scala で作成した数値分類子のコードを検討し、関数型の世界でのユニット・テストに簡単に触れた後、コードの再利用を容易にする関数型の 2 つの手法、部分適用とカリー化について説明します。そして最後に、再帰が関数型の考え方にいかに適合しているかを明らかにします。

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」の第 1 回と第 2 回では、関数型プログラミングについてのトピックをいくつか取り上げ、Java およびその関連言語との関係を詳しく調べました。今回の記事でもこのツアーを引き継ぎ、これまでの記事で扱ってきた数値分類子を今度は Scala で作成して、カリー化、部分適用、そして再帰などの学究的な色合いを持つトピックを議論します。

Scala での数値分類子

数値分類子の Scala バージョンを最後まで取っておいた理由は、少なくとも Java 開発者にとって、このバージョンが最も構文的に理解しやすいからです (数値分類子の要件を復習しておくと、1 より大きい正の整数は完全数、過剰数、または不足数のいずれかとして分類しなければなりません。完全数とは、その数の約数 (ただし、その数自身を除く) の和がその数自身と等しくなる数のことです。過剰数とはその数の約数の和がその数自身よりも大きくなる数のことで、不足数とはその数の約数 (ただし、その数自身を除く) の和がその数より小さくなる数のことです)。リスト 1 に、Scala で作成したバージョンを記載します。

リスト 1. Scala での数値分類子

```
package com.nealford.conf.ft.numberclassifier

object NumberClassifier {

  def isFactor(number: Int, potentialFactor: Int) =
    number % potentialFactor == 0

  def factors(number: Int) =
    (1 to number) filter (number % _ == 0)

  def sum(factors: Seq[Int]) =
    factors.foldLeft(0)(_ + _)

  def isPerfect(number: Int) =
    sum(factors(number)) - number == number

  def isAbundant(number: Int) =
    sum(factors(number)) - number > number

  def isDeficient(number: Int) =
    sum(factors(number)) - number < number
}
```

Scala を初めて目にするとしても、このコードはすんなり理解できるでしょう。これまでのバージョンと同じく、このバージョンでも興味深いメソッドは、`factors()` と `sum()` です。`factors()` メソッドは、1 から対象の数値までの数値のリストを取得し、そのメソッドに Scala の組み込み `filter()` メソッドを適用します。その際、フィルタリング基準 (つまり、述部のこと) として使用されるメソッド右側のコード・ブロックは、Scala の暗黙パラメーターを利用しています。暗黙パラメーターでは、名前付き変数が必要ない場合、無名のプレースホルダー (`_` 文字) を使用することができます。Scala の構文には柔軟性があるおかげで、演算子を呼び出す場合と同じ方法で `filter()` を呼び出すことができます。お望みであれば、代わりに `(1 to number).filter((number % _ == 0))` を使ってもコードは機能します。

`sum()` メソッドが使用するのは、今ではお馴染みの「`fold left`」操作です (Scala では、`foldLeft()` メソッドとして実装されています)。この例では変数に名前を付ける必要はないため、`_` をプレースホルダーとして使用することによって、単純で簡潔な構文でコード・ブロックを定義しています。この `foldLeft()` メソッドは、連載の第 1 回で取り上げた Functional Java ライブラリー (「参考文献」を参照) に用意されている同じ名前のメソッドと同じく、以下のタスクを実行します。

1. 初期値を取得し、演算処理によってその値とリストの最初の要素を加算します。
2. その結果を取得し、同じ演算処理をリストの次の要素に適用します。
3. これをリストの最後に達するまで繰り返します。

これは、加算などの演算を数値のリストに対して適用する方法を一般化したバージョンです。つまり、ゼロから初めて、最初の要素をそこに加算し、その結果を取得して次の要素に加算するという処理を、リストの最後の要素まで続けます。

ユニット・テスト

これまでに記載したバージョンにはユニット・テストについては記載していませんでしたが、すべてのサンプル・コードではテストが行われます。Scala の場合、ユニット・テストには ScalaTest

という効果的なライブラリーを利用することができます (「参考文献」を参照)。リスト 2 に、リスト 1 の `isPerfect()` メソッドを検証するために私が作成した最初のユニット・テストを記載します。

リスト 2. Scala の数値分類子を対象としたユニット・テスト

```
@Test def negative_perfection() {
  for (i <- 1 until 10000)
    if (Set(6, 28, 496, 8128).contains(i))
      assertTrue(NumberClassifier.isPerfect(i))
    else
      assertFalse(NumberClassifier.isPerfect(i))
}
```

上記のユニット・テストを作成してはみたものの、読者の皆さんと同じように、関数型により即した考え方を学ぼうとしている私にとって、リスト 2 のコードには気に入らない点が 2 つありました。1 つは、何らかの処理を行うために繰り返しの処理を行っていることです。これは、命令型の考え方を示しています。もう 1 つ気に入らない点は、このコードでは `if` 文による true/false の処理のキャッチオールが考慮されていないことです。ここで解決しようとしている問題は何であるかを考えると、数値分類子が完全数ではない数値を完全数として識別しないようにすることを確実にする必要があります。そこで、多少表現を変えた、この問題に対するソリューションをリスト 3 に記載します。

リスト 3. 完全数の分類を対象とした代替テスト

```
@Test def alternate_perfection() {
  assertEquals(List(6, 28, 496, 8128),
               (1 until 10000) filter (NumberClassifier.isPerfect(_)))
}
```

リスト 3 でアサートしている内容は、1 から 100,000 までの範囲にあり、完全数である数値だけが、既知の完全数のリストに含まれるというものです。このように、関数型の考え方は、コードだけでなく、そのコードのテストについての考え方にも及びます。

部分適用とカリー化

上記のリストをフィルタリングする際の関数型の手法は、関数型プログラミングの言語およびライブラリーで共通する手法です。(リスト 3 の `filter()` に関して) コードをパラメーターとして渡す機能を使用するということは、コードの再利用についての別の考え方を説明しています。従来のデザインパターンによるオブジェクト指向プログラミングに慣れているとしたら、この手法を Gang of Four の著書『オブジェクト指向における再利用のためのデザインパターン』(「参考文献」を参照) で説明されている Template Method (テンプレート・メソッド) デザインパターンと比較してください。Template Method パターンは、基底クラスにアルゴリズムのスケルトンを定義し、アルゴリズムの詳細については抽象メソッドを使用してオーバーライドすることで、子クラスのそれぞれに任せます。関数型の手法では、合成を使用することによって、機能を複数のメソッドに渡し、各メソッドに応じてその機能を適用させることができます。

カリー化も、コードの再利用を可能にする手法です。数学者の Haskell Curry (Haskell プログラミング言語の名前の由来でもあります) にちなんで名付けられた、このカリー化という手法は、複数の引数を取る関数を変換して、1 つの引数のみを取る関数のチェーンとして呼び出せるようにします。カリー化に密接に関係する手法には、部分適用があります。部分適用は、ある関数の 1 つ

または複数の引数に 1 つの固定値を割り当てることで、アリティー (関数に渡されるパラメーターの数) が小さい別の関数を生成するという手法です。この 2 つの違いを理解するには、まず、カリー化の一例としてリスト 4 の Groovy コードを見てください。

リスト 4. Groovy のカリー化

```
def product = { x, y -> return x * y }

def quadrate = product.curry(4)
def octate = product.curry(8)

println "4x4: ${quadrate.call(4)}"
println "5x8: ${octate(5)}"
```

リスト 4 ではまず、2 つのパラメーターを受け付けるコード・ブロックとして `product` を定義します。次に、Groovy の組み込み `curry()` メソッドを使用して、`product` を 2 つの新しいコード・ブロック、`quadrate` と `octate` のビルディング・ブロックとして使用します。Groovy では、コード・ブロックを簡単に呼び出せるようになっていて、明示的に `call()` メソッドを実行するか、あるいは提供されている言語レベルの構文糖を使用して、コード・ブロック名の後に括弧で囲んだパラメーター (例えば `octate(5)`) を配置すればよいのです。

部分適用はカリー化と似ていますが、結果として生成される関数が取る引数の数を 1 つに制限していないという点でカリー化の範囲をさらに広げた手法です。Groovy ではカリー化と部分適用を処理するために共通して `curry()` メソッドを使用します (リスト 5 を参照)。

リスト 5. Groovy の `curry()` メソッドを共通して使用する部分適用とカリー化の比較

```
def volume = { h, w, l -> return h * w * l }
def area = volume.curry(1)
def lengthPA = volume.curry(1, 1) //partial application
def lengthC = volume.curry(1).curry(1) // currying

println "The volume of the 2x3x4 rectangular solid is ${volume(2, 3, 4)}"
println "The area of the 3x4 rectangle is ${area(3, 4)}"
println "The length of the 6 line is ${lengthPA(6)}"
println "The length of the 6 line via curried function is ${lengthC(6)}"
```

リスト 5 の `volume` コード・ブロックは、直方体の体積をお馴染みの公式で計算します。次に作成している `area` コード・ブロック (直方体の面積を計算) では、`volume` の最初の寸法 (高さの `h`) の値を `1` に固定しています。volume を、線分の長さを返すコード・ブロックのビルディング・ブロックとして使用するには、部分適用またはカリー化のいずれかを適用することができます。`lengthPA` では、最初の 2 つのパラメーターをそれぞれ `1` に固定して部分適用を使用している一方、`lengthC` では同じ結果を出すためにカリー化を 2 回適用しています。この 2 つの違いは微妙なもので、しかも最終結果は同じですが、関数型プログラマーが聞いているところでカリー化と部分的適用という言葉を同じ意味で使用すると、訂正されることは間違いありません。

関数型プログラミングでは、命令型言語で使用するメカニズムとは異なる新たなビルディング・ブロックによって、命令型言語と同じ目的を果たすことができます。これらのビルディング・ブロック相互の関係は綿密に考案されています。前に、コードを再利用するためのメカニズムとして合成を紹介しましたが、当然、カリー化と合成を組み合わせて使用することもできます。リスト 6 の Groovy コードを見てください。

リスト 6. 部分適用の合成

```
def composite = { f, g, x -> return f(g(x)) }
def thirtyTwoer = composite.curry(quadrate, octate)

println "composition of curried functions yields ${thirtyTwoer(2)}"
```

リスト 6 では、2 つの関数を合成する `composite` コード・ブロックを作成した後、このコード・ブロックを使用して `thirtyTwoer` コード・ブロックを作成し、部分適用を使用して 2 つのメソッドを合成しています。

このように、部分適用とカリー化を使用することで、Template Method デザインパターンのようなメカニズムと同様の目的を果たします。例えばリスト 7 に示すように、`adder` コード・ブロックをベースにすることによって、`incrementer` コード・ブロックを作成することができます。

リスト 7. 異なるビルディング・ブロック

```
def adder = { x, y -> return x + y }
def incrementer = adder.curry(1)

println "increment 7: ${incrementer(7)}"
```

もちろん、Scala はカリー化をサポートします。これは、リスト 8 に記載する Scala マニュアルから抜粋したスニペットを見れば一目瞭然です。

リスト 8. Scala のカリー化

```
object CurryTest extends Application {

  def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

  def dividesBy(n: Int)(x: Int) = ((x % n) == 0)

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
  println(filter(nums, dividesBy(2)))
  println(filter(nums, dividesBy(3)))
}
```

リスト 8 のコードには、`filter()` メソッドで使用する `dividesBy()` メソッドの実装方法が示されています。カリー化を使用して、`dividesBy()` の最初のパラメーターをコード・ブロックの作成に使用する値に固定し、`filter()` メソッドに匿名関数を渡します。対象の数値をパラメーターとして渡して作成されたコード・ブロックを渡すと、Scala はカリー化によって新しい関数を生成します。

再帰によるフィルタリング

関数型プログラミングに密接に関連するトピックには、再帰もあります。再帰とは、(ウィキペディアによると)「あるものについて記述する際に、記述しているもの自身への参照が、その記述中に現れること」です。実際には、これは (常に、慎重かつ確実に終了条件を用意して) メソッド自体から同じメソッドを呼び出して処理を繰り返すという、コンピューター・サイエンス的な方法です。再帰によって理解しやすいコードになることは多々ありますが、それは問題の核心が、

リストの終わりに達するまで同じことを何度も繰り返さなければならないということにあるからです。

リストをフィルタリングする場合を考えてください。その場合、私は繰り返しの手法を適用し、フィルタリング基準を受け入れてコンテンツをループで処理し、不要な要素をフィルタリング処理で除外します。リスト 9 に、Groovy による単純なフィルタリングの実装を記載します。

リスト 9. Groovy のフィルタリング

```
def filter(list, criteria) {
  def new_list = []
  list.each { i ->
    if (criteria(i))
      new_list << i
  }
  return new_list
}

modBy2 = { n -> n % 2 == 0 }

l = filter(1..20, modBy2)
println l
```

リスト 9 の `filter()` メソッドは、`list` と `criteria` (リストのフィルタリング方法を指定するコード・ブロック) をパラメーターとして受け取り、そのリストを繰り返し処理し、リストの項目が述部の条件を満たす場合には、その項目を新しいリストに追加します。

ここで、リスト 8 をもう一度見てください。このコードは、Scala でフィルタリング機能を再帰によって実装したものです。この実装は、関数型言語でリストを扱う際の共通パターンに従っています。リストの 1 つの見方は、リストはその最前列 (head) にある項目と、残りすべての項目で構成されているというものです。多くの関数型言語には、このイディオムを使用してリストを繰り返し処理するための固有のメソッドがあります。`filter()` メソッドの場合、最初にリストが空であるかどうかを確認します。これは、このメソッドにとって極めて重要な終了条件です。リストが空であれば、メソッドは単にリターンするだけです。空でなければ、パラメーターとして渡された述部の条件 (p) を使用します。この条件が true の場合 (つまり、この項目をリストに含めたい場合)、現在の head を取得し、リストの残りの項目をフィルタリングして作成した新しいリストを返します。述部の条件を満たさない場合、フィルタリングされた残りの項目だけからなる新しいリストを返します (最初の要素は除外されたリスト)。この Scala のリスト作成演算子によって、両方の場合のリターン条件が極めて読みやすく、理解しやすいものになります。

おそらく、読者の皆さんは今のところ再帰をまったく使用していないのではないでしょうか。皆さんのツール・ボックスの一部にすらなっていないかもしれません。けれども、その理由の一端は、ほとんどの命令型言語では中途半端にしか再帰をサポートしていないことから、再帰の使用を必要以上に難しくしているところにあります。関数型言語では簡潔な構文とサポートを追加することで、再帰をコードの再利用の簡単な手段の 1 つにしています。

まとめ

今回の記事では引き続き、関数型の観点で考える世界での特徴を詳しく探りました。記事の大半では、フィルタリングのさまざまな使用方法と実装方法を説明することになりましたが、これは

それほど驚くべきことではありません。関数型プログラミングを突き詰めて行くと、その大半はリストを扱うことに行きつきます。したがって、多くの関数型パラダイムはリストを中心に成り立っています。このことから、関数型の言語やフレームワークに、リストを対象とした機能が有り余るほど用意されているのは頷けます。

次回の記事では、関数型プログラミングのパラダイムを探るツアーを締めくくります。

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。