関数型の考え方: 実にさまざまな変換処理

同義のメソッドに隠された類似点

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2012年 10月 25日

関数型プログラミングの構成体は、今やあらゆる主要言語で使われるようになっていますが、これらの構成体を見分けるのは容易ではありません。それは、これらの構成体には一般的な名前が幅広く使われているためです。連載「関数型の考え方」の今回の記事では、異なる 7 種類の関数型フレームワークおよび言語で同じ例に対するコードを作成し、それぞれの類似点と相違点を探ります。

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。

関数型プログラミング言語でコードを再利用する方法がオブジェクト指向言語での方法とは異なることは、「連結と合成、第 2 回」でのトピックとして説明しました。オブジェクト指向言語では多くのデータ構造にさまざまな処理が含まれますが、関数型言語にはそのようなデータ構造はほとんどありません。オブジェクト指向言語は、クラス固有のメソッドを作成するよう促します。この場合、繰り返し出現するパターンを取り込めば、そのパターンを再利用することができます。一方、関数型言語が再利用を実現するために促している方法は、データ構造に対して一般的な変換処理を適用し、個々のケースに合わせて処理をカスタマイズするために高階関数を使用するというものです。

すべての関数型言語で (そして Java において関数型プログラミングをサポートするフレームワークの多くで) 共通して使用されているデータ構造や処理はありますが、同じ名前で使用されていることはあまりありません。混乱を招くネーミングにより、基礎となる概念は同じであっても、ある言語での知識を他の言語に適用することが難しくなっています。

今回の記事の目標は、こうしたある言語での知識を他の言語に適用するのを容易にすることです。判定と繰り返しを必要とする単純な問題を例に、その問題のソリューションを 5 つの言語 (Java、Groovy、Clojure、JRuby、Scala) と Java 対応の 2 つの関数型フレームワーク (Functional Java および Totally Lazy) で実装します (「参考文献」を参照)。実装する対象はどれも同じですが、実装方法の詳細は言語によってかなり異なります。

従来の Java

例で取り上げる問題は、整数が素数 (1 とその値自身以外に約数を持たない整数) であるかどうかを判別するという問題です。素数を判別するためのアルゴリズムはいくつかあります (いくつかのソリューションを「連結と合成、第 1 回」で紹介しています)。ここでは、数値の約数を明らかにしてから、約数の和がその数値に 1 を足した値と等しくなる (この場合、その数値は素数です) かどうかを調べるというソリューションを使用します。最も効率的なアルゴリズムであるとは言えませんが、私が目標としているのは効率性ではなく、コレクションの一般的なメソッドの異なる実装を示すことです。

リスト 1 に、従来の Java による実装を記載します。

リスト 1. 従来の Java による素数分類子

```
public class PrimeNumberClassifier {
    private Integer number;

    public PrimeNumberClassifier(int number) {
        this.number = number;
    }

    public boolean isFactor(int potential) {
        return number % potential == 0;
    }

    public Set<Integer> getFactors() {
        Set<Integer> factors = new HashSet<Integer>();
        factors.add(1);
        factors.add(number);
        for (Integer i = 2; i < number; i++)
            if (isFactor(i))
                factors.add(i);
        return factors;
    }

    public int sumFactors() {
        int sum = 0;
        for (int i : getFactors())
            sum += i;
        return sum;
    }

    public boolean isPrime() {
        return sumFactors() == number + 1;
    }
}
```

「関数型の考え方」の以前の記事を読んだことがあれば、リスト 1 の `getFactors()` メソッドで使われているアルゴリズムには見覚えがあることでしょう。このアルゴリズムの中核は、約数の候補を見つけて抽出する `isFactor()` メソッドです。

Groovy

Groovy はその進化の過程で関数型の構成体を数多く追加してきたことから、Groovy による実装 (リスト 2 に記載) は、Java による実装とはかなり違って見えます。

リスト 2. Groovy による素数分類子

```
class PrimeNumberClassifier {
  private int number;

  PrimeNumberClassifier(int number) {
    this.number = number
  }

  public def isFactor(int potential) {
    number % potential == 0;
  }

  public def getFactors() {
    (1..number).findAll { isFactor(it) }.toSet()
  }

  public def sumFactors() {
    getFactors().inject(0, {i, j -> i + j})
  }

  public def isPrime() {
    sumFactors() == number + 1
  }
}
```

リスト 1 のメソッドに相当するリスト 2 の 2 つのメソッドは、単にその構文が変更されているだけではありません。まず `getFactors()` は、Groovy の `Range` クラスを使用して約数の候補を表します。`findAll()` メソッドはこのコレクションの各要素にコード・ブロックを適用し、そのコード・ブロックから `true` が返された項目で構成されるリストを返します。このコード・ブロックが取る唯一の引数は、検査の対象となる要素です。ここでは Groovy の便利な省略形を使用して、コード・ブロックをさらに簡潔にしました。例えば、このコード·ブロックは、`(1..number).findAdd {i-> isFactor(i) }` とすることもできますが、同じ引数を繰り返すのは冗長です。リスト 2 に示されているように、Groovy では、長い引数を暗黙の `it` に置き換えるオプションを使用することができます。

リスト 2 で注目に値するもう 1 つのメソッドは、`sumFactors()` メソッドです。`getFactors()` メソッドによって一連の数値を生成した後、この数値セットを使用して、`inject()` メソッドを呼び出します。このメソッドは、畳み込み処理を行う Groovy のコレクションのメソッドで、1 番目の引数を初期シード値として使用し、2 番目の引数に指定されたコード・ブロックを使用して、コレクションの各要素を合成していきます。リスト 2 でコード・ブロック引数として指定されている `{i, j-> i + j}` は、2 つの数値の和を返します。`inject()` メソッドは最初の要素から順番に各要素に対してこのブロックを適用して、リストの数値の和を計算します。

関数型メソッドを高階関数と組み合わせて使用すると、難解なコードになる場合があります。リスト 2 のそれぞれのメソッドでは、処理が 1 行で記述されていますが、個々のメソッドに分解したほうが有益です。メソッドを機能ごとに分解して、現行の問題のコンテキストに沿った意味のある名前を付けることで、論理的に理解しやすくなります。

Scala

リスト 3 に、Scala による素数分類子を記載します。

リスト 3. Scala による素数分類子

```
object PrimeNumberClassifier {
  def isFactor(number: Int, potentialFactor: Int) =
    number % potentialFactor == 0

  def factors(number: Int) =
    (1 to number) filter (isFactor(number, _))

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

  def isPrime(number: Int) =
    sum(factors(number)) == number + 1
}
```

Scala による実装は、大幅に簡潔になっているだけでなく、他にも多くの点で異なっています。必要なインスタンスは 1 つだけなので、この実装では素数分類子を `class` ではなく、`object` にしました。`factors()` メソッドが使用している実装は、リスト 2 の Groovy による実装と同じですが、その構文は異なります。具体的に言うと、`filter()` メソッド (Groovy の `findAll()` に相当する Scala のメソッド) を数値の範囲 `(1 to number)` に適用します。このメソッドの述部として使用しているのは、リスト 3 の最初に定義されている `isFactor()` メソッドです。Scala では、引数のプレースホルダーも使用することができます。上記の例でそれに該当するのは、「`_`」です。

リスト 3 の `sum()` メソッドでは、Scala の `foldLeft()` メソッドを使用しています。このメソッドは Groovy の `inject()` と同義です。この例では、ゼロをシード値として使用し、両方の引数にプレースホルダーを使用しました。

Clojure

JVM での最近の Lisp 実装である Clojure は、リスト 4 のように顕著に異なる構文となっています。

リスト 4. Clojure による素数分類子

```
(ns prime)

(defn factor? [n, potential]
  (zero? (rem n potential)))

(defn factors [n]
  (filter #(factor? n %) (range 1 (+ n 1))))

(defn sum-factors [n]
  (reduce + (factors n)))

(defn prime? [n]
  (= (inc n) (sum-factors n)))
```

リスト 4 で使用されているすべてのメソッドは、Java 開発者にとって異質なものに見えますが、このコードが実装するアルゴリズムは、これまでに使用してきたアルゴリズムと同じです。`(factor?)` メソッドは、除算の剰余 (Clojure での `rem` 関数の値) がゼロであるかどうかを調べます。`(factors)` メソッドが使用する Clojure の `(filter)` メソッドは、2 つの引数を取りま

す。1 番目の引数は、述部コード・ブロックです。このブロックを各要素で実行して、その要素がフィルター基準を満たすかどうかを示すブール値の結果を要求します。Clojure の無名関数を表す `#(factor? n %)` 構文では、Clojure の引数で置き換わる `%` が使用されています。`(filter)` 関数の 2 番目の引数は、フィルタリング対象のコレクションです。この例の場合、1 から対象の整数に 1 を足した数値までの範囲をフィルタリングします。範囲には、最後の要素は含まれません。

リスト 4 の `(sum-factors)` メソッドで使用している Clojure の `(reduce)` メソッドは、Groovy の `inject()` および Scala の `foldLeft()` と同義です。上記の例で行っている処理は単純な `+` 演算子による演算であり、Clojure にとって、これは 2 つの引数を取って結果を返す他のあらゆるメソッドとまったく変わりがありません。

慣れるまでは手強い構文かもしれませんが、Clojure による実装は極めて簡潔になります。Groovy による実装の場合と同じく、わずか 1 行の関数であっても、それぞれの関数の行にはかなりの意味が含まれていることがあるため、関数にはその内容を判断できるような名前を付けることが重要です。

JRuby

Ruby の JVM 実装である JRuby には、誕生して以来、同じく多数の関数型の構成体が追加されてきました。リスト 5 に記載する、(J)Ruby による素数分類子を見てください。

リスト 5. JRuby による素数分類子

```
class PrimeNumberClassifier
  def initialize(num)
    @num = num
  end

  def factor?(potential)
    @num % potential == 0
  end

  def factors
    (1..@num).select { |i| factor?(i) }
  end

  def sum_factors
    factors.reduce(0, :+)
  end

  def prime?
    (sum_factors == @num + 1)
  end
end
```

リスト 5 の `factors` メソッドでは、Groovy の `findAll` とも、Scala および Clojure の `filter` メソッドとも同義の `select` メソッドを追加しています。JRuby の便利な機能の 1 つは、メソッドに簡単にエイリアスを付けられる機能であり、さまざまなコンテキストで都合の良い名前でメソッドを使用できることです。案の定、JRuby には `select` メソッドに対して `find_all` というエイリアスがありますが、このエイリアスはイディオムとしてそれほど一般的には使われていません。

リスト 5 の `sum_factors` メソッドには、他のいくつかの言語を真似て、JRuby の `reduce` メソッドを使用しました。Clojure と同じく、JRuby の演算子は奇妙な名前が付けられたメソッドです。Ruby では、加算のためのメソッドを、プラス記号を使って「`:+`」で指定することができま

す。読みやすくするために、Clojure と Ruby ではどちらも、ブール値を返させる関数には疑問符を追加することができます。さらに Ruby には、その性質に合うように、`reduce` メソッドのエイリアスとして `inject` メソッドがあります。

Functional Java

いくつかの関数型プログラミング・ライブラリーでは、Java の変形を今でも使用している人々にも対応するために、関数型の構成体によって Java を増補しています。Functional Java は、そのようなフレームワークの 1 つです。リスト 6 に、Java と Functional Java を使用して作成した素数分類子を記載します。

リスト 6. Functional Java による素数分類子

```
public class FjPrimeNumberClassifier {
    private int number;

    public FjPrimeNumberClassifier(int number) {
        this.number = number;
    }

    public boolean isFactor(int potential) {
        return number % potential == 0;
    }

    public List<Integer> getFactors() {
        return range(1, number + 1)
                .filter(new F<Integer, Boolean>() {
                    public Boolean f(final Integer i) {
                        return isFactor(i);
                    }
                });
    }

    public int sumFactors() {
        return getFactors().foldLeft(fj.function.Integers.add, 0);
    }

    public boolean isPrime() {
        return sumFactors() == number + 1;
    }
}
```

リスト 6 の `getFactors()` メソッドは、`range` で `filter()` メソッドを使用します (Clojure と同じく、この `range` でも最後の値は含まれません。したがって、範囲の定義は `number + 1` までとなっています)。Java にはまだ高階関数がないことから、Functional Java はその組み込み `F` クラスの匿名内部クラス・インスタンスを代用し、Generics を使って型を引数にします。

Scala と同じように、Functional Java には `foldLeft()` メソッドがあり、このメソッドで (この例の場合は) 数値とシード値の和を計算するためにあらかじめ定義されたコード・ブロックを受け入れます。

Totally Lazy

Totally Lazy は、多くの「遅延 (lazy)」コレクションを Java に追加する、Java 用の関数型ライブラリーです。遅延データの構造は要素を事前に定義するのではなく、次の値が要求されたときに、その値をどのように生成するかに関するルールをエンコードします。Totally Lazy で実装した素数分類子は、リスト 7 のようになります。

リスト 7. Totally Lazy による素数分類子

```
public class TlPrimeNumberClassifier {
    private int number;

    public TlPrimeNumberClassifier(int number) {
        this.number = number;
    }

    public boolean isFactor(Integer potential) {
        return (number % potential) == 0;
    }

    private List<Integer> listOfPotentialFactors() {
        List<Integer> results = new ArrayList<Integer>();
        for (int i = 1; i <= number + 1; i++)
            results.add(i);
        return results;
    }

    public boolean isPrime() {
        return (this.number + 1) ==
            Sequences.init(listOfPotentialFactors())
                     .filter(new Predicate<Integer>() {
                         @Override
                         public boolean matches(Integer other) {
                             return isFactor(other);
                      }})
                     .foldLeft(0, sum())
                     .intValue();
    }
}
```

リスト 7 の `isPrime()` メソッドは、`Sequences` クラスをすべての約数の候補 (つまり、1 から対象の数値までのすべての数値) のリストで初期化した後、このクラスの `filter()` メソッドを適用します。Totally Lazy では、この `filter()` メソッドが必要とするのは、`Predicate` クラスのサブクラスであり、`Predicate` クラスの多くの部分は一般的なケースを対象に実装済みです。上記の例では、`matches()` メソッドをオーバーライドし、約数のリストに含めるかどうかを判別するために `isFactor()` メソッドを指定しました。約数のリストが出来上がったら、`foldLeft` メソッドを使用して、このメソッドに指定した `sum()` メソッドで畳み込み操作を行います。

リスト 7 に記載した例では、大変な処理の大部分を `isPrime()` メソッドが行います。Totally Lazy のデータ構造は、いずれも遅延が関与することから、データ構造を組み合わせると複雑さが増すことがよくあります。例えば、リスト 8 に記載する `getFactors()` メソッドを見てください。

リスト 8. 遅延イテレーターを使用した Totally Lazy の `getFactors` メソッド

```
public Iterator<Integer> getFactors() {
    return Sequences
            .init(listOfPotentialFactors())
            .filter(new Predicate<Integer>() {
                @Override
                public boolean matches(Integer other) {
                    return isFactor(other);
                }
            })
            .iterator();
}
```

リスト 8 の `getFactors()` メソッドからの戻り型は、`Iterator<Integer>` です。けれども、これは遅延イテレーターであるため、コレクションを繰り返し処理するまで、コレクションには値がありません。このことから、遅延コレクションをテストするのが困難になっています。リスト 8 の Totally Lazy のサンプル・コードを対象にしたユニット・テストを見てください (リスト 9 を参照)。

リスト 9. Totally Lazy のコレクションのテスト

```
@Test
public void factors() {
    TlPrimeNumberClassifier pnc = new TlPrimeNumberClassifier(6);
    List<Integer> expected = new ArrayList<Integer>() {{
        add(1);
        add(2);
        add(3);
        add(6);
    }};
    Iterator<Integer> actual = pnc.getFactors();
    assertTrue(actual.hasNext());
    int count = 0;
    for (int i = 0; actual.hasNext(); i++) {
        assertEquals(expected.get(i), actual.next());
        count++;
    }
    assertTrue(count == expected.size());
}
```

遅延コレクションでは、コレクションをウォークスルーして値を取得した後に、遅延リストに想定よりも多くの要素が含まれていないことを確実にする必要があります。

他の言語やフレームワークと同じような優れた約数抽出方法のコードを Totally Lazy で作成することも可能ですが、その場合、例えば `<Iterator<Iterator<Number>>>` といったように、さらに複雑なデータ構造と格闘する結果になりかねません。

まとめ

今回の記事では、各種の関数型の言語とフレームワークで同じ振る舞いに対して使用されている名前の違いを解き明かしました。これらの言語とフレームワークでメソッド名が統一されることは決してないでしょう。けれども、Java ランタイムがクロージャーなどの構成体を追加するにつれ、相互運用は容易になってくるはずです。なぜなら、(例えば Totally Lazy での `<Iterator<Iterator<Number>>>` のような厄介な構成体が必要になる代わりに) 言語やフレームワーク間で共通の標準表現を共有できるようになるためです。

次回の記事では、`map` について調べることで、さまざまな関数型の言語やフレームワークでの変換処理の類似点を引き続き調査します。

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。