関数型の考え方: 関数型プログラミングの人気が高まっている理由

使用する言語を今すぐ変える気がないとしても、関数型プログラミングに関心を持つべき理由とは

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2013年 8月 01日

当面、Scala や Clojure などの関数型言語に移行する気はないとしても、Java 開発者は関数型のパラダイムを今すぐ学んでおくべきです。主流のプログラミング言語はいずれ、どれもが関数型に近いものになることでしょう。今回の記事で、Neal Ford がその理由を探ります。

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。

この連載ではこれまで毎回、関数型プログラミングを理解することが重要である理由を説明してきました。けれどもそうした理由のなかには、説明に何回かの記事を要し、複数の概念を組み合わせた大局的なコンテキストのなかで初めて完全に明らかになるものもあります。今回の記事では、これまでの記事で学んだ個々の話題を織り交ぜながら、関数型プログラミングが優勢になってきている理由のすべてを探ります。

コンピューター・サイエンスの短い歴史のなかで、主流のテクノロジーから実用的な分岐や学究的な分岐が生じることは時折ありました。1990年代の 4GL (第 4 世代言語) は実用的な分岐の一例であり、関数型プログラミングは学究的な分岐の一例です。そのような分岐は主流に加わることもあります。これが、関数型プログラミングに現在起こっていることです。関数型言語としては JVM 上で実行されるものが急成長しています。なかでも最も興味深い 2 つの新しい言語が Scala と Clojure ですが、関数型言語が急成長しているのは JVM に限った話ではありません。F# が第一級市民となっている .NET プラットフォームでも関数型言語は急成長しています。なぜこのよう

に、関数型プログラミングがあらゆるプラットフォームで採用されているのでしょうか？その答えは、時間のかかる、ありふれた処理を、ランタイムがより多く扱えるようになるにつれ、開発者はそのような処理の制御をランタイムに委ねられるようになってきたという事実にあります。

制御の委譲

1980年代初め、私が大学生だった頃、私達は Pecan Pascal と呼ばれる開発環境を使用していました。この環境にはユニークな機能があり、同じ Pascal コードを Apple II 上でも IBM PC 上でも実行することができました。Pecan の技術者達は「バイト・コード」と呼ばれる謎めいたものを使用することにより、この機能を実現していました。開発者は作成した Pascal コードを「バイト・コード」にコンパイルし、その「バイト・コード」をそれぞれのプラットフォーム用にネイティブに作成された「仮想マシン」上で実行しました。しかしそれは散々な体験でした！単純なクラスの割り当ての場合でさえ、コードは非常に低速で実行されました。当時のハードウェアは、とてもそうした難題に対処できなかったのです。

Pecan Pascal から 10 年後、Sun が Pecan Pascal と同じアーキテクチャーを使用した Java をリリースしました。それは非常に負荷が重いものでしたが、1990年代半ばのハードウェア環境では成功を収めました。また Java には自動ガーベッジ・コレクションなど、開発者にとって使い勝手のよい機能も追加されていました。C++ のような言語を扱ってきた経験から、私はもう一度ガーベッジ・コレクション機能のない言語でコーディングしたいとは決して思いません。私は、メモリー管理のような下位レベルの複雑な問題を解決する方法の検討に時間をかけるのではなく、上位レベルで抽象化することによってビジネス上の複雑な問題を解決する方法を検討することに時間をかけたいと思います。

Java ではメモリー管理の扱いが容易になりましたが、関数型プログラミング言語では他の中核的なビルディング・ブロックをさらに高い次元の抽象化に置き換えること、そして過程よりも結果にフォーカスすることを可能にします。

過程よりも結果

関数型プログラミングの特徴の 1 つは、繰り返し処理などのありふれた処理に関する詳細の多くを開発者から隠す、強力な抽象化が存在することです。この連載を通して使用している数値分類子の例では、ある数値が完全数、過剰数、または不足数のいずれであるかを調べます (完全な定義については、連載第 1 回に相当する記事を参照してください)。この問題を解決するための Java 実装をリスト 1 に記載します。

リスト 1. 合計をキャッシュする Java の数値分類子

```
import static java.lang.Math.sqrt;

public class ImpNumberClassifier {
    private Set<Integer> _factors;
    private int _number;
    private int _sum;

    public ImpNumberClassifier(int number) {
        _number = number;
        _factors = new HashSet<Integer>();
        _factors.add(1);
        _factors.add(_number);
        _sum = 0;
```

```
    }

    private boolean isFactor(int factor) {
        return _number % factor == 0;
    }

    private void calculateFactors() {
        for (int i = 1; i <= sqrt(_number) + 1; i++)
            if (isFactor(i))
                addFactor(i);
    }

    private void addFactor(int factor) {
        _factors.add(factor);
        _factors.add(_number / factor);
    }

    private void sumFactors() {
        calculateFactors();
        for (int i : _factors)
            _sum += i;
    }

    private int getSum() {
        if (_sum == 0)
            sumFactors();
        return _sum;
    }

    public boolean isPerfect() {
        return getSum() - _number == _number;
    }

    public boolean isAbundant() {
        return getSum() - _number > _number;
    }

    public boolean isDeficient() {
        return getSum() - _number < _number;
    }
}
```

リスト 1 のコードは、繰り返し処理によって約数を判別し、約数であれば合計する典型的な Java コードです。関数型プログラミング言語の場合、開発者は処理の詳細 (例えば (`calculateFactors()` で行われる) 繰り返し処理や、(`sumFactors()` で行われる) リストの値を合計するなどの変換処理) を高階関数および粒度の粗い抽象化に委ねるため、それらの詳細についてはあまり気に掛ける必要がありません。

粒度の粗い抽象化

繰り返し処理などに対処する抽象化が存在することで、保守しなければならないコードの量が減るため、エラーが発生する可能性のある箇所も少なくなります。リスト 2 に、Groovy で作成した簡潔な数値分類子のバージョンを記載します。このバージョンでは、Groovy の関数型スタイルのメソッドを使用しています。

リスト 2. Groovy の数値分類子

```
import static java.lang.Math.sqrt

class Classifier {
  def static isFactor(number, potential) {
```

```
    number % potential == 0;
  }

  def static factorsOf(number) {
    (1..number).findAll { isFactor(number, it) }
  }

  def static sumOfFactors(number) {
    factorsOf(number).inject(0, {i, j -> i + j})
  }

  def static isPerfect(number) {
    sumOfFactors(number) == 2 * number
  }

  def static isAbundant(number) {
    sumOfFactors(number) > 2 * number
  }

  def static isDeficient(number) {
    sumOfFactors(number) < 2 * number
  }
}
```

リスト 2 では、リスト 1 の合計のキャッシングを除くすべての処理を、明らかに少ないコードで行っています (合計のキャッシングは、後に記載する例で再び登場します)。例えば、約数を判別するための `factorsOf()` の繰り返し処理は、フィルター基準を使用したコード・ブロック (高階関数) を受け入れる `findAll()` メソッドを使用することで消え去っています。Groovy では、単一パラメーターのブロックで `it` を暗黙パラメーター名として使用することにより、さらにコード・ブロックを簡潔にすることができます。同様に、`sumOfFactors()` メソッドは `inject()` を使用してコード・ブロックを各要素に適用し (0 をシード値として使用)、2 つの値を 1 つの値に減らします。`{i, j -> i + j}` のコード・ブロックが返すのは 2 つのパラメーターの合計です。このコード・ブロックを適用して一度に 1 つのペアずつリストを「畳み込む」ことで、合計を算出します。

Java 開発者は、フレームワーク・レベルでの再利用に慣れているものです。オブジェクト指向言語での再利用に必要となる手法には、かなりの作業が伴います。けれども一般にその労力は、より大きな問題のために確保されているものです。関数型言語は、高階関数によるカスタマイズを取り込むことで、リストやマップといった基本データ構造をベースにした、より細かい粒度での再利用を可能にします。

少数のデータ構造で多数の処理

オブジェクト指向の命令型プログラミング言語では、再利用の単位は、(クラス図に取り込まれた) クラスとこれらのクラスがやりとりするメッセージです。オブジェクト指向の分野で大きな影響力を持つ書籍、『Design Patterns: Elements of Reusable Object-Oriented Software』(「参考文献」を参照) には、1 つのパターンにつき少なくとも 1 つのクラス図があります。OOP の世界で開発者に奨励されているのは、固有のデータ構造を作成して具体的な処理をメソッドという形でアタッチすることですが、関数型プログラミング言語では、これと同じやり方で再利用を実現しようとはしません。関数型プログラミング言語で好まれるのは、少数の重要なデータ構造 (リスト、セット、マップなど) と、これらのデータ構造に対する高度に最適化された処理を使用する方法です。この機構にデータ構造と高階関数を渡して「組み込み」、特定の用途に合わせてこの機構をカスタマイズします。例えば、リスト 2 の `findAll()` メソッドは、フィルター基準を判別する「組み

込み」高階関数としてのコード･ブロックを受け入れます。すると、この機構がフィルター基準を効率的な方法で適用し、フィルタリングされたリストを返すというわけです。

関数レベルでカプセル化することにより、カスタムのクラス構造を作成するよりも粒度の細かい基本的なレベルでの再利用が可能になります。この手法の利点の 1 つが、すでに Clojure で明らかになってきています。最近、ライブラリーに対して賢明な刷新が行われ、`map` 関数が自動的に並列処理されるように作成し直されました。つまり、開発者が手を加えなくても、すべてのマップ処理がパフォーマンス向上の恩恵を受けられるということです。

例えば、XML を構文解析する場合を考えてください。Java にはこの処理を行うためのフレームワークがかなりの数で存在しており、そのそれぞれが、カスタム・データ構造とメソッドのセマンティクスを使用します (例えば、SAX と DOM など)。Clojure はカスタム・データ構造を使用するように強要するのではなく、XML を標準の `Map` 構造に構文解析します。Clojure にはマップを処理するためのツールが数多く含まれているため、XPath スタイルのクエリーを実行するのは簡単で、リストを包含する組み込み関数 `for` を使用すればよいのです (リスト 3 を参照)。

リスト 3. Clojure での XML の構文解析

```
(use 'clojure.xml)

(def WEATHER-URI "http://weather.yahooapis.com/forecastrss?w=%d&u=f")

(defn get-location [city-code]
  (for [x (xml-seq (parse (format WEATHER-URI city-code)))
        :when (= :yweather:location (:tag x))]
    (str (:city (:attrs x)) "," (:region (:attrs x)))))

(defn get-temp [city-code]
  (for [x (xml-seq (parse (format WEATHER-URI city-code)))
        :when (= :yweather:condition (:tag x))]
    (:temp (:attrs x))))

(println "weather for " (get-location 12770744) "is " (get-temp 12770744))
```

リスト 3 では、Yahoo! の天気情報サービスにアクセスして、特定の都市の天気予報を取得します。Clojure は Lisp のバリエーションなので、このコードを理解するための最も簡単な方法は、内側から外側に向かって読むことです。サービス・エンドポイントの実際の呼び出しは、`(parse (format WEATHER-URI city-code))` で行われます。このコードは `String` の `format()` 関数を使用して `city-code` をストリングに埋め込みます。リスト包含関数 `for` は XML を構文解析し、`xml-seq` を使用してキャストした後、`x` というクエリー可能なマップに格納します。`:when` 述部は、一致基準の判定を行います。この例の場合、検索対象は (Clojure キーワードに変換された) `:yweather:condition` というタグです。

データ構造から値を抽出するための構文を理解するには、そのデータ構造の中に含まれている内容を調べると役立ちます。構文解析後に、関連する (天気情報サービスの) 呼び出しによって返されるデータ構造を以下に抜粋します。

```
({:tag :yweather:condition, :attrs {:text Fair, :code 34, :temp 62, :date Tue,
   04 Dec 2012 9:51 am EST}, :content nil})
```

Clojure はマップを扱うように最適化されていることから、キーワードはそれを収容するマップに対する関数になります。リスト 3 の `(:tag x)` の呼び出しは、「`:tag` キーに対応する値を `x` に格

納されているマップから取得する」ことを表しています。したがって、このキーに関連付けられたマップの値が `:yweather:condition` によって生成されます。そこには `attrs` マップが含まれており、このマップから同じ構文を使用して `:temp` を取得します。

Clojure で最初に怖気づいてしまうことの 1 つは、一見するとマップやその他のコアとなっているデータ構造を扱う方法が無数にあることです。けれどもそれは、Clojure では、ほとんどのものが、コアとなる最適化されたデータ構造に帰着しようとするという事実を反映しています。Clojure で行おうとしていることは、固有のフレームワークで構文解析された XML を取得することではなく、ツールがすでに用意されている既存の構造に XML を変換することです。

基本的なデータ構造に依存する利点は、Clojure の XML ライブラリーに見て取れます。1997年に、ツリー形式の構造 (XML 文書など) をトラバースするのに役立つ zipper というデータ構造が作成されました (「参考文献」を参照)。zipper では、座標方向を指定してツリーを構造的にナビゲートすることができます。例えば、ツリーのルートから `(-> z/down z/down z/left)` のようなコマンドを発行すれば、深さのレベルが 2 番目の左側にある要素にナビゲートすることができます。Clojure には、構文解析した XML を zipper に変換して、ツリー形式の構造全体にわたって一貫性のあるナビゲーションを可能にする関数がすでに存在しています。

新しいさまざまなツール

関数型プログラミングには、厄介な問題を簡潔な方法で解決する新たなタイプの手段が用意されています。例えば、Java 開発者が慣れていない遅延データ構造は、値の生成をできる限り遅らせる手段となります。斬新な関数型言語はこのような先進的な機能のサポートを提供する一方、一部のフレームワークはこの機能を Java に作り変えています。その一例として、Totally Lazy フレームワーク (「参考文献」を参照) を使用した数値分類子のバージョンをリスト 4 に記載します。

リスト 4. Totally Lazy による遅延および関数型データ構造を使用した Java の数値分類子

```
import com.googlecode.totallylazy.Predicate;
import com.googlecode.totallylazy.Sequence;

import static com.googlecode.totallylazy.Predicates.is;
import static com.googlecode.totallylazy.numbers.Numbers.*;
import static com.googlecode.totallylazy.predicates.WherePredicate.where;


public class Classifier {
  public static Predicate<Number> isFactor(Number n) {
      return where(remainder(n), is(zero));
  }

  public static Sequence<Number> getFactors(final Number n){
      return range(1, n).filter(isFactor(n));
  }

  public static Sequence<Number> factors(final Number n) {
      return getFactors(n).memorise();
  }

  public static Number sumFactors(Number n){
      return factors(n).reduce(sum);
  }

  public static boolean isPerfect(Number n){
```

```
      return equalTo(n, subtract(sumFactors(n), n));
  }

  public static boolean isAbundant(Number n) {
    return greaterThan(subtract(sumFactors(n), n), n);
  }

  public static boolean isDeficient(Number n) {
    return lessThan(subtract(sumFactors(n), n), n);
  }

}
```

Totally Lazy は、遅延コレクションと流れるようなインターフェース・メソッドを追加し、静的インポートを多用してコードを読みやすくします。次世代言語の機能が羨ましいと思ったら、ある程度調査すれば、特定の問題を解決するための個別の拡張機能が見つかるかもしれません。

言語を問題に対応させる

ほとんどの開発者は、複雑なビジネス上の問題を取り上げて、その問題を Java などの言語で表すことが自分の仕事であると誤解したまま作業しています。そのような誤解が生じる理由は、Java は言語として特に柔軟なわけではないことから、開発者は融通の効かない既存の構造に、どうにかして自分の考えをはめ込まなければならないためです。けれども、開発者が適応性のある言語を使用すると、問題を言語に従わせるのではなく、言語を問題に従わせる可能性が見えるようになります。その可能性を実証したのが、Ruby のような言語 (そしてこの言語に備わった、メインの言語よりも使い勝手の良いドメイン特化言語 (DSL) のサポート) です。最近の関数型言語は、さらに先を行っており、Scala は内部 DSL のホスト言語として設計されました。また、すべての Lisp (Clojure を含む) には、開発者が言語を問題に合わせることができるという点で、比類のない柔軟性があります。例えば、リスト 5 では、リスト 3 の天気の例を Scala の XML プリミティブを使用して実装しています。

リスト 5. XML のための Scala の構文糖

```
import scala.xml._
import java.net._
import scala.io.Source

val theUrl = "http://weather.yahooapis.com/forecastrss?w=12770744&u=f"

val xmlString = Source.fromURL(new URL(theUrl)).mkString
val xml = XML.loadString(xmlString)

val city = xml \\ "location" \\ "@city"
val state = xml \\ "location" \\ "@region"
val temperature = xml \\ "condition" \\ "@temp"

println(city + ", " + state + " " + temperature)
```

適応性を目的に設計された Scala では、演算子の多重定義や暗黙の型などの拡張機能を使用できます。リスト 5 の場合、`\\` 演算子を使用して XPath 風のクエリーを実行できるように Scala が拡張されています。

言語の傾向に合わせる

関数型プログラミングの目標の 1 つは、可変状態を最小限にすることです。リスト 1 には、2 つのタイプの共有された状態があります。`_factors` と `_number` は両方ともコードをテストしやすく

するために存在しているので (このコードのオリジナル・バージョンは、テストのしやすさが最大となるコードを示すために作成されています)、この 2 つを 1 つの大きな関数の中に組み込んでしまえば共有状態を排除することができます。一方、`_sum` は別の理由で存在します。私の見込みでは、このコードのユーザーがチェックしたい数値の分類は 1 つだけではありません (例えば、完全数でないことがわかった場合、おそらく次に過剰数であるかをチェックするはずです)。約数を合計する処理はコストが高くなる可能性があるので、代わりに遅延して初期化されるアクセサーを作成しました。このアクセサーは、最初の呼び出し時に合計を計算して結果を `_sum` メンバー変数に格納することで、以降の呼び出しを最適化します。

ガーベッジ・コレクションと同じように、今ではキャッシングも言語に任せることができるようになっています。リスト 2 に記載した Groovy の数値分類子では、リスト 1 に示されている合計の遅延初期化が省略されていますが、それと同じ機能を実装するとしたら、リスト 6 に示すように分類子を変更します。

リスト 6. ハンド・コーディングによるキャッシュの追加

```
class ClassifierCachedSum {
  private sumCache

  ClassifierCachedSum() {
    sumCache = [:]
  }

  def sumOfFactors(number) {
    if (sumCache.containsKey(number))
      return sumCache[number]
    else {
      def sum = factorsOf(number).inject(0, {i, j -> i + j})
      sumCache.putAt(number, sum)
      return sum
    }
  }
  // ... other code omitted
```

Groovy の最近のバージョンでは、リスト 6 のコードは不要になっています。リスト 7 の改善されたバージョンの分類子を見てください。

リスト 7. メモ化された数値分類子

```
class ClassifierMemoized {
  def static dividesBy = { number, potential ->
    number % potential == 0
  }
  def static isFactor = dividesBy.memoize()

  def static factorsOf(number) {
    (1..number).findAll { i -> isFactor.call(number, i) }
  }

  def static sumFactors = { number ->
    factorsOf(number).inject(0, {i, j -> i + j})
  }
  def static sumOfFactors = sumFactors.memoize()

  def static isPerfect(number) {
    sumOfFactors(number) == 2 * number
  }
```

```
  def static isAbundant(number) {
    sumOfFactors(number) > 2 * number
  }

  def static isDeficient(number) {
    sumOfFactors(number) < 2 * number
  }
}
```

純粋関数 (副次作用のない関数) であれば、いずれもリスト 7 の `sumOfFactors()` メソッドのようにメモ化することができます。関数をメモ化すると、ランタイムが繰り返し発生する値をキャッシングできるようになるため、キャッシュをハンド・コーディングする必要はなくなります。事実 (リスト 4 のコードで)、実際の処理を行う `getFactors()` と、`getFactors()` のメモ化バージョンである `factors()` メソッドとの関係に注目してください。Totally Lazy はまた、Java にメモ化機能を追加し、さらにもう 1 つの先進的な関数型の機能を主流のプログラミング言語に提供します。

ランタイムがより能力をアップして余分なオーバーヘッドが出てくると、開発者は時間のかかる、ありふれた処理を言語に任せて、より重要な問題を検討できるようになります。Groovy のメモ化は、数ある例の中の 1 つです。Totally Lazy などのフレームワークをはじめ、最近の言語はどれもが、基礎となるランタイムが許容する関数型の構成体を次々と追加しています。

まとめ

ランタイムが能力を得て、言語がより強力な抽象化を可能にするなか、開発の世界は次第に関数型へと移行しています。それに伴い、開発者達は結果を得る方法ではなく、結果が持つ意味についてより多くの時間をかけて検討できるようになってきています。高階関数のような抽象化が言語に現われると、これらの抽象化が高度に最適化された処理を行うためのカスタマイズ機構になります。これを利用すれば、開発者は、XML などの問題を処理するフレームワークを作成するのではなく、XML を既存のツールで処理できるデータ構造に変換することができます。

連載「関数型の考え方」は、20 回目となる今回の記事で中断しますが、新しい連載で、引き続き 3 つの次世代 JVM 言語を詳しく探っていきます。連載「Java.next」では、開発者が自分たちの近い将来を垣間見て、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 いて行えるように支援します。

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。