関数型の考え方: 関数型の観点で考える、第 2 回

関数型プログラミングおよび制御について探る

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2011年 7月 01日

関数型の言語およびフレームワークでは、繰り返し処理や、並行処理、状態管理などの一般的なコーディングの詳細はランタイムに制御させます。だからと言って、必要なときに制御を取り戻すことができないわけではありません。関数型の観点で考えるときに重要となる側面の 1 つは、どれだけの制御をどのような場合に引き渡すべきかを知ることです。

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。

この連載の第 1 回では、まず関数型プログラミングの特徴について説明し、関数型プログラミングに特徴的な概念が Java 言語、そしてより関数型の性質を持った言語ではどのように現れるかを紹介しました。今回の記事では、関数型プログラミングの重要な概念に焦点を当てる前回からのツアーを引き継ぎ、第一級関数、最適化、そしてクロージャーを取り上げます。けれども、この記事の根底にあるテーマは、制御です。いつ制御したいか、いつ制御する必要があるか、そしていつ制御を引き渡すべきかを学んでください。

第一級関数と制御

前回は、Functional Java ライブラリー (「参考文献」を参照) を使用して、関数型の `isFactor()` および `factorsOf()` メソッドを使った数値分類子の実装を紹介しました (リスト 1 を参照)。

リスト 1. 関数型の数値分類子

```
import fj.F;
import fj.data.List;
import static fj.data.List.range;
import static fj.function.Integers.add;
```

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

public class FNumberClassifier {

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

    public List<Integer> factorsOf(final int number) {
        return range(1, number+1).filter(new F<Integer, Boolean>() {
            public Boolean f(final Integer i) {
                return number % i == 0;
            }
        });
    }

    public int sum(List<Integer> factors) {
        return factors.foldLeft(fj.function.Integers.add, 0);
    }

    public boolean isPerfect(int number) {
        return sum(factorsOf(number)) - number == number;
    }

    public boolean isAbundant(int number) {
        return sum(factorsOf(number)) - number > number;
    }

    public boolean isDeficiend(int number) {
        return sum(factorsOf(number)) - number < number;
    }
}
```

`isFactor()` および `factorsOf()` メソッドでは、私はループ処理アルゴリズムの制御をフレームワークに任せました。したがって、処理を指定された回数繰り返す最善の方法は、フレームワークが決定するようになっています。フレームワーク (あるいは、Clojure や Scala などの関数型言語を選ぶとしたら、言語) によって実装を最適化できれば、当然メリットはあります。最初は、これほどの制御を引き渡すことに気が乗らないかもしれませんが、こうすることが、プログラミング言語およびランタイムでの全体的な傾向となっていることに注意してください。次第に開発者は、プラットフォームのほうが効率的に扱うことができる詳細な処理については、プラットフォームに任せるようになっています。私にとって、JVM でのメモリー管理は心配の範疇ではありません。プラットフォームがそれを忘れさせてくれるためです。もちろん、時にはプラットフォームにメモリー管理を任せると事態が余計にややこしくなることもありますが、日々のコーディング作業で得られるメリットに比べれば、たいしたことはありません。高階関数や第一級関数などの関数型言語の構成体のおかげで、私は抽象化の梯子を 1 段上に昇り、コードに目的の処理をどのように実行させるかではなく、コードにどんな処理を行なわせるかに専念することができます。

Java で関数型のコーディングを行うには、Functional Java フレームワークを使用したとしても厄介な作業になります。Java 言語には、そのための構文も構成体もないからです。そのような構文や構成体を持っている言語の場合、関数型コーディングはどのようになるのでしょうか。

Clojure での分類子

Clojure は、JVM のために設計された関数型の LISP です (「参考文献」を参照)。リスト 2 に記載する、Clojure で作成した数値分類子を見てください。

リスト 2. 数値分類子の Clojure での実装

```
(ns nealford.perfectnumbers)
(use '[clojure.contrib.import-static :only (import-static)])
(import-static java.lang.Math sqrt)

(defn is-factor? [factor number]
  (= 0 (rem number factor)))

(defn factors [number]
  (set (for [n (range 1 (inc number)) :when (is-factor? n number)] n)))

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

(defn perfect? [number]
  (= number (- (sum-factors number) number)))

(defn abundant? [number]
  (< number (- (sum-factors number) number)))

(defn deficient? [number]
  (> number (- (sum-factors number) number)))
```

リスト 2 のコードは、筋金入りの Lisp 開発者でなくても (特に、内側から外側へと読めるようになれば) 簡単に理解することができます。例えば、`is-factor?` メソッドは 2 つのパラメーターを取り、`factor` で `number` を除算すると、その剰余が 0 と等しくなるかどうかを聞いています。`perfect?`、`abundant?`、`deficient?` の各メソッドについても、リスト 1 の Java 実装を参照すれば簡単に理解できるはずです。

`sum-factors` メソッドが使用するのは、組み込み `reduce` メソッドです。`sum-factors` は、各要素の最初のパラメーターとして提供された関数 (この例では `+`) を使って、リストから 1 度に 1 つの要素を減らします。`reduce` メソッドは、言語やフレームワークによって異なる形で現れます。例えば、リスト 1 の Functional Java では、`foldLeft()` メソッドという形になっています。`factors` メソッドは数値のリストを返すため、ここではリストを 1 つずつ処理して、各要素を累積されていく合計に足していきます。この合計が、`reduce` の戻り値です。このように、高階関数と第一級関数の観点から考えることに慣れてくれば、コードのノイズを大幅に削減することができます。

`factors` メソッドはシンボルのランダムな集合のように見えるかもしれませんが、Clojure の強力なリスト操作機能の 1 つであるリスト内包表記を目にしたことがあれば、理解できるはずです。前と同じく、`factors` を理解する最も簡単な方法は、内側から外側へと読んでいくことです。言語特有の用語が出てきても、混乱しないでください。Clojure の `for` キーワードは `for` ループを意味するのではありません。そうではなく、あらゆるフィルタリングや変換を行う際に使われる構成体であると考えてください。この例では、1 から (`number` + 1) までの範囲の数値を、`is-factor?` 述部 (リスト 2 で定義した `is-factor` メソッドです。第一級関数を多用していることに注意してください) を使ってフィルタリングし、一致する数値を返すように指定しています。この操作から返されるのは、フィルタリング基準を満たす数値のリストです。これらの数値は、重複を取り除くために 1 つのセットにしています。

新しい言語を学ぶのは骨が折れますが、その機能を理解すれば、関数型言語を学ぶだけの価値は大いにあります。

最適化

関数型スタイルへの転向によってもたらされるメリットの 1 つは、関数型の言語またはフレームワークが提供する高階関数のサポートを利用できることです。けれども、その制御を手放したくない場合にはどうなるのでしょうか。前の例では、繰り返し処理メカニズム内部の振る舞いをメモリー・マネージャーの内部構造になぞらえました。ほとんどの場合、これらの詳細を心配しなくて済むことは嬉しいことですが、最適化および同様の微調整を行う場合のように、詳細が重要になることもあります。

「関数型の観点で考える、第 1 回」に記載した Java による 2 つの 数値分類子では、約数を判別するコードを最適化しました。最適化をせずに、そのまま素直にモジュロ (%) 演算子を使用して 2 から対象の数値までのすべての数値について調べ、それが対象の数値の約数であるかどうかを判別するアルゴリズムは、極めて非効率です。このアルゴリズムは、約数がペアであることを利用すれば最適化することができます。例えば 28 の約数を探している場合、2 を見つければ、それと同時に 14 も約数であることがわかります。約数をペアで取得できるとしたら、対象の数値の平方根まで調べるだけで済みます。

Java では簡単だった最適化は、Functional Java を利用した場合は不可能のように思えます。なぜなら、開発者が繰り返し処理メカニズムの実装を直接制御していないためです。けれども関数型の観点で考える方法を学ぶ過程では、別の発想を働かせることができるように、この種の制御をするという考えを手放して、別の発想ができるようにする必要があります。

関数型の観点でこの問題について考えてみると、必要なのは、「1 から `number` までの数値から number の約数をすべて抽出し、`isFactor()` 述部に一致する約数のみを保持すること」であると言い換えることができます。これを実装しているのが、リスト 3 です。

リスト 3. `isFactor()` メソッド

```
public List<Integer> factorsOf(final int number) {
    return range(1, number+1).filter(new F<Integer, Boolean>() {
        public Boolean f(final Integer i) {
            return number % i == 0;
        }
    });
}
```

宣言型の観点からすると簡潔ではあるものの、リスト 3 のコードはすべての数値について調べるため、かなり非効率です。最適化の方法を理解すれば (約数をペアで取得することで、数値の平方根までのみを調べること)、問題をさらに以下のように言い換えることができます。

1. 1 から対象の数値の平方根までの範囲で、対象の数値の約数をすべて抽出します。
2. 対象の数値を、取得した約数のそれぞれで除算してペアの約数を取得し、それを約数のリストに追加します。

これを実現することを念頭に置けば、Functional Java ライブラリーを使って `factorsOf()` メソッドを最適化したものを作成することができます (リスト 4 を参照)。

リスト 4. 最適化された約数検出メソッド

```
public List<Integer> factorsOfOptimzied(final int number) {
    List<Integer> factors =
        range(1, (int) round(sqrt(number)+1))
        .filter(new F<Integer, Boolean>() {
            public Boolean f(final Integer i) {
                return number % i == 0;
            }});
    return factors.append(factors.map(new F<Integer, Integer>() {
                                      public Integer f(final Integer i) {
                                          return number / i;
                                      }}))
                                      .nub();
}
```

リスト 4 のコードは、前述のアルゴリズムをベースに、Functional Java フレームワークに必要な型破りな構文を使って作成されています。このコードではまず、1 から対象の数値の平方根プラス 1 (すべての約数を確実に取得するため、プラス 1 します) までの数値を取得します。次に、前のバージョンと同じモジュロ演算子を Functional Java コード・ブロック内にラップして使用し、結果を抽出します。抽出されたリストは、`factors` 変数に保存します。(内側から読んで) 4 番目に、この約数のリストを取得し、`map()` 関数を実行します。この関数は、各要素に対してコード・ブロックを実行し (各要素を新しい値にマッピングします)、新しいリストを生成します。約数のリストには、対象の数値の平方根までの約数がすべて含まれるので、今度はこのリストの各約数を対象の数値で除算して、それぞれのペアの約数を取得する必要があります。これが、`map()` メソッドに送られたコード・ブロックの処理内容です。ペアの約数のリストを取得したところで、5 番目にこのリストを元のリストに追加します。仕上げのステップとして、`Set` ではなく `List` に約数を保持しているという事実を考慮する必要があります。List メソッドは、このようなタイプの操作には便利ですが、このアルゴリズムの副次作用は、整数の平方根が現れると重複したエントリーができてしまうことです。例えば、対象の数値が 16 だとすると、整数の平方根 4 が約数のリストに重複して 2 つ含まれることになります。便利な `List` メソッドを使い続けるために必要なことは、その `nub()` メソッドを最後に呼び出して、すべての重複を削除することのみです。

関数型プログラミングのような高度な抽象化を使用するときには、実装の詳細に関して通常よく理解していないからといって、必要に応じて手を加えられないというわけではありません。Java プラットフォームは下位レベルの詳細をほとんど遮蔽しますが、覚悟を決めれば、必要な詳細レベルまで掘り下げることもできます。それは関数型プログラミングの構成体でも同様なので、通常は積極的に抽象化に詳細を任せることにし、詳細が極めて重要で抽象化に任せられないときのために、時間を確保しておいてください。

これまでに記載した Functional Java コードのすべてで圧倒的に目立っているのは、ブロック構文です。ブロック構文では Generics と匿名内部クラスを、ある種の擬似コード・ブロックであるクロージャー・タイプの構成体として使用します。クロージャーは、関数型言語の共通機能の 1 つです。関数型の世界で、クロージャーをこれほどまでに有用にしている理由とは一体何でしょうか。

クロージャーを特別なものにしている理由

クロージャーとは、その内部で参照されるすべての変数を暗黙的にバインドする関数のことです。別の言い方をすれば、この関数 (メソッド) は、自身が参照するものを取り巻くコンテキストをカプセル化します。関数型言語や関数型フレームワークでは頻繁にクロージャーを移植可能な

実行メカニズムとして使用し、`map()` などの高階関数に変換コードとして渡します。Functional Java は匿名内部クラスを使って「真の」クロージャーの振る舞いを模倣するものの、Java はクロージャーをサポートしないため、匿名内部クラスは完全にはクロージャーを模倣することはできません。では、「真の」クロージャーの振る舞いとは何を意味するのでしょうか。

リスト 5 は、クロージャーを特別なものにしている一例です。このコードは、コード・ブロック・メカニズムを介してクロージャーをサポートする Groovy で作成されています。

リスト 5. クロージャーを説明する Groovy コード

```
def makeCounter() {
  def very_local_variable = 0
  return { return very_local_variable += 1 }
}

c1 = makeCounter()
c1()
c1()
c1()
c2 = makeCounter()

println "C1 = ${c1()}, C2 = ${c2()}"
// output: C1 = 4, C2 = 1
```

`makeCounter()` メソッドは、最初に適切な名前のローカル変数を定義し、それからその変数を使用するコード・ブロックを返します。`makeCounter()` メソッドの戻りタイプは値ではなく、コード・ブロックであることに注目してください。このコード・ブロックは、ローカル変数の値をインクリメントして、その値を返すだけに過ぎません。このコードに配置した 2 つの明示的な `return` 呼び出しは両方とも Groovy ではオプションですが、これらの呼び出しがなければ、このコードはさらに謎めいたものなってしまいます！

`makeCounter()` メソッドを実際に試してみるために、コード・ブロックを `C1` 変数に割り当てた後、3 回呼び出しています。コード・ブロックを実行するために使用している Groovy の構文糖は、コード・ブロックの変数の隣に括弧を配置することです。次に、`makeCounter()` を再び呼び出し、コード・ブロックの新しいインスタンスを `C2` に割り当てます。そして最後に `C1` を `C2` と併せて再び実行します。コード・ブロックのそれぞれが個別の `very_local_variable` のインスタンスを追跡していることに注意してください。コンテキストをカプセル化するとは、そういう意味です。ローカル変数がメソッド内で定義されているとしても、コード・ブロックはそのローカル変数を参照することから、コード・ブロックとそのローカル変数はバインドされます。これはつまり、コード・ブロックのインスタンスは、その存続期間中、変数の経過を追うという意味です。

この振る舞いと最も近い Java での振る舞いは、リスト 6 のようになります。

リスト 6. Java での `MakeCounter`

```
public class Counter {
    private int varField;

    public Counter(int var) {
        varField = var;
    }

    public static Counter makeCounter() {
        return new Counter(0);
    }

    public int execute() {
        return ++varField;
    }
}
```

`Counter` クラスにはいくつかの変形が考えられますが、状態の管理に行き詰ってしまうことに変わりはありません。これは、クロージャーを使用することが、関数型の考え方 (つまり、ランタイムに状態の管理を任せるということ) を示す典型例となる理由を説明しています。自分で無理矢理フィールドを作成して状態を慎重に管理するのではなく (これには、自分が作成したコードをマルチスレッド環境で使用するという恐ろしい可能性も含まれます)、状態を目に見えないところで言語やフレームワークに管理させてください。

次の Java リリースでは、ようやくクロージャーが用意される予定です (この件については、幸いなことにこの記事で説明する範囲を超えています)。Java にクロージャーが登場することで、2 つの喜ばしいメリットがもたらされるはずです。その 1 つは、フレームワークおよびライブラリーを作成するための機能が大幅に単純化されること、そしてそれと同時に構文が改善されることです。もう 1 つのメリットは、JVM で動作するすべての言語に共通のクロージャー・サポートが提供されることです。クロージャーは多くの JVM 言語でサポートされていますが、そのすべての言語が独自のバージョンを実装しなければならないため、それらの言語の間でクロージャーを受け渡すのが煩雑な作業になっています。Java 言語が単一のフォーマットを定義すれば、他のすべての言語はその共通フォーマットを利用できるというわけです。

まとめ

下位レベルの詳細の制御を他に委ねることは、ソフトウェア開発での全般的な傾向となっています。私たちはこれまでもガーベッジ・コレクション、メモリー管理、そしてハードウェアの違いに関する責任を喜んで引き渡してきましたが、関数型プログラミングは、さらに上のレベルの抽象化への移行を意味します。それは、繰り返し処理、並行処理、状態管理などのより一般的なコーディングの詳細を可能な限りランタイムに任せるということです。だからと言って、必要な場合に制御を取り戻せないというわけではありません。ただし、制御が必要なときに取り戻せるのであって、制御を強制されるわけではありません。

次回の記事では、引き続き Java での関数型プログラミングの構成体を詳しく探り、カリー化および部分メソッドの適用について紹介して Java 関連のトピックを締めくくります。

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。