関数型の考え方: 関数型の観点で考える、第 1 回

関数型プログラマーのような考え方を習得する

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2011年 7月 01日

バグが少ない上に生産性が高いという特徴を引っ提げた関数型プログラミングへの関心が、最近、爆発的に高まっています。けれども多くの開発者が、ある種のジョブに対して関数型言語のどこが魅力的であるのか理解しようとして、理解できずに終わっています。新しい言語の構文を学ぶのは簡単ですが、今までとは違う考え方を学ぶのは大変です。連載「関数型の考え方」の第 1 回目では、著者の Neal Ford が関数型プログラミングの特徴的な概念をいくつか紹介し、これらの概念を Java と Groovy の両方で適用する方法を説明します。

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。

ちょっとの間、自分は木こりであると想像してみてください。あなたは森で使う斧として、最上の斧を持っています。そのおかげで、木こり仲間の間では最も高い生産性を誇っています。そんなある日のこと、1 人の人物がやってきて、新しい方法で木を伐採するツールとしてチェーンソーを紹介し、その優れている点を褒め称えました。その人はチェーンソーの販売員で、話に説得力があったので、あなたはチェーンソーを買いました。けれども、その使い方がわかりません。試しに持ち上げて、力いっぱい木に振り下ろしてみましたが、それでは今までの木の伐採方法と変わりがありません。あなたはすぐに、この目新しいチェーンソーというものは単なる一時的流行だと結論付けて、再び斧で木を伐採することにしました。すると、別の誰かがやってきて、チェーンソーのエンジンをかける方法を教えてくれました。

この話はチェーンソーを関数型プログラミングに置き換えてみると、おそらく皆さんにも関係してくるはずです。まったく新しいプログラミング・パラダイムに伴う問題は、新しい言語を習得することではありません。結局のところ、言語の構文は細かい部分でしかありません。難しい部

分は、これまでとは異なる考え方を身に付けるところにあります。そこでチェーンソーのエンジンをかけるべく登場するのが、関数型プログラマーである私です。

連載「関数型の考え方」へようこそ。この連載で掘り下げるテーマは関数型プログラミングですが、関数型プログラミング言語についてのみ掘り下げるわけではありません。これから説明するように、「関数型」のやり方でコードを作成するには、設計、トレードオフ、再利用可能なさまざまなビルディング·ブロック、そしてその他多くの本質を理解することが関係してきます。この連載では可能な限り、関数型プログラミングに特徴的な概念を Java (または Java に近い言語) で使用する例を説明し、その後ほかの言語に話題を移し、Java 言語にはまだ存在しない機能を説明したいと思います。最初から、モナド (「参考文献」を参照) のような型破りで奥の深いトピックに飛び付くことはせずに (ただし、いずれはトピックとして取り上げます)、問題に関する新しい考え方を段階的に紹介していきます (皆さんは、あるところではこの新しい考え方をすでに適用しています。ただそれに気付いていないだけです)。

今回とそれに続く 3 回の記事は、関数型プログラミングのコアとなる概念をはじめ、関数型プログラミングに関係するトピックを駆け足で巡るツアーとなります。そのうちいくつかの概念については、連載を通して少しずつコンテキストとニュアンスを構築していくなかで再び取り上げ、詳しく説明します。このツアーの出発点としては、ある問題に関する 2 つの異なる実装を調べます。1 つは命令型で作成された実装、もう 1 つは、それよりも関数型に近い実装です。

数値分類子

異なるプログラミング・スタイルについて語るには、比較するためのコードが必要です。そこで最初に紹介する例は、私の著書『The Productive Programmer』(「参考文献」を参照) と以前に私が developerWorks に寄稿した連載記事「進化するアーキテクチャーと新方式の設計」の「Test-driven Design, Part 1」および「Test-driven Design, Part 2」に記載した、ある問題に対するコードを変形したものです。このコードを選んだ理由は、少なくとも一部には、上記の 2 つの記事ではコードの設計について詳しく説明していることが理由となっています。この 2 つの記事で称賛した設計に何も問題はありませんが、それとは異なる設計をこの記事で紹介する論拠については示すつもりです。

この問題で要件となっているのは、1 より大きい正の整数を、完全数、過剰数、または不足数のいずれかとして分類することです。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。同様に、過剰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) の和がその数自身よりも小さくなる数のことです。

命令型の数値分類子

リスト 1 に、上記の要件を満たす命令型のクラスを記載します。

リスト 1. `NumberClassifier` (問題に対する命令型によるソリューション)

```
public class Classifier6 {
    private Set<Integer> _factors;
    private int _number;

    public Classifier6(int number) {
        if (number < 1)
            throw new InvalidNumberException(
            "Can't classify negative numbers");
```

```
        _number = number;
        _factors = new HashSet<Integer>>();
        _factors.add(1);
        _factors.add(_number);
    }

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

    public Set<Integer> getFactors() {
        return _factors;
    }

    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 int sumOfFactors() {
        calculateFactors();
        int sum = 0;
        for (int i : _factors)
            sum += i;
        return sum;
    }

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

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

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

    public static boolean isPerfect(int number) {
        return new Classifier6(number).isPerfect();
    }
}
```

このコードで注目すべきいくつかの点を以下に記載します。

- 広範なユニット・テストがあります (その理由の 1 つは、このコードはテスト駆動型開発について説明するために作成されたからです)。
- このクラスは、多数のメソッドが凝集された形で構成されています。これは、クラスを作成する際にテスト駆動型開発を使用したことによる副次効果です。
- `calculateFactors()` メソッドではパフォーマンスの最適化がなされています。このクラスの本質は、約数を集めて合計し、最終的に分類できるようにすることです。約数は常にペアで取得することができます。例えば、問題の数値が 16 の場合には、約数 2 を取得すれば、2 x 8 = 16 であることから 8 も取得できることになります。約数をペアで取得すれば、対象の数値

の平方根まで調べるだけで済みます。これがまさに、`calculateFactors()` メソッドが行っていることです。

(多少) 関数型に近い分類子

同じテスト駆動型開発手法を使用して、分類子の別のバージョンを作成しました。リスト 2 に、このバージョンを記載します。

リスト 2. 多少関数型に近い数値分類子

```
public class NumberClassifier {

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

    static public Set<Integer> factors(int number) {
        HashSet<Integer> factors = new HashSet<Integer>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(number, i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

    static public int sum(Set<Integer> factors) {
        Iterator it = factors.iterator();
        int sum = 0;
        while (it.hasNext())
            sum += (Integer) it.next();
        return sum;
    }

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

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

    static public boolean isDeficient(int number) {
        return sum(factors(number)) - number < number;
    }
}
```

分類子の 2 つのバージョンにはわずかな違いしかありませんが、その違いは重要です。一番の違いは、リスト 2 では意図的に状態が共有されないようにしていることです。状態の共有をなくすこと (または少なくとも減らすこと) は、関数型プログラミングで優先される抽象化の 1 つです。そのため、メソッド間で中間結果として状態を共有する (リスト 1 の `factors` フィールドを参照) 代わりに、上記のリストではメソッドを直接呼び出すようにすることで、状態を使わなくしています。設計の観点からすると、これによって `factors()` メソッドは長くなりますが、`factors` フィールドがメソッドから「洩れ出す」ことはありません。リスト 2 のバージョンは一貫して static メソッドで構成されていることにも注目してください。メソッド間で共有する情報はないため、スコープを設定してカプセル化する必要性は少なくなります。これらのメソッドに対しては、メソッドが要求しているタイプの入力パラメーターを渡せば、これらのメソッドはすべて完全に機能します (これは、純粋関数という概念の一例です。この概念については、今後の記事で詳しく説明します)。

関数

関数型プログラミングは、コンピューター・サイエンスのなかでも最近爆発的に関心が高まっている分野として、広範かつ無秩序な広がりを見せています。JVM では、バグが少なく、生産性に優れ、簡潔で、これまでよりも儲かるなどといった決まり文句とともに新しい関数型言語 (Scala や Clojure など) とフレームワーク (Functional Java や Akka など) が登場しています (「参考文献」を参照)。ここでは、初めから関数型プログラミング言語の全体をトピックとして取り上げるのではなく、いくつかの重要な概念に焦点を当て、これらの概念から派生するいくつかの興味深い関連事項を追って行こうと思います。

オブジェクト指向言語の主要な抽象化がクラスであるのと同じく、関数型プログラミングの中核にあるのは関数です。処理のビルディング・ブロックを形作る関数には、従来の命令型言語にはない特徴があります。

高階関数

高階関数とは、他の関数を引数として取るか、あるいは他の関数を結果として返すことができる関数のことです。このような構成体は、Java 言語にはありません。これに最も近い形としては、実行する必要のあるメソッドの「ホルダー」としてクラス (しばしば匿名クラス) を使用することくらいです。Java にはスタンドアロン関数 (またはメソッド) というものはないため、関数を関数の戻り値にすることも、パラメーターとして渡すこともできません。

関数型言語で高階関数の機能が重要となる理由は、少なくとも 2 つあります。1 つは、高階関数を使用するということは、言語の各部分がどのように組み合わされるかを推測できることを意味するからです。例えば、リストをトラバースして各要素に 1 つ (または複数) の高階関数を適用する汎用メカニズムを作成することで、クラス階層でメソッドの類をすべて排除することができます (後で、この構成体の例を紹介します)。そして 2 つ目の理由として、関数を戻り値として使用できるようにすれば、極めて動的に適応可能なシステムを作成する機会が作り出されることになります。

高階関数を使用して解決することが効を奏する問題は、関数型言語に特有のものではありません。けれども問題を解決する方法は、関数型の観点で考えると違ってきます。例えば、リスト 3 を見てください (コード・ベースから抜粋した部分です)。このコードは、保護されたデータ・アクセスを行うメソッドの例です。

リスト 3. 再利用できる可能性のあるコード・テンプレート

```
public void addOrderFrom(ShoppingCart cart, String userName,
                     Order order) throws Exception {
    setupDataInfrastructure();
    try {
        add(order, userKeyBasedOn(userName));
        addLineItemsFrom(cart, order.getOrderKey());
        completeTransaction();
    } catch (Exception condition) {
        rollbackTransaction();
        throw condition;
    } finally {
        cleanUp();
    }
}
```

リスト 3 のコードは、初期化を行ってから何らかの処理を実行して、すべてが上手く行った場合にはトランザクションを完了し、そうでなければロールバックし、最後にリソースをクリーンアップします。明らかに、このコードの定型部分は再利用できる可能性があります。このような場合、オブジェクト指向の言語では、構造を作成して再利用可能にするのが通常です。そこで、これからこの例に Gang of Four のデザイン・パターン (「参考文献」を参照) のうち、Template Method (テンプレート・メソッド) パターンと Command (コマンド) パターンの 2 つを組み合わせます。Template Method パターンに従えば、共通の定型コードを継承階層に移し、アルゴリズムの詳細を子クラスに任せることになります。Command デザイン・パターンは、周知の実行セマンティクスを持つクラスのなかに振る舞いをカプセル化する手段となります。この 2 つのパターンをリスト 3 のコードに適用すると、リスト 4 の結果になります。

リスト 4. リファクタリングした注文用のコード

```
public void wrapInTransaction(Command c) throws Exception {
    setupDataInfrastructure();
    try {
        c.execute();
        completeTransaction();
    } catch (Exception condition) {
        rollbackTransaction();
        throw condition;
    } finally {
        cleanUp();
    }
}

public void addOrderFrom(final ShoppingCart cart, final String userName,
                         final Order order) throws Exception {
    wrapInTransaction(new Command() {
        public void execute() {
            add(order, userKeyBasedOn(userName));
            addLineItemsFrom(cart, order.getOrderKey());
        }
    });
}
```

リスト 4 では、コードの汎用部分を抽出して `wrapInTransaction()` メソッド (このメソッドのセマンティクスは、おそらくお気付きのとおり、基本的には Spring の `TransactionTemplate` のシンプル・バージョンです) の中に入れ、`Command` オブジェクトを作業単位として渡します。`addOrderFrom()` メソッドはコマンド・クラスの匿名内部クラスの作成定義にまで縮小され、2 つの作業項目をラップします。

必要な振る舞いをコマンド・クラスにラップするのは、純粋に Java の設計の成果物であり、ここにはスタンドアロンの振る舞いの類は一切含まれません。Java での振る舞いはすべて、クラス内部に含まれていなければなりませんが、Java 言語の設計者たちでさえも、この設計の不備にはすぐに気付きました。後から考えれば、クラスにアタッチされていない振る舞いはあり得ないと考えるのは、少し浅はかです。JDK 1.1 ではこの欠陥を是正するために、匿名内部クラスが追加されました。これにより少なくとも、ほんのわずかなメソッド (構造的ではない純粋な関数) で多数の小さなクラスを作成するための構文糖が提供されました。Steve Yegge 氏の「Execution in the Kingdom of Noun」(「参考文献」を参照) は、Java のこの側面に関する非常に愉快なエッセイです。

本当に必要なのはクラス内のメソッドだけだとしても、Java では `Command` クラスのインスタンスを作成せざるを得ません。クラス自体が提供するメリットは何もなく、フィールドもなければ、(Java が自動生成するコンストラクターを別にすると) コンストラクターも状態も一切ありません。クラスはただ単に、振る舞いをメソッドの中に含めるラッパーとして機能します。一方、関数型言語ではクラスではなく、高階関数によって対処します。

ほんの束の間、Java 言語を離れるとしたら、クロージャーを使用することで、関数型プログラミングの理想にセマンティックな面で近づくことができます。リスト 5 に示す例は、行っていることはリスト 4 のリファクタリングしたコードの例と同じですが、ここでは Java の代わりに Groocy (「参考文献」を参照) を使用しています。

リスト 5. コマンド・クラスの代わりに Groovy のクロージャーを使用した場合

```
def wrapInTransaction(command) {
  setupDataInfrastructure()
  try {
    command()
    completeTransaction()
  } catch (Exception ex) {
    rollbackTransaction()
    throw ex
  } finally {
    cleanUp()
  }
}

def addOrderFrom(cart, userName, order) {
  wrapInTransaction {
    add order, userKeyBasedOn(userName)
    addLineItemsFrom cart, order.getOrderKey()
  }
}
```

Groovy の場合、波括弧 `{}` の中身はすべてコード・ブロックです。コード・ブロックであればパラメーターとして渡すことができるので、高階関数を真似ることができます。裏では Groovy によって Command デザイン・パターンが実装されています。Groovy では、クロージャー・ブロックのそれぞれが実は Groovy クロージャー・タイプのインスタンスです。クロージャーには `call()` メソッドが組み込まれており、クロージャー・インスタンスを格納する変数の後に空の括弧を配置すると、このメソッドが自動的に呼び出されます。Groovy は、適切なデータ構造とそれに対応する構文糖を一緒に言語自体に組み込むという方法で、関数型プログラミングのような振る舞いを実現しました。今後の記事で紹介するように、Groovy には Java の機能のみならず、他の関数型プログラミングの機能も組み込まれています。また、クロージャーと高階関数との興味深い比較についても、今後の記事で取り上げる予定です。

第一級関数

関数型言語での関数は、第一級関数と見なされます。これは、他のあらゆる言語構成体 (変数など) を使えるところであれば、関数も同じくそこで使えることを意味します。第一級関数の存在は、関数を思いがけない方法で使えるようにすることから、ソリューションについて違った考え方を持たざるを得なくなります。例えば、比較的一般的な操作を (詳細に微妙な意味を持たせて) 標準データ構造に適用するなどのソリューションです。このようなソリューションが、関数型言語での基本的な考え方、つまり「ステップではなく、結果に重点を置く」という考え方への転換を浮き彫りにします。

命令型プログラミング言語では、アルゴリズムのアトミックなステップのそれぞれについて考える必要があります。このことは、リスト 1 のコードを見れば明らかです。数値分類子を解決するには、約数を収集する方法を正確に理解しなければなりませんでした。それは、数値をループ処理して約数であることを判別する具体的なコードを作成しなければならないことを意味します。けれども、リストをループ処理して各要素に対して処理を行うというのは、実に一般的なことのように思えます。そこで、この数値分類コードを Functional Java フレームワークを使って実装し直すと、リスト 6 のコードになります。

リスト 6. 関数型の数値分類子

```
public class FNumberClassifier {

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

    public List<Integer> factors(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(factors(number)) - number == number;
    }

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

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

リスト 6 とリスト 2 の主な違いは、`sum()` および `factors()` の 2 つのメソッドにあります。`sum()` メソッドは、Functional Java の List クラスのメソッドの 1 つ、`foldLeft()` を利用しています。これは、リストの畳み込みの一般化である、catamorphism と呼ばれるリスト操作の概念のバリエーションの 1 つです。この例での「fold left」は、以下の意味を持ちます。

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。

これはまさに、数値のリストを合計するときの処理です。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、リストの要素の最後まで続けます。Functional Java は高階関数を提供し (この例では、`Integers.add` 列挙)、自動的にその高階関数を適用します (もちろん、実際には Java に高階関数はありませんが、特定のデータ構造と型に制限すれば、高階関数にかなり近いものを作成することができます)。

リスト 6 で興味深いもう 1 つのメソッドは、`factors()` です。このメソッドは、前述の「ステップではなく、結果に重点を置く」というアドバイスの説明になります。ある数値の約数を見つけるという問題の本質は何でしょうか。別の言い方に置き換えると、対象の数値までのすべての数値のリストがある場合、そのなかのどれが対象の数値の約数であるかを判別する方法を考えてみてください。これが意味するのは、フィルタリング処理です。数値のリスト全体をフィルタリングすることで、基準に適合しない数値を除外することができます。このメソッドは要するに、まず 1 から指定の数値 (この指定の数値を範囲に含めるために `+1` としています) までの範囲の数値を取得し、次に `f()` メソッドのコードに基づいて数値のリストをフィルタリングし (これは、特定のデータ型のクラスを作成できるようにする、Functional Java の方法です)、最後に値を返すというものです。

プログラミング言語全般での傾向として、上記のコードは見た目よりも遥かに大きな概念を表しています。かつては、メモリーの割り当て、ガーベッジ・コレクション、ポインターなど、あらゆる厄介な作業を開発者が処理しなければなりませんでした。やがて、この厄介な役割を次第に言語が引き受けるようになってきました。コンピューターの能力が増すにつれ、ますます多くの一般的な (自動化できる) タスクが言語とランタイムに任されるようになっています。私は Java 開発者として、メモリーに関するすべての問題を言語に任せることにすっかり慣れていますが、関数型プログラミングは、言語に任せられる一般的なタスクの範囲をより具体的な詳細まで含められるように広げています。そのうち、問題の解決に必要なステップについて考える時間は少なくなり、その分、プロセスの観点で考えることに、より多くの時間をかけられるようになってくるでしょう。この連載が進んで行ったときに、その例をいくつも紹介します。

まとめ

関数型プログラミングとは、具体的なツールのセットあるいは言語というよりは、考え方です。この連載第 1 回目の記事では、単純な設計の決定から、問題の意欲的な見直しに至るまで、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ました。そして単純な Java クラスを関数型に近くなるように作成し直した後、関数型プログラミングによる方法を、従来の命令型言語を使った方法とは区別するいくつかのトピックを掘り下げて説明しました。

この記事では、重要かつ広範囲にわたって適用される 2 つの考え方を紹介しました。1 つ目の考え方は、「ステップではなく、結果に重点を置く」というものです。関数型プログラミングは、問題を異なる方法で見せようとします。それは、ソリューションを組み立てるためのビルディング·ブロックが Java とは異なるためです。そして 2 つ目の考え方は、この連載を通して説明する、「開発者は一般的なタスクの詳細をプログラミング言語とランタイムに任せ、自分が作成しているプログラムに特有の問題に専念できるようにする」というものです。次回の記事では、引き続き関数型プログラミングの全般的な特徴に目を向け、関数型プログラミングが現在のソフトウェア開発にどのように適用されるかを検討します。

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。