関数型の考え方: 変換処理と最適化

さまざまな関数型の言語間の比較の続き

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2012年 11月 15日

各種の関数型の言語とフレームワークには、同じ抽象化と振る舞いでも、その名前が異なるものが数多くあります。連載「関数型の考え方」の今回の記事では、著者の Neal Ford が前回の記事で紹介したソリューションのアルゴリズムを改善し、キャッシングを追加することによってソリューションを最適化します。これにより、それぞれの言語やフレームワークで最適化に必要な変更がどのように提供されるの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。

関数型プログラミングのルーツは、数学とコンピューター・サイエンスにありますが、どちらも専門用語について確固たる見解があります。言語やフレームワークの設計者たちが、各自の好みの用語体系を作成したとしても、結局は基礎となるパラダイムにすでに決まった名前があることがわかります。関数型プログラミングのパラダイムを学ぶのが難しい理由は、使われている用語に一貫性がないためです。

前回の「実にさまざまな変換処理」では、素数を分類するという問題を取り上げ、そのソリューションを JVM 上で動作する関数型言語と Java 対応の 2 つの関数型フレームワークで実装しました。今回の記事ではその続きとして、前回の記事で使用したアルゴリズムをいくつかの方法で最適化し、それぞれの言語での最適化による変更を紹介します。前回と同じく、関数型のツールや言語によって異なる点を示しますが、今回は機能で使われている用語とその機能が使用できる状況に関する違いを明らかにします。具体的な機能の例として使用するのは、`map`、`filter`、`memoize` です。

Pure Java での素数分類子の最適化

ここで取り上げている問題は、ある数値が素数であるかどうかを判別することです。つまり、その対象とする数値の約数が 1 とその数値自身のみであるかどうかを判別します。この問題を解決するアルゴリズムはいくつかありますが、この記事では関数型プログラミングの世界におけるフィルタリングとマッピングについて説明することにしました。

前回の記事で素数を判別するためのアルゴリズムに採用した単純な手法では、最適なパフォーマンスで実行されるコードよりも、単純なコードであることの方が優先されました。今回は、何種類かの関数型の概念を用いてアルゴリズムを最適化します。さらに、同じ数値を分類するためにクラスを複数回呼び出すという状況に対して、各バージョンを最適化します。

リスト 1 に、素数を判別するために最初に作成した Java コードを記載します。

リスト 1. Java による素数分類子のオリジナル・バージョン

```
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;
    }
}
```

リスト 1 では、`getFactors()` メソッドが 2 から分類対象の数値までの約数候補を繰り返し処理しますが、これではかなり非効率です。約数は常にペアで出現することを考えてください。つまり、約数が 1 つ見つかれば、単純な除算によってその対となるもう 1 つの約数を突き止めることができます。したがって、対象の数値まですべての数値を繰り返し処理する代わりに、数値の平方根までを繰り返し処理すれば、約数のペアを取得することができます。このように改善した後の `getFactors()` メソッドを、リスト 2 に記載する素数分類子全体を改善したバージョンの中に示します。

リスト 2. Pure Java の最適化バージョン

```
public class PrimeNumber {
    private Integer number;
    private Map<Integer, Integer> cache;

    public PrimeNumber() {
        cache = new HashMap<Integer, Integer>();
    }

    public PrimeNumber setCandidate(Integer number) {
        this.number = number;
        return this;
    }

    public static PrimeNumber getPrime(int number) {
        return new PrimeNumber().setCandidate(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 (int i = 2; i < sqrt(number) + 1; i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

    public int sumFactors() {
        int sum = 0;
        if (cache.containsValue(number))
            sum = cache.get(number);
        else
            for (int i : getFactors())
                sum += i;
        return sum;
    }

    public boolean isPrime() {
        return number == 2 || sumFactors() == number + 1;
    }

}
```

リスト 2 の `getFactors()` メソッドでは、2 から対象の数値の平方根に 1 を足した値 (1 を足しているのは、丸め誤差に対処するためです) までを繰り返し処理し、約数をペアで取得します。このコードの場合、完全平方数に関するエッジ・ケースがあるため、`Set` を返すことが重要となります。例えば、数値 16 の平方根は 4 です。`getFactors()` メソッドの中で、`Set` ではなく `List` を使ったとしたら、リストに重複した 4 が生成されることになります。このようなエッジ・ケースを見つけるユニット・テストは存在します！

リスト 2 で行っているもう 1 つの最適化は、複数の呼び出しに関するものです。このコードの一般的な用途が、素数を見つけるために同じ数値を複数回にわたって評価することだとしたら、リスト 1 の `sumFactors()` メソッドが行う計算では非効率です。そこで、リスト 2 の `sumFactors()`

メソッドには、クラス全体のキャッシュを作成して、そこに計算済みの値を保持するようにしました。

2 つ目の最適化を実現するには少し曖昧なクラス設計が必要になるため、インスタンスがキャッシュの所有者として機能できるようにするにはステートフルにせざるを得ません。この点は改善することが可能ですが、その改善は以降の例ではささいなことなので、このバージョンでは無視します。

Functional Java での最適化

Functional Java (「参考文献」を参照) は、関数型の機能を Java に追加するフレームワークです。Functional Java のバージョンで最適化の影響を受けるメソッドには、`getFactors()` と `sumFactors()` があります。これらのメソッドのオリジナル (最適化されていない) バージョンをリスト 3 に記載します。

リスト 3. Functional Java の `getFactors()` メソッドと `sumFactors()` メソッドのオリジナル・バージョン methods

```
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);
}
```

リスト 3 の `getFactors()` メソッドは、1 から対象の数値に 1 を足した値 (Functional Java の `range` は、最後の値を範囲に含めないため) までの範囲を `isFilter()` メソッドでフィルタリングして、約数のリストに含めるかどうかを判別します。リスト 4 に、Functional Java による素数分類子の最適化バージョンを記載します。

リスト 4. Functional Java の最適化バージョン

```
import fj.F;
import fj.data.List;
import java.util.HashMap;
import java.util.Map;
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 FjPrimeNumber {
    private int candidate;
    private Map<Integer, Integer> cache;

    public FjPrimeNumber setCandidate(int value) {
        this.candidate = value;
        return this;
```

```
    }

    public FjPrimeNumber(int candidate) {
        this.candidate = candidate;
        cache = new HashMap<Integer, Integer>();
    }

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

    public List<Integer> getFactors() {
        final List<Integer> lowerFactors = range(1, (int) round(sqrt(candidate) + 1))
                .filter(new F<Integer, Boolean>() {
                    public Boolean f(final Integer i) {
                        return isFactor(i);
                    }
                });
        return lowerFactors.append(lowerFactors.map(new F<Integer, Integer>() {
            public Integer f(final Integer i) {
                return candidate / i;
            }
        }))
        .nub();
    }

    public int sumFactors() {

        if (cache.containsKey(candidate))
            return cache.get(candidate);
        else {
            int sum = getFactors().foldLeft(add, 0);
            cache.put(candidate, sum);
            return sum;
        }
    }

    public boolean isPrime() {
        return candidate == 2 || sumFactors() == candidate + 1;
    }
}
```

リスト 4 の `getFactors()` メソッドでは、最適化する前と同じ `range()` メソッドと `filter()` メソッドを前よりも選択的に使用しています。最初の範囲では、`filter()` メソッドをリスト 3 と同じように使用して、平方根までの約数を取得しますが、このメソッドの 2 行目で、Functional Java の `map()` メソッドを使用して平方根より大きい約数を生成します。`map()` メソッドはコレクションに含まれる各要素に関数を適用することにより、変換処理が行われたコレクションを返します。平方根より大きい約数を集めたこのリストが、平方根以下の約数 (`lowerFactors`) に追加されます。このリストは、Functional Java の `nub()` メソッドに対する最後のメソッド呼び出しによってセットに変換されるため、完全平方数の重複問題が軽減されます。

リスト 4 の `sumFactors()` の最適化では、リスト 2 の Pure Java バージョンと同様のキャッシュを使用します。つまり、Pure Java バージョンと同じく、クラスをステートフルにせざるを得ないということです。

Groovy での最適化

Groovy の `getFactors()` メソッドと `sumFactors()` メソッドのオリジナル・バージョンをリスト 5 に記載します。

リスト 5. Groovy の `getFactors()` メソッドと `sumFactors()` メソッドのオリジナル・バージョン

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

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

Groovy では、`findAll()` メソッドが数値の範囲をフィルタリングした後、`sumFactors()` メソッドが Groovy の `inject()` メソッドを使用して各要素にコード・ブロックを適用し、このリストを 1 つの要素にまで絞り込みます (このコード・ブロックは集約操作として各ペアの和をとるため、この 1 つの要素が和になります)。リスト 6 に、Groovy による素数分類子の最適化バージョンを記載します。

リスト 6. Groovy の最適化バージョン

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

class PrimeNumber {
  static def isFactor(potential, number) {
    number % potential == 0;
  }

  static def factors = { number ->
    def factors = (1..sqrt(number)).findAll { isFactor(it, number) }
    factors.addAll factors.collect { (int) number / it}
    factors.toSet()
  }

  static def getFactors = factors.memoize();

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

  static def isPrime(number) {
    number == 2 || sumFactors(number) == number + 1
  }
}
```

Functional Java バージョンと同じく、リスト 6 の `factors()` メソッドは、平方根を使用して約数を分割し、`toSet()` メソッドで結果のリストを 1 つのセットに変換します。Functional Java と Groovy との間の主な違いは、用語の違いです。Functional Java での `filter()` メソッドおよび `foldLeft()` メソッドは、それぞれ Groovy での `findAll()` と `inject()` メソッドと同義です。

リスト 6 の最適化ソリューションは、これまでの Java バージョンとは大幅に異なります。クラスにステートフルな性質を追加する代わりに、私が使用したのは Groovy の `memoize()` メソッドです。リスト 6 の `factors` メソッドは純粋関数であるため、パラメーターだけに依存し、それ以外の状態にはまったく依存しません。要件が満たされると、Groovy ランタイムは `memoize()` メソッドを使って値を自動的にキャッシュし、それによって `factors()` メソッドのキャッシュ・バージョン (`getFactors()` という名前のメソッド) が返されます。これは、関数型プログラミングでは、開発者が保守しなければならないメカニズム (例えばキャッシングなど) の数を削減できるこ

とを示す好例です。メモ化については、この連載の記事「関数型のデザイン・パターン: 第 1 回」で詳しく説明しています。

Scala での最適化

リスト 7 に、Scala での `factors()` メソッドと `sum()` メソッドのオリジナル・バージョンを記載します。

リスト 7. Scala の `factors()` および `sum()` メソッドのオリジナル・バージョン

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

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

リスト 7 のコードでは、名前が重要ではない引数に対して、便利な _ プレースホルダーを利用しています。リスト 8 に、Scala による素数分類子の最適化バージョンを記載します。

リスト 8. Scala の最適化バージョン

```
import scala.math.sqrt;

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

  def factors(number: Int) = {
    val lowerFactors = (1 to sqrt(number).toInt) filter (isFactor(number, _))
    val upperFactors = lowerFactors.map(number / _)
    lowerFactors.union(upperFactors)
  }

  def memoize[A, B](f: A => B) = new (A => B) {
    val cache = scala.collection.mutable.Map[A, B]()
    def apply(x: A): B = cache.getOrElseUpdate(x, f(x))
  }

  def getFactors = memoize(factors)

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

  def isPrime(number: Int) =
    number == 2 || sum(getFactors(number)) == number + 1
}
```

最適化された `factors()` メソッドは、これまでの例 (リスト 3 など) と同じ手法を Scala の構文に当てはめて使用しているので、簡単に理解できる実装となっています。

Scala には標準のメモ化機能はありませんが、今後、標準として追加されることが示唆されています。メモ化はさまざまな方法で実装することができます。簡単な実装は、Scala の組み込み可変マップと便利な `getOrElseUpdate()` メソッドを利用します。

Clojure での最適化

リスト 9 に、Clojure バージョンの `factors` メソッドと `sum-factors` メソッドを記載します。

リスト 9. `factors` メソッドと `sum-factors` メソッドのオリジナル・バージョン

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

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

最適化する前の他のバージョンと同様、オリジナル・バージョンの Clojure コードは、1 から対象の数値に 1 を足した値までの範囲をフィルタリングした後、Clojure の `reduce` 関数によって `+` 関数を各要素に適用することで和を求めています。リスト 10 に、Clojure による素数分類子の最適化バージョンを記載します。

リスト 10. Clojure の最適化バージョン

```
(ns primes)

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

(defn factors [n]
  (let [factors-below-sqrt (filter #(factor? n %) (range 1 (inc (Math/sqrt n))))
        factors-above-sqrt (map #(/ n %) factors-below-sqrt)]
    (concat factors-below-sqrt factors-above-sqrt)))

(def get-factors (memoize factors))

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

(defn prime? [n]
  (or (= n 2) (= (inc n) (sum-factors n))))
```

`factors` メソッドは、これまでの例 (リスト 3 など) で使用した最適化アルゴリズムと同じアルゴリズムを使用して、1 から平方根プラス 1 までの範囲をフィルタリングし、平方根以下の約数を取得します。その構文は、`(filter #(factor? n %) (range 1 (inc (Math/sqrt n))))` です。Clojureバージョンでは独自のシンボル (%) を無名パラメーターに使用します。この点は、リスト 8 の Scala バージョンと同様です。`#(/ n %)` 構文は、構文糖による `(fn [x] (/ n x))` の省略形として、無名関数となります。

Clojure には、Groovy バージョンと同じく、`memoize` 関数によって純粋関数をメモ化する機能が組み込まれているので、2 番目の最適化を実装するのは簡単なことです。

まとめ

今回の記事ではこの話題の締めくくりとして、関数型プログラミングにおける同じような概念とその組み込み機能が、言語やフレームワークによって異なる名前に転じてきたことを説明しました。関数の命名に関して言うと、Groovy は (例えば、一般的な `filter()` ではなく `findAll()` と名付けたり、`map()` ではなく `collect()` と名付けたりするなど) 風変わりな言語です。そして、キャッシングを簡単かつ安全に実装する上ではメモ化を使用できるかどうかが大きな違いをもたらします。

次回の記事では、関数型のさまざまな言語とフレームワークにおける遅延について詳しく探ります。

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。