関数型の考え方: Groovy に隠された関数型の機能、第 3 回

メモ化とキャッシング

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2012年 3月 01日

最近の動的言語には、平凡なタスクで開発者の手をわずらわせないように多くの関数型の機能が組み込まれています。今回の記事では、Groovy の関数レベルでキャッシングするメリットを探り、命令型の手法と対比させます。またキャッシングのタイプとして、メソッド内でキャッシュする場合と、外部でキャッシュする場合の 2 つを例に、命令型の手法と関数型の手法それぞれの利点と欠点を検討します。

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。

プログラミング言語が開発者に代わって処理する下位レベルの詳細が多くなればなるほど、コードにバグが入り込んだり、コードが複雑なものになったりする可能性は限られてきます (その典型的な例は、JVM でのガーベッジ・コレクションとメモリー管理です)。これまでの記事で強調してきたように、関数型言語 (および関数型の機能を備えた動的言語) の特徴の 1 つは、平凡な詳細の制御を選択的に言語とランタイムに任せられることです。例えば「Groovy に隠された関数型の機能、第 1 回」では、再帰によって、開発者が状態を維持管理する必要がなくなることを説明しました。今回の記事では、Groovy でのキャッシングによって、同じく状態を維持管理する必要がなくなることを説明します。

キャッシングは一般的な要件です (そして、なかなか見つかりにくいバグの原因でもあります)。オブジェクト指向の言語では、キャッシングは一般にデータ・オブジェクト・レベルで行われるため、開発者自らがキャッシングの管理をしなければなりません。一方、関数型言語の多くは、「メモ化」と呼ばれるメカニズムを使って、関数レベルでキャッシングを組み込みます。この記事では、キャッシングを関数で使用するケースとして、クラス内でキャッシュする場合と、外部

呼び出しによってキャッシュする場合について調べていきます。また、キャッシングを実装する方法として、ハンド·コーディングによって状態を維持管理する方法と、メモ化による方法について説明します。

メソッド・レベルのキャッシング

この連載を 1 回でも読んだことがあれば、数値分類子の問題 (連載の最初の記事で説明しました) についてはお馴染みのことでしょう。今回の記事で出発点とするのは、リスト 1 に記載する Groovy での Classifier です (これは、前回の記事で作成しました)。

リスト 1. Groovy での Classifier

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

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

  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 1 に示されている `factorsOf()` メソッドの内部アルゴリズムは、「関数型の観点で考える、第 2 回」で説明した手法を使って最適化することもできますが、今回の例では、関数レベルのキャッシングによって最適化することのほうに興味があります。

この `Classifier` クラスは数値を分類するためのクラスです。したがって、一般的な使用法では、同じ数値で複数のメソッドを実行して数値を分類することになります。その一例として、リスト 2 のコードを見てください。

リスト 2. 数値分類子の一般的な使用法

```
//...
if (Classifier.isPerfect(n)) print '!'
else if (Classifier.isAbundant(n)) print '+'
else if (Classifier.isDeficient(n)) print '-'
//...
```

現状の実装では、分類するためのメソッドを呼び出すごとに、約数の和の計算を繰り返さなければなりません。これは、クラス内でキャッシュする一例であり、それぞれの数値に対して

`sumOfFactors()` メソッドが何度も呼び出されるのが通常です。けれども、この一般的な使用ケースでは効率的な手法とは言えません。

和をキャッシュする

コードを効率化する 1 つの方法は、既に実行済みのコストのかかる処理の成果を利用することです。そこで、コストのかかる約数の和の生成処理を数値ごとに 1 度だけで済ませるようにしたいと思います。その目的で、計算結果を格納するキャッシュを作成します (リスト 3 を参照)。

リスト 3. 和をキャッシュする

```
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
    }
  }
  //... remainder of code unchanged
}
```

リスト 3 では、コンストラクター内に `sumCache` という名前のハッシュを作成しています。`sumOfFactors()` メソッドの中で、パラメーターの和がすでにキャッシュされているかどうかをチェックし、キャッシュされている場合には、その和を返します。和がキャッシュされていない場合には、コストのかかる計算を行って、その和をキャッシュしてから返します。

コードは複雑になっていますが、その成果は結果を見れば明らかです。記事のすべてのサンプル・コードで実行する一連のユニット・テストは、すべてリスト 4 に示すパターンに従っています。

リスト 4. テスト

```
class ClassifierTest {
  def final TEST_NUMBER_MAX = 1000
  def classifier = new ClassifierCachedSum()

  @Test
  void classifier_many_checks_without_caching() {
    print "Nonoptimized:               "
    def start = System.currentTimeMillis()
    (1..TEST_NUMBER_MAX).each {n ->
      if (Classifier.isPerfect(n)) print '!'
      else if (Classifier.isAbundant(n)) print '+'
      else if (Classifier.isDeficient(n)) print '-'
    }
    println "\n\t ${System.currentTimeMillis() - start} ms"
    print "Nonoptimized (2nd):        "
    start = System.currentTimeMillis()
    (1..TEST_NUMBER_MAX).each {n ->
      if (Classifier.isPerfect(n)) print '!'
      else if (Classifier.isAbundant(n)) print '+'
```

```
      else if (Classifier.isDeficient(n)) print '-'
    }
    println "\n\t ${System.currentTimeMillis() - start} ms"
  }
```

リスト 4 のテストを実行すると、その結果にはキャッシングが役立つことが示されます (リスト 5 を参照)。

リスト 5. 和をキャッシュした結果の分析

```
Test for range 1-1000
Nonoptimized:
  577 ms
Nonoptimized (2nd):
  280 ms
Cached sum:
  600 ms
Cached sum (2nd run):
  50 ms
```

リスト 5 に示されているように、1 回目の実行では、最適化されていないバージョン (リスト 1) の所要時間は 577 ミリ秒、キャッシュ・バージョンの所要時間は 600 ミリ秒でした。この 2 つのケースには、目立った違いはありません。一方、最適化されていないバージョンの 2 回目の実行では、結果は 280 ミリ秒になっています。1 回目と 2 回目の結果の差は、ガーベッジ・コレクションなどの環境要因によるものと考えられます。キャッシュ・バージョンの 2 回目の実行では、処理速度が劇的に改善され、その結果はわずか 50 ミリ秒です。キャッシュ・バージョンを 2 回目に実行するときには、値はすべてキャッシュに入れられています。したがって、2 回目ではハッシュからの読み取り速度を測定していることになります。最適化されていないバージョンとキャッシュ・バージョンとの違いは、1 回目の実行では取るに足らないものでしたが、2 回目の実行では顕著になって現れています。これは、外部キャッシングの一例です。つまり、コードを呼び出すと、キャッシュ内のすべての結果が使用されるため、2 回目の実行は極めて高速になります。

和をキャッシュすると、パフォーマンスに大幅な違いがもたらされますが、それには代償を伴います。それは、`ClassifierCachedSum` の中に純粋な静的メソッドを組み込めなくなることです。内部キャッシュは状態を表します。したがって、キャッシュを操作するすべてのメソッドを静的ではないメソッドにしなければなりません。そうなると、他の部分にも影響が波及します。何らかの Singleton ソリューション (「参考文献」を参照) を作成するという方法もありますが、その場合には複雑さも増してきます。また、キャッシュ変数を制御していることから、(例えば、テストを使用するなどして) 正確さを確実にする必要もあります。キャッシングによってパフォーマンスは向上するものの、コードに予想外の複雑さと保守の負担が加わるという代償を支払わなければなりません (「参考文献」を参照)。

何もかもキャッシュする

和をキャッシュすることで速度が大幅に向上するとしたら、再利用できる可能性のあるすべての中間結果をキャッシュしてみるのはどうでしょうか。それが、リスト 6 のコードでやろうとしていることです。

リスト 6. 何もかもキャッシュする

```
class ClassifierCached {
```

```
  private sumCache, factorCache

  ClassifierCached() {
    sumCache = [:]
    factorCache = [:]
  }

  def sumOfFactors(number) {
    sumCache.containsKey(number) ?
      sumCache[number] :
      sumCache.putAt(number, factorsOf(number).inject(0, {i, j -> i + j}))
  }

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

  def factorsOf(number) {
    factorCache.containsKey(number) ?
      factorCache[number] :
      factorCache.putAt(number, (1..number).findAll { i -> isFactor(number, i) })
  }

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

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

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

}
```

リスト 6 の `ClassifierCached` には、約数の和を対象とするキャッシュと、数値の約数を対象とするキャッシュの両方を追加しました。上記では、リスト 3 に記載されている冗長な構文を使う代わりに三項演算子を使用しています。三項演算子は、この例で驚くほどの表現力を発揮します。例えば `sumOfFactors()` メソッドの中では、三項演算子の条件の部分を使用してキャッシュ内に特定の数値があるかどうかをチェックします。Groovy では、メソッドの最後の行は、そのメソッドの戻り値です。したがって、その戻り値がキャッシュに格納されていれば、キャッシュ内の値が返されます。そうでない場合には、数値を計算して結果をキャッシュに入れてから、その値を返します (Groovy の `putAt()` メソッドが、値をハッシュに追加して返します)。リスト 7 に、このコードの結果を記載します。

リスト 7. 何もかもキャッシュした場合のテスト結果

```
Test for range 1-1000
Nonoptimized:
  577 ms
Nonoptimized (2nd):
  280 ms
Cached sum:
  600 ms
Cached sum (2nd run):
  50 ms
Cached:
  411 ms
Cached (2nd run):
  38 ms
```

何もかもキャッシュした場合 (これは、テスト実行ではまったく新しいクラスおよびインスタンス変数です) の実行時間は 1 回目では 411 ミリ秒で、キャッシュに結果が入れられた後の 2 回目では、わずか 38 ミリ秒にまで大幅に短縮されています。優秀な結果とは言え、この手法はスケーラビリティーに関してはそれほど優れていません。リスト 8 に、5,000 の数値をテストした場合の結果を記載します。キャッシュ・バージョンは、キャッシュの準備ができるまでは実行にかなり長い時間がかかりますが、その後の実行では読み取り時間が大幅に短縮されます。

リスト 8. 5,000 の数値を分類した場合のテスト結果

```
Test for range 1-5000
Nonoptimized:
  6199 ms
Nonoptimized (2nd):
  5064 ms
Cached sum:
  5962 ms
Cached sum (2nd run):
  93 ms
Cached:
  6494 ms
Cached (2nd run):
  41 ms
```

10,000 の数値を分類した場合の結果は、上記よりも悲惨になってきます (リスト 9 を参照)。

リスト 9. 10,000 の数値の分類 (試行)

```
Test for range 1-10000
Nonoptimized:
  43937 ms
Nonoptimized (2nd):
  39479 ms
Cached sum:
  44109 ms
Cached sum (2nd run):
  289 ms
Cached:
java.lang.OutOfMemoryError: Java heap space
```

リスト 9 に示されているように、キャッシング・コードに責任を持つ開発者は、コードの正確さと実行条件の両方に注意しなければなりません。これは、可変要素の好例です。つまり、開発者はコードの可変要素である状態を維持管理し、状態に対する影響を詳しく分析する必要があります。

メモ化

関数型プログラミングでは、再利用可能なメカニズムをランタイムに組み込むことによって、可変要素を最小限にしようと目指します。メモ化は、プログラミング言語に組み込まれた機能であり、再帰関数の戻り値の自動キャッシングを可能にします。言い換えると、メモ化は、私がリスト 3 とリスト 6 で作成したコードを自動的に作成します。メモ化は多くの関数型言語でサポートされており、Groovy でも最近、メモ化のサポートを追加しました (「参考文献」を参照)。

Groovy で関数をメモ化するには、関数をコード・ブロックとして定義した上で、`memoize()` メソッドを実行して関数を返します。すると、その関数の実行結果がキャッシュされます。

和をメモ化する

リスト 3 で行ったように `sumOfFactors()` のキャッシングを実装するために、`sumOfFactors()` メソッドをメモ化します (リスト 10 を参照)。

リスト 10. 和をメモ化する

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

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

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

 // remainder of code unchanged
}
```

リスト 10 では、`sumFactors()` メソッドをコード・ブロックとして作成しています (「`=`」とパラメーターの配置に注意してください)。これはかなり一般的なメソッドなので、どこかのライブラリーから取り込むという方法も考えられます。このメソッドをメモ化するために、`sumOfFactors` という名前を `memoize()` メソッド呼び出しとして関数参照で割り当てます。

メモ化したコードを実行すると、リスト 11 の結果になります。

リスト 11. 和をメモ化した場合のテスト結果

```
Test for range 1-1000
Nonoptimized:
  577 ms
Nonoptimized (2nd):
  280 ms
Cached sum:
  600 ms
Cached sum (2nd run):
  50 ms
Cached:
  411 ms
Cached (2nd run):
  38 ms
Partially Memoized:
  228 ms
Partially Memoized (2nd):
  60 ms
```

部分的なメモ化の 1 回目の実行結果は、最適化していない場合の 2 回目の実行結果とほぼ同じです。どちらの場合も、メモリーとその他の環境に関する問題が解決されていることから、同じような結果になるのは納得できます。けれども、部分的なメモ化を 2 回目に実行すると、和をキャッシュするために作成したコードに匹敵するほど劇的に速度が向上します。この高速化は、文字通り、元のコードを 2 行変更しただけで実現されています (具体的には、これまでの `sumOfFactors()` の代わりとなる `sumFactors()` メソッドをコード・ブロックとして作成し、こ

のコード・ブロックのメモ化したインスタンスを指すように `sumOfFactors()` を定義したことです)。

何もかもメモ化する

前に何もかもキャッシュしたように、結果を再利用できる可能性のあるすべての要素をメモ化してみましょう。その場合の分類子をリスト 12 に記載します。

リスト 12. 何もかもメモ化する

```
class ClassifierMemoized {
  def static dividesBy = { number, potential ->
    number % potential == 0
  }
  def static isFactor = dividesBy.memoize()
//  def static isFactor = dividesBy.memoizeAtMost(100)

  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
  }
}
```

リスト 6 に記載した何もかもキャッシュする場合と同様に、何もかもメモ化する場合にも利点と欠点があります。リスト 13 は、1,000 の数値を分類した場合の結果です。

リスト 13. 1,000 の数値に対し、何もかもメモ化した場合のテスト結果

```
Test for range 1-1000
Nonoptimized:
  577 ms
Nonoptimized (2nd):
  280 ms
Cached sum:
  600 ms
Cached sum (2nd run):
  50 ms
Cached:
  411 ms
Cached (2nd run):
  38 ms
Partially Memoized:
  228 ms
Partially Memoized (2nd):
  60 ms
```

```
Memoized:
  956 ms
Memoized(2nd)
  19 ms
```

何もかもメモ化した場合、1 回目の実行速度は遅くなりますが、以降の実行では、他のどの場合よりも処理時間が短縮されています。ただし、これは数値の数が少ない場合に限ります。リスト 8 でテストした命令型のキャッシング・ソリューションと同じく、数値の数が多くなると、パフォーマンスは急激に劣化します。実際、メモ化した場合に 5,000 の数値で実行すると、メモリー不足が発生します。命令型の手法を堅牢なものにするためには、保護手段を講じ、注意深く実行コンテキストを把握しなければなりません (これも、命令型の可変要素の例です)。メモ化の場合には、最適化は関数レベルで行われます。リスト 14 のメモ化の結果を見てください。

リスト 14. メモ化をチューニングした結果

```
Test for range 1-10000
Nonoptimized:
  41909 ms
Nonoptimized (2nd):
  22398 ms
Memoized:
  55685 ms
Memoized(2nd)
  98 ms
```

リスト 14 に記載しているのは、`memoize()` の代わりに `memoizeAtMost(1000)` メソッドを呼び出した結果です。メモ化をサポートする他の言語と同じように、Groovy には結果を最適化するのに役立つメソッドがあります。これらのメソッドを表 1 に記載します。

表 1. Groovy のメモ化メソッド

メソッド	説明
`memoize()`	キャッシュを使用したクロージャーを作成します。
`memoizeAtMost()`	サイズに上限を設けたキャッシュを使用したクロージャーを作成します。
`memoizeAtLeast()`	キャッシュ・サイズを自動的に調整し、サイズに下限を設けたキャッシュを使用したクロージャーを作成します。
`memoizeBetween()`	キャッシュ・サイズを自動的に調整し、サイズに上限と下限を設けたキャッシュを使用したクロージャーを作成します。

命令型の手法の場合、コードは開発者が所有しており、責任も開発者にあります。関数型言語では、標準的な構成体に適用できる汎用的な機構を (場合によっては代替関数という形でカスタマイズできるように) 作成します。関数は基本的な言語要素です。したがって、関数レベルで最適化すれば、高度な機能を代償なしで手に入れることができます。この記事で説明したメモ化を使用したコードは、少ない数の数値では、キャッシュを使用するように作成したコードよりもパフォーマンスに優れています。

まとめ

今回の記事では、この連載全体にわたって共通するテーマについて説明しました。それは、関数型プログラミングでは可変要素を最小限にすることで、物事を容易にするというものです。記事

では、同じ数値を複数のカテゴリーに対してテストするという、一般的な数値分類子の使用ケースに対する 2 つの異なるキャッシング・ソリューションを比較しました。キャッシュを使用するコードを作成するのは簡単ですが、その場合、コードにステートフル性と複雑さが加わります。メモ化などの関数型言語の機能を使用すれば、関数レベルでキャッシングを追加できるため、命令型のコードよりも優れた結果を実現することができます (しかも、事実上、コードを変更する必要もありません)。関数型プログラミングは可変要素をなくし、開発者が問題そのものの解決に力を注げるようにします。

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。