関数型の考え方: Groovy に隠された関数型の機能、第 2 回

メタプログラミングと関数型 Java を組み合わせる

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2012年 2月 03日

Groovy でメタプログラミングと関数型プログラミングを組み合わせると、強力な組み合わせになります。この記事では、Groovy に組み込まれている関数型の機能を利用するメソッドを、メタプログラミングによって `Integer` データ型に追加する方法を説明します。さらにメタプログラミングを使用して、Functional Java フレームワークに用意された豊富な関数型の機能をシームレスな方法で Groovy に組み込む方法も説明します。

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。

前回の記事では、Groovy に隠されたすぐに使える関数型の機能をいくつか紹介し、Groovy の基本構成要素を使って無限リストを作成する方法を説明しました。今回の記事でも引き続き、Groovy で行う関数型プログラミングについて探っていきます。

Groovy はマルチパラダイム言語であり、オブジェクト指向、メタプログラミング、そして関数型のプログラミング・スタイルをサポートしています。これらのプログラミング・スタイルは、そのほとんどが互いに直交しています (囲み記事「直交性」を参照)。メタプログラミングは、言語とその言語のコア・ライブラリーに機能を追加するために使用することができます。このメタプログラミングに関数型プログラミングを組み合わせれば、作成するコードをより関数型に近づけることも、サード・パーティーの関数型ライブラリーに機能を追加して、Groovy でのそのライブラリーの動作を改善することも可能です。この記事ではまず、Groovy の `ExpandoMetaClass` でクラスに機能を追加する方法を紹介し、それからこのメカニズムを利用して Groovy に Functional Java ライブラリー (「参考文献」を参照) を組み込む方法を説明します。

© Copyright IBM Corporation 2012

ExpandoMetaClass によるオープン・クラス

直交性

直交 (orthogonal) は、数学やコンピューター・サイエンスを含め、複数の分野で定義されています。数学の分野では、2 つのベクトルの成す角が 90 度である場合に直交していると言い、その 2 つのベクトルは互いに影響を及ぼし合わないことを意味します。コンピューター・サイエンスの分野では、直交コンポーネントとは互いに影響 (または副次効果) を及ぼさないコンポーネントのことです。これと同様に、Groovy における関数型プログラミングとメタプログラミングは、互いに干渉し合わないことから直交関係にあると言えます。つまり、メタプログラミングを使用したからといって、関数型の構成体を使用できなくなるわけではありません。その逆も然りです。関数型プログラミングとメタプログラムが直交するということは、連携できないという意味ではなく、互いに干渉し合わないというだけに過ぎません。

オープン・クラスは、Groovy のとりわけ強力な機能の 1 つです。オープン・クラスでは、既存のクラスを再オープンして機能を追加または削除することができます。これは、サブクラス化とは異なります。サブクラス化は、既存の機能から新しい機能を派生させる方法ですが、オープン・クラスは `String` などのクラスを再オープンして、新しいメソッドをクラスに追加するために使用することができます。ライブラリーをテストする際には、この機能を多用して `Object` クラスに検証メソッドを追加することから、今ではアプリケーションのすべてのクラスに検証メソッドが含まれるようになっています。

Groovy でのオープン・クラスの手段には、カテゴリーと `ExpandoMetaClass` の 2 つがあります (「参考文献」を参照)。この記事の例にはどちらを使用することもできますが、ここでは構文が多少単純であるという理由から、`ExpandoMetaClass` を使用することにしました。

この連載を続けて読まれている方は、ずっと取り上げ続けている数値分類子の例はお馴染みのことでしょう。Groovy での完全な `Classifier` (リスト 1 を参照) には、Groovy 固有の関数型構成体が使用されます。

リスト 1. Groovy での完全な `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
  }
```

```
  static def nextPerfectNumberFrom(n) {
    while (!isPerfect(++n));
    n
  }
}
```

この Groovy でのメソッドの実装方法についてわからない点がある場合は、連載のこれまでの記事を参照してください (特に、「連結と合成、第 2 回」と「Groovy に隠された関数型の機能、第 1 回」)。上記のクラスのメソッドを使用するには、「通常の」関数型の方法で (つまり `Classifier.isPerfect(7)` のようにして) メソッドを呼び出すことができます。その一方で、メタプログラミングを使用することで、これらのメソッドを直接 `Integer` クラスに「連結」し、ある数値がどのカテゴリーに属しているのかを、その数値に「問い合わせる」ことができます。

上記に示されているメソッドを `Integer` クラスに追加するために使用しているのは、Groovy がそれぞれのクラスに事前定義している `metaClass` プロパティーです。(リスト 2 を参照)。

リスト 2. `Integer` に分類を追加する

```
Integer.metaClass.isPerfect = {->
  Classifier.isPerfect(delegate)
}

Integer.metaClass.isAbundant = {->
  Classifier.isAbundant(delegate)
}

Integer.metaClass.isDeficient = {->
  Classifier.isDeficient(delegate)
}
```

メタプログラミング・メソッドの初期化

メタプログラミングを使用したメソッドは、最初の呼び出しが試行されるよりも前に追加する必要があります。メソッドを初期化するのに最も安全な場所は、そのメソッドを使用するクラスの静的イニシャライザーの中です (静的イニシャライザーは、そのクラスの他のイニシャライザーよりも前に実行されることが保証されているため)。けれども、複数のクラスにメソッドを追加しなければならない場合、この方法では複雑さが増してきます。メタプログラミングを多用するアプリケーションは、最終的にはブートストラップ・クラスを使用して、適切なタイミングで初期化が行われるようにするのが通常です。

リスト 2 では、`Classifier` の 3 つのメソッドを `Integer` に追加しています。これで、Groovy のすべての整数がこれらのメソッドを使えるようになりました (Groovy にはプリミティブ・データ型の概念がありません。Groovy では定数でさえも、`Integer` を基本データ型として使用します)。各メソッドを定義するコード・ブロックの中では、事前定義された `delegate` パラメーターにアクセスします。この delegate は、クラスのそのメソッドを呼び出しているオブジェクトの値を表すパラメーターです。

メタプログラミング・メソッドを初期化した後 (囲み記事「メタプログラミング・メソッドの初期化」を参照)、カテゴリーについて数値に「問い合わせ」ます (リスト 3 を参照)。

リスト 3. メタプログラミングを使用して数値を分類する

```
@Test
void metaclass_classifiers() {
  def num = 28
  assertTrue num.isPerfect()
  assertTrue 7.isDeficient()
  assertTrue 6.isPerfect()
  assertTrue 12.isAbundant()
}
```

リスト 3 は、新規に追加されたメソッドが変数と定数の両方で実行できることを示しています。こうなれば、特定の数値の分類を (場合によっては列挙として) 返すメソッドを `Integer` に追加するのは簡単な話です。

既存のクラスに新しいメソッドを追加すること自体は、そのメソッドが呼び出すコードが厳格な関数型だとしても、関数型に特有のことではありません。けれども、メソッドをシームレスに追加できるということは、Functional Java ライブラリーなどのサード・パーティー・ライブラリーを簡単に組み込めるということであり、それによって明らかな関数型の機能が追加されるということです。Functional Java ライブラリーは、連載の「関数型の観点で考える、第 2 回」で数値分類子を実装するために使用しました。今回はこのライブラリーを使用して、完全数の無限ストリームを作成します。

メタプログラミングによるデータ型のマッピング

Groovy は基本的に Java の方言であるため、Functional Java のようなサード・パーティー・ライブラリーを組み込むのは簡単ですが、メタプログラミングでデータ型のマッピングをすれば、サード・パーティー・ライブラリーがより緊密に Groovy に組み込まれ、この 2 つのつなぎ目が目立たなくなります。Groovy には (`Closure` クラスを使用した) ネイティブのクロージャー・タイプがありますが、Functional Java にはクロージャーという贅沢な機能はまだないため (Functional Java は、Java 5 の構文に依存しています)、コード作成者は `f()` メソッドが含まれる汎用 `F` クラスと Generics を使用しなければなりません。しかし Groovy の `ExpandoMetaClass` を使用すれば、メソッドとクロージャー・タイプをマッピングするメソッドを作成することで、この両者の違いを解決することができます。

ここでは、Functional Java に含まれる `Stream` クラスにマッピングを追加します。このクラスは無限リストを抽象化しますが、Functional Java の `F` インスタンスを渡す代わりに Groovy のクロージャーを渡せるようにしたいので、`Stream` クラスに多重定義したメソッドを追加して、クロージャーを `F` の `f()` メソッドにマッピングします (リスト 4 を参照)。

リスト 4. `ExpandoMetaClass` を使用してデータ型をマッピングする

```
Stream.metaClass.filter = { c -> delegate.filter(c as fj.F) }
//    Stream.metaClass.filter = { Closure c -> delegate.filter(c as fj.F) }
Stream.metaClass.getAt = { n -> delegate.index(n) }
Stream.metaClass.getAt = { Range r -> r.collect { delegate.index(it) } }
```

最初の行で `Stream` に作成する `filter()` メソッドが、クロージャー (コード・ブロックの `c` パラメーター) を受け付けます。2 行目 (コメント行) は、`Closure` の型宣言が追加されている点を除けば、最初の行と同じです。型宣言の追加によって Groovy がコードを実行する方法に影響を及ぼすことはありませんが、ドキュメントのことを考えると 2 行目のほうが望ましいかもしれません。

コード・ブロックの本体は、`Stream` の既存の `filter()` メソッドを呼び出して、Groovy のクロージャーを Functional Java の `fj.F` クラスにマッピングします。このマッピングを行うために使用しているのは、いくらか魔法がかった Groovy の `as` 演算子です。

Groovy の `as` 演算子はクロージャーをインターフェース定義に無理やり当てはめて、インターフェースに必要なメソッドにクロージャー・メソッドをマッピングできるようにします。一例として、リスト 5 のコードを見てください。

リスト 5. `as` を使用して軽量のイテレーターを作成する

```
def h = [hasNext : { println "hasNext called"; return true},
         next : {println "next called"}] as Iterator

h.hasNext()
h.next()
println "h instanceof Iterator? " + (h instanceof Iterator)
```

リスト 5 の例では、2 つの名前と値のペアでハッシュを作成しています。それぞれの名前は文字列で (Groovy では、ハッシュのキーはデフォルトが文字列型なので、二重引用符で括る必要はありません)、値はコード・ブロックです。`as` 演算子がこのハッシュを `Iterator` インターフェースにマッピングするには、`hasNext()` メソッドと `next()` メソッドの両方が必要になります。このマッピングを行った後は、ハッシュをイテレーターとして扱えるようになるため、上記リストの最後の行は `true` を出力します。インターフェースにメソッドが 1 つしかない場合、またはインターフェースのすべてのメソッドに単一のクロージャーをマッピングする場合には、ハッシュを使わずに、`as` を直接使用してクロージャーを関数にマッピングすることができます。リスト 4 の最初の行に話を戻すと、ここで渡されるクロージャーは、メソッドが 1 つしかない `F` クラスにマッピングされます。リスト 4 で 2 つの `getAt` メソッド (一方は数値を受け付けており、もう一方は `Range` を受け付けています) をそれぞれにマッピングしている理由は、`filter` が動作するには、この 2 つのメソッドが必要だからです。

この新たにメソッドを追加した `Stream` を使用すれば、無限数列を扱うことができます (リスト 6 を参照)。

リスト 6. Groovy で Functional Java の無限ストリームを使用する

```
@Test
void adding_methods_to_fj_classes() {

  def evens = Stream.range(0).filter { it % 2 == 0 }
  assertTrue(evens.take(5).asList() == [0, 2, 4, 6, 8])
  assertTrue(evens[3..6] == [6, 8, 10, 12])
}
```

リスト 6 では、整数をクロージャー・ブロックでフィルタリングして、0 から始まる偶数の無限リストを作成しています。無限数列をまとめて一度に取得することはできないため、必要な要素の数だけ `take()` を実行する必要があります。リスト 6 の残りの部分では、ストリームがどのように機能するかを実演するアサーションのテストが示されています。

Groovy での無限ストリーム

前回の記事では、Groovy で無限の遅延リストを実装する方法を説明しましたが、今回は手作業で作成する代わりに、Functional Java による無限数列を使って実装してみます。

完全数の無限 Stream を作成するには、Groovy クロージャーが認識されるようにするために、さらに 2 つのメソッド・マッピングが Stream に必要になります (リスト 7 を参照)。

リスト 7. 完全数のストリーム用に追加する 2 つのメソッド・マッピング

```
Stream.metaClass.asList = { delegate.toCollection().asList() }
Stream.metaClass.static.cons = { head, closure -> delegate.cons(head, closure as fj.P1) }
// Stream.metaClass.static.cons =
//  { head, Closure c -> delegate.cons(head, ['_1':c] as fj.P1)}
```

リスト 7 では、Functional Java ストリームをリストに変換しやすくするために、asList() 変換メソッドを作成しています。上記で実装しているもう 1 つのメソッドは、多重定義した cons() メソッドです。これは、新しいリストを作成する Stream のメソッドです。通常、無限リストを作成するときには、データ構造にリストの先頭要素、そして後続要素としてクロージャー・ブロックを含めます。このクロージャーが呼び出されると、クロージャーによって次の要素が生成されるという仕組みです。この Groovy の完全数のストリームでは、cons() が Groovy のクロージャーを受け付けられることを Functional Java に認識させなければなりません。

as を使用して、複数のメソッドを持つインターフェースに単一のクロージャーをマッピングすれば、そのインターフェースのどんなメソッドを呼び出しても、そのクロージャーが実行されるようになります。このような形の単純なマッピングは、Functional Java クラスのほとんどの場合に有効に機能しますが、いくつかのメソッド・マッピングでは fj.F クラスのメソッドではなく、fj.P1 クラスのメソッドを使わなければなりません。そのような場合でも、その他のメソッドは fj.P1 クラスの他のメソッドにマッピングされることはないことから、単純なマッピングで切り抜けられるケースはあります。その一方、より高い精度が求められる場合には、リスト 7 のコメント行に示されている複雑なマッピングを使用して、クロージャーにマッピングされた _1() メソッドでハッシュを作成する必要があります。一見、奇異に見えるこの _1() メソッドは最初の要素を返す fj.P1 クラスの標準メソッドです。

メタプログラミングで Stream のメソッドをマッピングすれば、後はリスト 1 の Classifier を使って完全数の無限ストリームを作成することができます (リスト 8 を参照)。

リスト 8. Functional Java と Groovy を使用した完全数の無限ストリーム

```
import static fj.data.Stream.cons
import static com.nealford.ft.metafunctionaljava.Classifier.nextPerfectNumberFrom

def perfectNumbers(num) {
  cons(nextPerfectNumberFrom(num), { perfectNumbers(nextPerfectNumberFrom(num))})
}

@Test
void infinite_stream_of_perfect_nums_using_functional_java() {
  assertEquals([6, 28, 496], perfectNumbers(1).take(3).asList())
}
```

コードが冗長にならないように、Functional Java の cons() にも、Classifier に作成した nextPerfectNumberFrom() メソッドにも静的インポートを使用しています。perfectNumbers() メソッドは完全数の無限数列を返すために cons() メソッドを実行しますが、これによって返される数列では perfectNumbers() メソッドにシードとして渡された数よりも大きい最初の完全数が 1 番目の要素となり、クロージャー・ブロックが 2 番目の要素となります。そしてこのクロー

ジャー・ブロックが 1 番目の要素として次の完全数を、2 番目の要素としてさらに次の完全数を計算するためのクロージャーを返すことで、無限数列が実現されます。テストでは、1 から始まる完全数のストリームを生成し、次の 3 つの完全数を取得して、この 3 つの数値がリストに一致することをアサートしています。

まとめ

開発者がメタプログラミングのことを考えるときには、自分が作成したコードのことだけを考え、他の誰かが作成したコードに機能を追加するという手段は考えないものです。Groovy では、`Integer` のような組み込みクラスに新しいメソッドを追加できるだけでなく、Functional Java などのサード・パーティー・ライブラリーにも新規メソッドを追加することができます。メタプログラミングと関数型プログラミングの組み合わせは、極めて少ないコードで大きな力をもたらす上に、シームレスなコードにすることができます。

Functional Java クラスは、Groovy から直接呼び出すこともできますが、このライブラリーのビルディング・ブロックの大半は真のクロージャーと比べると手際の良さに欠けます。メタプログラミングを使用して Functional Java のメソッドをマッピングし、これらのメソッドが使い勝手の良い Groovy のデータ構造を認識できるようにすれば、Functional Java と Groovy 両方の優れた面を活かすことができます。Java がネイティブのクロージャー・タイプを定義するまでは、開発者はタイプの異なる言語の間で、度々このような多言語マッピングを行うことになります。バイトコード・レベルでは、Groovy のクロージャーと Scala のクロージャーの間にも違いがあります。Java に標準が設けられた暁には、これらの言語間でのやりとりがランタイムのレベルにまで引き下げられ、この記事で説明したようなマッピングの必要がなくなるはずですが、その日が来るまでは、このようなマッピングが簡潔ながらも強力なコードの作成に役立ちます。

次回の記事では、関数型プロプログラミングがランタイムで実現可能な最適化を取り上げ、Groovy でのメモ化の例を紹介します。

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。

© Copyright IBM Corporation 2012
(www.ibm.com/legal/copytrade.shtml)
商標
(www.ibm.com/developerworks/jp/ibm/trademarks/)