関数型の考え方: 関数型のデザイン・パターン、第 1 回

関数型の世界でのパターンの現れ方

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2012年 4月 05日

一般に関数型プログラミングにはデザイン・パターンが存在しないと考えられていますが、関数型プログラミングにもデザイン・パターンは存在します。ただし、関数型プログラミングにおけるデザイン・パターンは、見た目も振る舞いもオブジェクト指向プログラミングにおけるデザイン・パターンとは異なることがあります。連載「関数型の考え方」の今回の記事では、Neal Ford が関数型パラダイムでのパターンの現れ方を検討し、ソリューションによる違いを説明します。

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。

関数型の世界を代表する開発者のなかには、デザイン・パターンの概念には不備があるため、関数型プログラミングには必要ないと主張する人々がいます。その主張は、狭義のデザイン・パターンには当てはまるかもしれませんが、内容的にはデザイン・パターンの使用方法に関するものというよりは、デザイン・パターンのセマンティクスに関するものと言えます。デザイン・パターンの概念 (名前を付けて分類された、一般的な問題に対するソリューション) は関数型でも変わりませんが、パラダイムによってはパターンが異なる形で現れることがあります。関数型の世界ではビルディング・ブロックにしても、問題に対する取り組み方にしても通常のプログラミングとは異なるため、従来の Gang of Four のパターン (「参考文献」を参照) のなかには、関数型には適用されないパターンもあれば、同じ問題に対するパターンがまったく異なる方法を取ることもあります。今回と次回の記事で 2 回にわたり、従来のデザイン・パターンのいくつかを取り上げて調査し、関数型の考え方でこれらのパターンを見直します。

関数型プログラミングの世界では一般に、従来のデザイン・パターンが以下の 3 つのいずれかの形として現れます。

- パターンが言語に組み込まれた形。
- パターンのソリューションは関数型パラダイムにも存在するが、実装の詳細が異なるという形。
- ソリューションが、他の言語や他のパラダイムにはない機能を使用して実装されるという形 (例えば、メタプログラミングを使用する多くのソリューションは簡潔で洗練されたものになりますが、Java ではメタプログラミングを使えません)。

今回から 2 回にわたって、上記の 3 つのケースについて順に調査していきます。この記事ではまず、よく知られているパターンを取り上げます。そのほとんどは、最近の言語に完全に、あるいは部分的に組み込まれています。

ファクトリーとカリー化

カリー化は、多くの関数型言語に共通する機能です。数学者の Haskell Curry (Haskell プログラミング言語の名前の由来でもあります) にちなんで名付けられた、このカリー化という手法は、複数の引数を取る関数を変換して、1 つの引数のみを取る関数のチェーンとして呼び出せるようにします。カリー化に密接に関係する手法には、部分適用があります。部分適用は、ある関数の 1 つまたは複数の引数に固定値を割り当てることで、元の関数よりもアリティー (関数に渡される引数の個数) の値が小さい別の関数を生成するという手法です。この 2 つの手法については、「関数型の観点で考える、第 3 回」で説明しました。

デザイン・パターンのコンテキストでは、カリー化は関数のファクトリーの役割を果たします。関数型プログラミング言語に共通する機能の 1 つに、関数を他のあらゆるデータ構成体として機能することを可能にする、第一級 (あるいは高階) 関数があります。第一級関数のおかげで、何らかの基準に応じて他の関数を返す関数を簡単に作成できるわけですが、これはファクトリーの本質です。例えば、2 つの数値を加算する汎用的な関数がある場合、カリー化をファクトリーとして使用することで、常に一方の数値をその引数と加算する関数を作成することができます。これが、リスト 1 の Groovy で実装されているインクリメンターです。

リスト 1. 関数ファクトリーとしてのカリー化

```
def adder = { x, y -> return x + y }
def incrementer = adder.curry(1)

println "increment 7: ${incrementer(7)}" // prints "increment 7: 8"
```

リスト 1 では、1 つ目の引数を `1` としてカリー化し、引数を 1 つ取る関数を返します。要するに、関数ファクトリーを作成したということです。

使用している言語がこの類の振る舞いをネイティブにサポートする場合、その振る舞いが他のものに対するビルディング・ブロックとして、対象の大小を問わずに使用される傾向があります。例えば、リスト 2 に記載する Scala の例を見てください。

リスト 2. Scala で何気なく使用されているカリー化

```
object CurryTest extends Application {

  def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

  def dividesBy(n: Int)(x: Int) = ((x % n) == 0)

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
  println(filter(nums, dividesBy(2)))
  println(filter(nums, dividesBy(3)))
}
```

リスト 2 のコードは、Scala のドキュメント (「参考文献」を参照) で紹介されている再帰とカリー化の両方を使用したサンプル・コードの 1 つです。`filter()` メソッドは、パラメーター `p` を使用して再帰的に整数のリストをフィルタリングします。`p` は、述語関数 (関数型の世界において、ブール関数を表す一般的な用語) です。`filter()` メソッドは渡されたリストが空であるかどうかを調べ、空の場合には単純にそのままリストを返します。渡されたリストが空でなければ、述語関数をそのリストの先頭の要素 (`xs.head`) に適用し、フィルタリングの結果として生成される新しいリストにその要素を含めるべきかどうかを調べます。述語関数によって含めるべきであると判断されると、その要素を先頭要素とし、渡されたリストの残りの要素をフィルタリングした結果を後続要素とする新たなリストが返されます。含めるべきでないと判断されると、渡されたリストの残りの要素をフィルタリングした結果だけが返されます。

リスト 2 がパターンの観点から興味深い点は、`dividesBy()` メソッドで何気なくカリー化が使用されていることです。`dividesBy()` は 2 つの引数を取り、2 番目の引数が 1 番目の引数で割り切れるかどうかによって、`true` または `false` を返します。ただし、このメソッドが `filter()` メソッドの呼び出しの一部として呼び出される場合には、呼び出しには 1 つの引数しか使用されず、カリー化された関数となります。`filter()` メソッドの中では、この関数が述語関数として使用されます。

この例は、(記事の冒頭で説明した) 関数型プログラミングでデザイン・パターンが現れる 3 つの形のうち、最初の 2 つを表しています。1 つ目の形では、カリー化が言語あるいはランタイムに組み込まれるため、関数ファクトリーの概念が浸透していることから、追加の構成体は必要ありません。2 つ目の形は、私が説明したい異なる実装についてのポイントを示しています。一般的な Java プログラマーが、リスト 2 のような方法でカリー化を使用することは決してないはずです。つまり、ここではコードを実際に移植可能なコードにはしておらず、汎用的な関数から特定の場合にのみ対応した関数を作成することも、もちろん考えていません。実際、命令型言語を使用する開発者のほとんどは、デザイン・パターンをこのように使用しようと考えることはないはずです。汎用的な `dividesBy()` メソッドから特定の場合にのみ対応した dividesBy() メソッドを作成することは小さな問題のように思えますが、デザイン・パターンは (ほとんど構成体に頼って問題を解決するため、大量にオーバーヘッドが実装されることになり)、大きな問題に対するソリューションであるように思えるからです。ただし、カリー化をこのように使用したからと言って、カリー化という名前以外の特別な名前を正式な名前として用いてよいということにはなりません。

第一級関数とデザイン・パターン

第一級関数を用いると、多くの一般的に使用されているデザイン・パターンが大幅に単純化されます (機能を移植可能にするためのオブジェクト・ラッパーが必要なくなるため、Command パターンに至ってはその存在自体が不要になります)。

Template Method パターン

第一級関数を使用すると、使われる見込みのない構成体が削除されるので、Template Method パターン (「参考文献」を参照) の実装はより単純なものになります。Template Method パターンはメソッドのなかでアルゴリズムのスケルトンを定義した上で、一部のステップをサブクラスに任せ、アルゴリズムの構造を変えることなく、これらのステップをサブクラスに定義させます。リスト 3 に、Groovy での標準的な Template Method パターンの実装を記載します。

リスト 3. 標準的な Template Method パターンの実装

```
abstract class Customer {
  def plan

  def Customer() {
    plan = []
  }

  def abstract checkCredit()
  def abstract checkInventory()
  def abstract ship()

  def process() {
    checkCredit()
    checkInventory()
    ship()
  }
}
```

リスト 3 の `process()` メソッドは、`checkCredit()`、`checkInventory()`、および `ship()` メソッドを呼び出します。この 3 つは抽象メソッドであるため、その定義はサブクラスによって指定されることになります。

第一級関数は、他のあらゆるデータ構成体として機能できることから、リスト 3 のサンプル・コードは、コード・ブロックを使用してリスト 4 のように再定義することができます。

リスト 4. 第一級関数を使用した Template Method パターン

```
class CustomerBlocks {
  def plan, checkCredit, checkInventory, ship

  def CustomerBlocks() {
    plan = []
  }

  def process() {
    checkCredit()
    checkInventory()
    ship()
  }
}
```

```
class UsCustomerBlocks extends CustomerBlocks{
  def UsCustomerBlocks() {
    checkCredit = { plan.add "checking US customer credit" }
    checkInventory = { plan.add "checking US warehouses" }
    ship = { plan.add "Shipping to US address" }
  }
}
```

リスト 4 では、アルゴリズムのステップは単なるクラスのプロパティーとなっており、他のあらゆるプロパティーと同じように値を指定できるようになっています。これは、言語の機能が実装の詳細のほとんどを組み込んでいる一例です。このパターンについては、問題に対するソリューション (ステップを後続のハンドラーに任せるというソリューション) として説明する価値がまだありますが、その実装は単純化されます。

上記の 2 つのソリューションは、等価なものではありません。リスト 3 に記載した「従来の」Template Method パターンの例では、抽象クラスが従属メソッドを実装するためにはサブクラスが必要です。もちろん、サブクラスが空のメソッド本体を作成するだけの場合もありますが、抽象クラスのメソッド定義はある種のドキュメンテーションの形を取ることで、サブクラスの作成者にそのことを考慮に入れるように認識させます。一方、柔軟性が要求されるような状況には、メソッド宣言の厳格さは適さない場合があります。そのような場合には、例えば処理を行うために必要な複数のメソッドのリストを引数に取るような `Customer` クラスを作成することができます。

コード・ブロックを扱うための機能が充実している言語は、開発者にとって使いやすいものになります。例えば、サブクラスの作成者がステップのいくつかをスキップできるようにしたい場合を考えてみてください。Groovy には、オブジェクトのメソッドを呼び出す前に、そのオブジェクトがヌルではないことを確実にし、アクセスによって例外が発生するのを防ぐための特殊な演算子 (`?.`) があります。一例として、リスト 5 の `process()` 定義を見てください。

リスト 5. コード・ブロックの呼び出しに例外の発生を防ぐための演算子を追加する

```
def process() {
  checkCredit?.call()
  checkInventory?.call()
  ship?.call()
}
```

リスト 5 では、サブクラスを実装する誰もが、どの子メソッドを指定することも可能となります。その場合、指定した以外の子メソッドはブランクのままにしておいても支障をきたすことはありません。

Strategy パターン

第一級関数によって単純になる、もう 1 つのよく使用されるデザイン・パターンは、Strategy パターンです。Strategy パターンはアルゴリズムのファミリーを定義し、それぞれのアルゴリズムをカプセル化して互いに置き換えられるようにします。アルゴリズムを使用するクライアントとは独立してアルゴリズムを変更できるようにする Strategy パターンでは、第一級関数によってストラテジーを容易に作成および操作することができます。

従来の Strategy パターンでは、数値の積を計算するための実装はリスト 6 のようになります。

リスト 6. Strategy パターンを使用した 2 つの数字の積

```
interface Calc {
  def product(n, m)
}

class CalcMult implements Calc {
  def product(n, m) { n * m }
}

class CalcAdds implements Calc {

  def product(n, m) {
    def result = 0
    n.times {
      result += m
    }
    result
  }
}
```

リスト 6 には、2 つの数値の積のインターフェースを定義しました。このインターフェースの実装には、2 つの異なる具象クラス (ストラテジー) を使用しています。1 つは乗算を使用したクラス、もう 1 つは加算を使用したクラスです。これらのストラテジーをテストするために、リスト 7 のテスト・ケースを作成しました。

リスト 7. 積のストラテジーのテスト

```
class StrategyTest {
  def listOfStrategies = [new CalcMult(), new CalcAdds()]

  @Test
  public void product_verifier() {
    listOfStrategies.each { s ->
      assertEquals(10, s.product(5, 2))
    }
  }
}
```

リスト 7 から予想できるように、どちらのストラテジーも同じ値を返します。コード・ブロックを第一級関数として使用することで、前のサンプル・コードで使用されている定型部分の大半を取り除くことができます。今度は、リスト 8 に記載する指数のストラテジーを見てください。

リスト 8. 定型部分を減らした指数のテスト

```
@Test
public void exp_verifier() {
  def listOfExp = [
      {i, j -> Math.pow(i, j)},
      {i, j ->
        def result = i
        (j-1).times { result *= i }
        result
      }]

  listOfExp.each { e ->
    assertEquals(32, e(2, 5))
    assertEquals(100, e(10, 2))
    assertEquals(1000, e(10, 3))
  }
}
```

リスト 8 では指数のストラテジーとして、Groovy コード・ブロックを使用した 2 つのストラテジーをインラインで直接定義しました。Template Method パターンの例と同じく、形式と引き換えに便利さを取っているというわけです。従来の手法に従うと、各ストラテジーを名前と構成体で囲まなければなりません。場合によっては、この形が望ましいこともあります。注意すべき点として、リスト 8 のコードには、より厳しい安全手段を追加するという選択肢がある一方、従来の手法が課す制約を迂回するのは簡単ではありません。これは、関数型プログラミングとデザイン・パターンについての議論というよりも、動的であるか、静的であるかの議論です。

第一級関数の存在によって影響を受けるパターンは、ほとんどが言語に組み込まれているパターンの例です。次のセクションでは、セマンティクスを維持する一方、実装が変わるパターンについて説明します。

Flyweight パターンとメモ化

Flyweight パターンは、共有という手段で、粒度の細かい多数のオブジェクト参照をサポートする最適化手法です。このパターンでは、オブジェクトのプールを常に使用できるようにして、特定のビューへの参照をプールに作成します。Flyweight パターンが使用するのは、正準オブジェクトの考え方です。つまり、1 つの代表オブジェクトが、そのタイプの他のすべてのオブジェクトを表わします。例えば、特定の消費者向け製品があるとすると、その製品の正準なバージョンがそのタイプのすべての製品を表すといった具合です。アプリケーションでは、ユーザーごとに製品のリストを作成する代わりに、正準製品のリストを 1 つ作成し、各ユーザーはそのリスト内にそれぞれの製品への参照を持つようにします。

例えば、リスト 9 のクラスは、コンピューターのタイプをモデル化しています。

リスト 9. コンピューターのタイプをモデル化する単純なクラス

```
class Computer {
  def type
  def cpu
  def memory
  def hardDrive
  def cd
}

class Desktop extends Computer {
  def driveBays
  def fanWattage
  def videoCard
}

class Laptop extends Computer {
  def usbPorts
  def dockingBay
}

class AssignedComputer {
  def computerType
  def userId

  public AssignedComputer(computerType, userId) {
    this.computerType = computerType
    this.userId = userId
  }
}
```

すべてのコンピューターの仕様が同じである場合、これらのクラスの中に、例えばユーザーごとの新しい `Computer` インスタンスを作成するのでは非効率的です。そこで、`AssignedComputer` がコンピューターをユーザーに関連付けます。

このコードをさらに効率的にする一般的な方法は、Factory パターンと Flyweight パターンを組み合わせることです。そこでリスト 10 に記載する、正準なコンピューター・タイプを生成するためのシングルトン・ファクトリーについて検討してみましょう。

リスト 10. Flyweight パターンを適用したコンピューター・インスタンスに対するシングルトン・ファクトリー

```
class ComputerFactory {
  def types = [:]
  static def instance;

  private ComputerFactory() {
    def laptop = new Laptop()
    def tower = new Desktop()
    types.put("MacBookPro6_2", laptop)
    types.put("SunTower",  tower)
  }

  static def getInstance() {
    if (instance == null)
      instance = new ComputerFactory()
    instance
  }

  def ofType(computer) {
    types[computer]
  }
}
```

`ComputerFactory` クラスは、考えられるコンピューター・タイプのキャッシュを作成した後、`ofType()` メソッドを使用して該当するインスタンスを提供します。これは、Java で作成する場合の従来のシングルトン・ファクトリーです。

その一方、Singleton も同じくデザイン・パターンの 1 つであり (「参考文献」を参照)、ランタイムによって組み込まれるパターンの好例の 1 つです。今度はリスト 11 に記載する、Groovy が提供する `@Singleton` アノテーションを使用して単純化した `ComputerFactory` について検討してみましょう。

リスト 11. 単純化されたシングルトン・ファクトリー

```
@Singleton class ComputerFactory {
  def types = [:]

  private ComputerFactory() {
    def laptop = new Laptop()
    def tower = new Desktop()
    types.put("MacBookPro6_2", laptop)
    types.put("SunTower",  tower)
  }

  def ofType(computer) {
    types[computer]
  }
}
```

このファクトリーが正準インスタンスを返すことをテストするために、リスト 12 に記載するユニット・テストを作成しました。

リスト 12. 正準タイプのテスト

```
@Test
public void flyweight_computers() {
  def bob = new AssignedComputer(ComputerFactory.instance.ofType("MacBookPro6_2"), "Bob")
  def steve = new AssignedComputer(ComputerFactory.instance.ofType("MacBookPro6_2"),
  "Steve") assertTrue(bob.computerType == steve.computerType)
}
```

インスタンス間で共通する情報を保存するのは賢明な考えです。関数型プログラミングでも、この考えを適用したいと思いますが、実装の詳細はかなり異なります。これは、パターンのセマンティクスは変わらない一方、実装は変わる (望ましいことに、単純化される) という例です。

前回の記事で、メモ化について説明しました。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、メモ化された関数を使用すれば、ランタイムでは自動的に値がキャッシュされるようにすることができます。メモ化は、Groovy の最新のバージョンでサポートされるようになっています (「参考文献」を参照)。リスト 13 に定義した関数を見てください。

リスト 13. Flyweight パターンのメモ化

```
def computerOf = {type ->
  def of = [MacBookPro6_2: new Laptop(), SunTower: new Desktop()]
  return of[type]
}

def computerOfType = computerOf.memoize()
```

リスト 13 では、`computerOf` 関数内に正準タイプが定義されています。この関数のメモ化されたインスタンスを作成するには、Groovy ランタイムで定義されている `memoize()` メソッドを呼び出します。

リスト 14 に、2 つの手法の呼び出しを比較するユニット・テストを記載します。

リスト 14. 手法の比較

```
@Test
public void flyweight_computers() {
  def bob = new AssignedComputer(ComputerFactory.instance.ofType("MacBookPro6_2"), "Bob")
  def steve = new AssignedComputer(ComputerFactory.instance.ofType("MacBookPro6_2"),
  "Steve") assertTrue bob.computerType == steve.computerType

  def sally = new AssignedComputer(computerOfType("MacBookPro6_2"), "Sally")
  def betty = new AssignedComputer(computerOfType("MacBookPro6_2"), "Betty")
  assertTrue sally.computerType == betty.computerType
}
```

最終的な結果は同じですが、実装の詳細には大きな違いがあることに注目してください。「従来の」デザイン・パターンでは、ファクトリーとして機能する新しいクラスを作成し、2 つのパ

ターンを実装しました。一方、関数型バージョンでは、単一のメソッドを実装してから、メモ化されたバージョンを返しました。キャッシングなどの詳細をランタイムに任せるということは、ハンド・コーディングによる実装で失敗する可能性が少なくなることを意味します。この例の場合、Flyweight パターンのセマンティクスは維持されながらも、実装は大幅に単純化されています。

まとめ

今回の記事では、関数型プログラミングでデザイン・パターンのセマンティクスが現れる 3 つの形を紹介しました。まず、パターンは言語またはランタイムに組み込まれる場合があります。この形の例については、Factory、Strategy、Singleton、および Template Method といったデザイン・パターンを使用して説明しました。2 番目の形では、パターンがそのセマンティクスを維持する一方、完全に異なる実装となります。その例を説明するために使用したのは、クラスを使用した場合とメモ化を使用した場合の Flyweight パターンです。3 番目の形では、関数型言語とランタイムが完全に異なる機能を使用し、まったく異なる方法で問題を解決できるようにします。

次回の記事では、引き続きデザイン・パターンと関数型プログラミングの共通部分を調査し、3 番目の形の例を紹介します。

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。