関数型の考え方: Groovy に隠された関数型の機能、第 1 回

Groovy に潜んでいる宝

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2012年 1月 06日

時代とともに、プログラミング言語やランタイムが私たちに代わって処理してくれる雑事は次第に多くなってきています。こうした傾向を示す典型的な例が関数型言語ですが、最近の動的言語にも、開発者の作業を楽にする関数型の機能が数多く取り込まれるようになってきました。今回の記事では、Groovy に隠されている関数型の機能について調べ、再帰によって状態が隠される仕組みと、遅延リストを作成する方法について説明します。

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。

告白しますが、私はガーベッジ・コレクションが行われないプログラミング言語で再び作業したいとは決して思いません。かといって、これまで長年 C++ のような言語で経験を積んできた私としては、今さら最近の言語の利便性に屈したいとも思いません。最近の言語が便利になったことは、ソフトウェア開発の進化の歴史を物語っています。ソフトウェア開発では、ごくありふれた細かな処理を行う (そしてその処理を隠蔽する) ための抽象化層を積み重ねていきますが、コンピューターの能力が向上するにつれ、より多くのタスクを言語とランタイムに任せるようになってきました。ほんの 10 年ほど前には、本番アプリケーションには遅すぎるという理由で、開発者はインタープリター言語を使うのを避けていましたが、今では一般的に使われるようになりました。関数型言語の多くの機能にしても、10 年前は処理にあまりにも時間がかかりすぎていましたが、今や開発者の時間と作業を最大限に効率化するまでになったため、これらの機能を使用するという選択は至って妥当なものになっています。

この連載で取り上げている機能の多くは、関数型の言語やフレームワークが単純で詳細な処理をどのように扱うのかを明らかにしてくれますが、関数型言語を使用しなくても、関数型の構成体

から恩恵を得ることはできます。今回の記事と次回の記事では、すでに Groovy に組み入れられている関数型プログラミングがどのような形をしているのかを明らかにします。

Groovy の関数型風リスト

Groovy は、Java コレクション・ライブラリーを大幅に増強したものを提供しており、関数型の構成体も追加されています。これによってもたらされる恩恵の 1 つは、リストに対して異なった見方ができることです。この見方は、最初はたいしたことがないように思えるものの、実はいくつかの興味深いメリットをもたらしてくれます。

別の見方でリストを捉える

主に C 言語のような言語 (Java を含む) を扱ってきた開発者は、リストという概念をインデックス付きのコレクションとして捉えるはずです。こうした見方をすれば、インデックスを明示的に指定しなくても、リスト 1 の Groovy コードに示すように、コレクションを容易に繰り返し処理することができます。

リスト 1. (隠れた) インデックスを使用したリストのトラバース

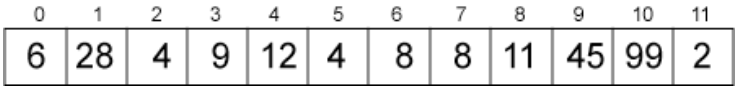
```
def perfectNumbers = [6, 28, 496, 8128]

def iterateList(listOfNums) {
  listOfNums.each { n ->
    println "${n}"
  }
}

iterateList(perfectNumbers)
```

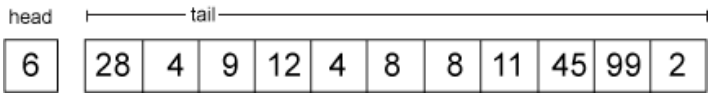
明示的なインデックスへのアクセスが必要な場合に備え、Groovy にはインデックスをコード・ブロックへのパラメーターとして指定する `eachWithIndex()` イテレーターがあります。リスト 1 では `iteraterList()` メソッド内でインデックスを使用していませんが、それでもリストをスロットの順序付きコレクションと見なしていることに変わりはありません (図 1 を参照)。

図 1. インデックス付きスロットとしてのリスト



多くの関数型言語では、リストに対して多少異なった見方をしていますが、幸い、Groovy はその見方を共有しています。それは、リストをインデックス付きスロットと見なすのではなく、リストを先頭要素 (head) とそれ以外の後続要素 (tail) の組み合わせとして捉えるという見方です (図 2 を参照)。

図 2. 先頭要素 (head) と後続要素としてのリスト



先頭要素 (head) と後続要素 (tail) の組み合わせとしてリストを捉えれば、再帰によってリストを繰り返し処理することができます (リスト 2 を参照)。

リスト 2. 再帰を使用したリストのトラバース

```
def recurseList(listOfNums) {
  if (listOfNums.size == 0) return;
    println "${listOfNums.head()}"
    recurseList(listOfNums.tail())
}

recurseList(perfectNumbers)
```

リスト 2 の `recurseList()` メソッドでは、まず、パラメーターとして渡されたリストに要素が含まれているかどうかを確認します。要素が含まれていなければ、そこで処理を完了してリターンすることができます。要素が含まれている場合は、Groovy の `head()` メソッドでリスト内の先頭要素を出力した後、リストの残りの要素で再帰的に `recurseList()` メソッドを呼び出します。

再帰には、使用するプラットフォームに起因する技術的な限界があるため (「参考文献」を参照)、万能の解決策にはなりません。しかし、項目数が少ないリストには問題なく再帰を使用することができます。この再帰に伴う限界はそのうち改善もしくは完全に解決されるであろう、と見込んでいる私にとっては、むしろ再帰がコードの構造に与える影響を調べることのほうに関心があります。再帰に伴う欠点を考えると、再帰を使用した場合の利点はすぐには見えてこないかもしれません。その利点を理解するためには、リストのフィルタリング問題を考えてみてください。リスト 3 に、リストと述部 (ブール・テスト) を受け入れて、特定の項目がリストに含まれているかどうかを判別するフィルタリング・メソッドの例を示します。

リスト 3. Groovy での命令型フィルタリング

```
def filter(list, p) {
  def new_list = []
  list.each { i ->
    if (p(i))
      new_list << i
  }
  new_list
}

modBy2 = { n -> n % 2 == 0}

l = filter(1..20, modBy2)
```

リスト 3 のコードは単純で、保持したい要素のためのホルダー変数を作成し、リストを繰り返し処理し、包含述部を使用して各要素をチェックし、フィルタリングした項目のリストを返します。フィルタリング基準を指定するコード·ブロックは、`filter()` を呼び出すときに渡します。

リスト 3 のフィルタリング·メソッドを、今度は再帰を使用して実装します。リスト 4 を見てください。

リスト 4. Groovy での再帰を使用したフィルタリング

```
def filter(list, p) {
  if (list.size() == 0) return list
  if (p(list.head()))
    [] + list.head() + filter(list.tail(), p)
  else
    filter(list.tail(), p)
}

l = filter(1..20, {n-> n % 2 == 0})
```

リスト 4 の `filter()` メソッドでは、まず、渡されたリストのサイズをチェックして、要素が含まれていない場合はリストを返します。要素が含まれている場合は、リストの先頭要素をフィルタリング述部に対して照合します。フィルタリング基準を満たせば、その要素をリストに追加します (常に正しい型を返すように、空の状態となっている初期状態のリストを使用します)。先頭要素がフィルタリング基準を満たさなければ、後続要素を再帰的にフィルタリングしていきます。

リスト 3 とリスト 4 の違いによって浮き彫りにされる重要な疑問は、状態について誰が面倒を見るのかという点です。命令型のバージョンでその役目を担うのは、私に他なりません。私が `new_list` という名前の新規変数を作成し、私が変数の内容を追加し、処理が終わったら私がその変数を返さなければなりません。一方、再帰バージョンでは言語が戻り値を管理し、再帰がメソッドの呼び出しに対してリターンするたびに、戻り値がスタックに積まれます。リスト 4 の`filter()` メソッドを終了する際には、必ず同じ return 呼び出しが実行されることに注意してください。これによって、スタックには中間値が積まれることになります。

ガーベッジ・コレクションほど劇的に開発作業を改善するものではありませんが、上記の例は、プログラミング言語での重要な傾向を示しています。その傾向とは、可変の構成要素からの解放です。開発者がリストの中間結果に決して手をつけられないとしたら、リストを操作する過程でバグが紛れ込んでしまうこともありません。

リストに対してこのように見方を変えると、リストのサイズや範囲などといった他の側面についても新しい取り組みが可能になります。

Groovy の遅延リスト

あらゆる関数型言語に共通する機能の 1 つは、遅延リストです。遅延リストでは、必要なときにだけリストの中身が生成されるため、処理コストが高いリソースの初期化を、そのリソースがどうしても必要になるまで遅らせることができます。また、遅延リストによって無限数列を作成することもできます。無限数列とは、上限のないリストのことです。前もってリストの大きさを指定する必要がない場合は、必要なだけリストを大きくさせていくことができます。

まずは、リスト 5 に Groovy で遅延リストを使用する例を記載してから、その実装を説明します

リスト 5. Groovy で遅延リストを使用する

```
def prepend(val, closure) { new LazyList(val, closure) }

def integers(n) { prepend(n, { integers(n + 1) }) }

@Test
public void lazy_list_acts_like_a_list() {
  def naturalNumbers = integers(1)
  assertEquals('1 2 3 4 5 6 7 8 9 10',
      naturalNumbers.getHead(10).join(' '))
  def evenNumbers = naturalNumbers.filter { it % 2 == 0 }
  assertEquals('2 4 6 8 10 12 14 16 18 20',
      evenNumbers.getHead(10).join(' '))
}
```

リスト 5 で最初に記述されている `prepend()` メソッドは、新規 `LazyList` を作成して、そこに値を追加できるようにするメソッドです。次の `integers()` メソッドは、この `prepend()` メソッドを使用して整数のリストを返します。`prepend()` メソッドには 2 つのパラメーターを渡します。1 つは

リストの初期値であり、もう 1 つは次の値を生成するためのコードが含まれるコード・ブロックです。`integers()` メソッドはファクトリー・メソッドのように振る舞い、指定された先頭要素の値と、後続要素の値の計算方法に基づいて整数の遅延リストを作成して返します。

リストから値を取得するためのメソッドは、`getHead()` です。このメソッドは、引数に指定された数だけ、リストの先頭から値を返します。リスト 5 の `naturalNumbers` は、すべての整数の遅延評価による数列です。これらの整数の一部を取得するために、必要な整数の数を指定して、`getHead()` メソッドを呼び出します。アサーションが示しているように、このメソッドによって最初の 10 個の自然数のリストが返されます。次に `filter()` メソッドを使用して偶数値の遅延リストを取得し、さらに `getHead()` メソッドを呼び出して最初の 10 個の偶数値を取り出します。

リスト 6 に、`LazyList` の実装を記載します。

リスト 6. `LazyList` の実装

```
class LazyList {
  private head, tail

  LazyList(head, tail) {
    this.head = head;
    this.tail = tail
  }

  def LazyList getTail() { tail ? tail() : null }

  def List getHead(n) {
    def valuesFromHead = [];
    def current = this
    n.times {
      valuesFromHead << current.head
      current = current.tail
    }
    valuesFromHead
  }

  def LazyList filter(Closure p) {
    if (p(head))
      p.owner.prepend(head, { getTail().filter(p) })
    else
      getTail().filter(p)
  }
}
```

遅延リストに格納する先頭要素 (head) と後続要素 (`tail`) は、コンストラクターの中で指定します。`getTail()` メソッドは、後続要素がヌルではないことを確認した上で、後続要素を取得します。`getHead()` メソッドは、リストの先頭から一度に 1 つずつ既存の要素を抽出し、後続要素に新しい値を生成するよう要求することで、返さなければならない数だけ要素を収集します。つまり、`n.times {}` を呼び出すことで、要求された要素の数だけこの処理を行って値を収集し、メソッドの最後でその値を返すというわけです。

リスト 6 の `filter()` メソッドは、リスト 4 と同じ再帰手法を使用していますが、リスト 6 ではスタンドアロン関数としてではなく、リストの一部として実装されています。

遅延リストは Java にもありますが (「参考文献」を参照)、関数型の機能を持つ言語で実装するほうが遥かに簡単です。遅延リストは、リソースを生成するのにかなりの処理コストがかかる場合に威力を発揮します。例えば、完全数のリストを取得する場合などです。

完全数の遅延リスト

この連載を読み続けているとしたら、私のお気に入りのサンプル・コードである完全数検出プログラムについてはすっかりお馴染みのことでしょう (「関数型の考え方: 第 1 回」を参照してください)。このコードの今までのすべての実装に共通する欠点は、分類対象の数値を指定しなければならなかったことです。そこで、分類対象の数値を指定せずに、完全数の遅延リストを返すバージョンを作成したいものです。その目的のために、遅延リストをサポートし、関数型の要素を大幅に採り入れた極めてコンパクトな完全数検出プログラムを作成しました。リスト 7 にそのための NumberClassifier を示します。

リスト 7. `nextPerfectNumberFrom()` メソッドを組み込んで簡潔にした数値分類子のバージョン

```
class NumberClassifier {
  static def factorsOf(number) {
      (1..number).findAll { i -> number % i == 0 }
  }

  static def isPerfect(number) {
      factorsOf(number).inject(0, {i, j -> i + j}) == 2 * number
  }

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

`factorsOf()` および `isPerfect()` メソッドに含まれるコードの意味がよくわからない場合は、前回の記事でこれらのメソッドを取り上げている箇所を参照してください。新しく追加した `nextPerfectNumber()` メソッドは `isPerfect()` メソッドを使用して、パラメーターとして渡された数値の後で最初に登場する完全数を検出します。このメソッド呼び出しは、少数の値に対して実行する場合でも時間がかかります (特にこのコードはほとんど最適化されていないため、なおさら時間がかかります)。しかしそれは単に、完全数がそれほど多くないというだけの話です。

この新しいバージョンの `NumberClassifier` を使用すれば、完全数の遅延リストを作成することができます (リスト 8 を参照)。

リスト 8. 遅延して初期化される完全数のリスト

```
def perfectNumbers(n) {
  prepend(n,
    { perfectNumbers(nextPerfectNumberFrom(n)) }) };

@Test
public void infinite_perfect_number_sequence() {
  def perfectNumbers = perfectNumbers(nextPerfectNumberFrom(1))
  assertEquals([6, 28, 496], perfectNumbers.getHead(3))
}
```

上記では、リスト 5 で定義した `prepend()` メソッドを使用して、初期値を最初のパラメーターとして指定し、次の完全数の計算方法が実装されたクロージャー・ブロックを 2 つ目のパラメーターとして指定することで、完全数のリストを作成します。1 より後で最初に登場する完全数を使用してリストを初期化してから (`NumberClassifier.nextPerfectNumberFrom()` メソッドをより簡単に呼び出せるように、静的インポートを使用しています)、リストから最初の 3 つの完全数を返します。

新しい完全数を計算するには処理コストがかかるので、この計算を行うのは極力避けたいところです。上記のように、この計算の部分を遅延リストとして作成すれば、最適なタイミングが来るまで計算を遅らせることができます。

「リスト」を「番号付きスロット」として抽象化すると、無限数列については考えにくくなります。リストを「先頭要素」と「その残りの要素」として捉えれば、要素を構造ではなくリストに含まれるものとして考えられるようになり、遅延リストなどの機能を思い付けるようになります。

まとめ

開発者が生産性を飛躍的に高める方法の 1 つは、詳細を隠すための効果的な抽象化を作り上げることです。もし私たちが 1 と 0 を使ったコーディングを続けていたとしたら、何の進歩もなかったでしょう。関数型言語の魅力的な 1 つの側面は、開発者の手から離れるように、できるだけ多くの詳細を抽象化しようとするところです。JVM 関連の最近の動的言語では、すでに細かな処理の多くが抽象化されています。今回の記事では、リストがどのように構成されているかについてちょっと見方を変えるだけで、繰り返し処理での状態管理を言語に任せられるようになることを説明しました。リストを「先頭要素 (head)」と「後続要素 (tail)」として捉えることで、遅延リストや無限数列などを作成できるようになるのです。

次回の記事では、Groovy のメタプログラミングによって、プログラムをより関数型に近づける方法を説明します。Groovy のメタプログラミングを使えば、サード・パーティーの関数型ライブラリーを強化して、Groovy でのその有用性をさらに高めることができます。

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。