多忙な Java 開発者のための Scala ガイド: 電卓を作る、第 1 回

Scala の case クラスとパターン・マッチング

Ted Neward
Principal
Neward & Associates

2008年 8月 26日

DSL (Domain-specific language: ドメイン特化言語) がホットな話題となっています。関数型言語が大いにもてはやされる理由は、関数型言語を応用することで DSL のような言語を作成できる点にあります。「多忙な Java™ 開発者のための Scala ガイド」シリーズの第 8 回である今回は、Ted Neward が、「外部」DSL を作成する上での関数型言語の強力さを示す、電卓用の簡単な DSL の作成を開始します。彼はこの目的のために、Scala の新しい機能である case クラスを説明し、また昔からの関数型言語の機能であるパターン・マッチングを再検証します。

このシリーズの他の記事を見る

前回の記事は読者からのフィードバックを元に書いたものですが、その記事が公開された後、これまでこのシリーズで取り上げてきた例が少し簡単すぎるという不満やコメントが私に送られてきました。新しい言語を学ぶ際には、その初期段階で簡単な例を使うことは確かに妥当なことですが、その言語の適用範囲の奥の深さと強力さ、そしてそれによるメリットを示した、もっと「現実的な」内容を読者が要求することもまた非常に妥当なことです。そこで今回の記事と次回の記事の 2 回に分けて、DSL を作成する演習を行います。ここでは DSL として電卓用の簡単な言語を使います。

このシリーズについて

このシリーズでは、Ted Neward が皆さんと共に Scala プログラミング言語を深く掘り下げます。このとても楽しみな developerWorks のシリーズでは、Scala が最近もてはやされている理由を調べ、Scala の言語機能の実際の動作を調べます。Scala のコードと Java のコードの比較が重要な場合には両者のコードを並べて示しますが、(これから学ぶように) Scala の機能のうちの多くは、Java には直接対応するものがありません。そして Scala の魅力の多くがあるのはそこなのです。結局のところ、Java で可能ならば、手間をかけて Scala を学ぶ必要はないのです。

DSL (domain-specific language)

皆さんを監督する厳しいプロジェクト・マネージャーの下から抜け出すことができない (あるいは抜け出す時間がない) 方のために、簡単に要点を説明しましょう。DSL は、アプリケーションの持

つ力を、アプリケーションが本来属するはずの場所、つまりそのアプリケーションのユーザーの手に (再び) 取り戻そうとする試みにすぎません。

ユーザーが直接理解して使用できる新しいテキスト言語を定義することによって、プログラマーは、UI に関する要求とそれに対応する機能強化という終わりのないサイクルから実質的に解放されます。そしてスクリプトや他のツールの作成をユーザーに任せることができ、ユーザーはそれらを使うことで、自分達の作成するアプリケーションに新たな動作を追加できるようになります。おそらく多くの人が興味を惹かれる (そしてごく少数の人が嫌悪感を示すような) 非常に成功した DSL の代表的な例としては、スプレッドシートのセルのさまざまな計算や内容を表現するために使用される Microsoft® Office Excel の「言語」が挙げられるでしょう。さらには、SQL そのものが DSL だと主張する人もいるかもしれません。SQL の場合には、言語はリレーショナル・データベースとのやりとりに特化されています。(プログラマーが、従来の API での `read()`/`write()` 呼び出しを使って Oracle データベースからデータを取得しなければならないことを想像してみてください・・・考えるだけもゾッとします。)

この記事で作成する DSL は、数式を取得し、その式を評価するように設計された、電卓用の簡単な言語です。ここでの目標を突き詰めて言うと、ユーザーが比較的簡単な代数式を入力したときに、その式を評価して結果を生成するような、単純な言語を作成することです。簡単のため、この言語では、本格的な電卓であればサポートする必要のある多くの機能については省略します。ただしこの言語を教育用に限定したくはないため、十分な拡張性を持たせることにします。読者の皆さんは、この言語をもっと強力な言語の中核として使用することができ、そうすればゼロから言語を作らなくてもすむはずです。これはつまり、この言語は可能な限り拡張が容易でカプセル化可能な形を維持する必要があり、使用する上で大きな障害がないものでなければならない、ということです。

DSL に関する補足事項

DSL は大きなテーマであり、この記事の 1 つの段落ではとても紹介しきれないほどの豊富な内容があり、また拡張性に富んでいます。DSL に関して詳しく知りたい方は、この記事の最後に挙げた、Martin Fowler による「執筆途中の本」をぜひ読んでください。特に、「内部」DSL と「外部」 DSL の間の違いについての説明に注目してください。Scala は柔軟な構文と関数型の性質を持っているため、内部 DSL を作成する場合にも外部 DSL を作成する場合にも、Scala を強力な言語として活用することができます。

つまり目標は、(最終的に) クライアントが次のようなコードを作成できるようにすることです。

リスト 1. 電卓用 DSL の目標

```
// This is Java using the Calculator
String s = "((5 * 10) + 7)";
double result = com.tedneward.calcdsl.Calculator.evaluate(s);
System.out.println("We got " + result); // Should be 57
```

この目標まで 1 回の記事で到達することはできませんが、今回はそこに至るまでの一部を説明し、そして次回に全体を完成させることにします。

実装と設計の観点から考えると、最初にストリング・ベースのパーサーを作成し、そして「各文字を取得しながら評価していく」パーサーの流れに従って何かを作成し始める、という方式を採りたくなります。しかしこの方式は単純な言語の場合には有効かもしれませんが、あまり拡張性

がありません。この言語の目標は容易に拡張できるという点にあるため、実装に入る前に少し時間を取り、この言語の設計について考える必要があります。

基本的なコンパイラー理論をよく理解している人であれば、言語プロセッサー (インタープリターとコンパイラーの両方を含みます) の基本的な動作が少なくとも次の 2 つの基本フェーズで構成されることを知っているはずです。

- パーサーが入力テキストを取得し、それを AST (Abstract Syntax Tree: 抽象構文木) に変換します。
- (コンパイラーの場合には) コード・ジェネレーターが AST を取得し、必要なバイトコードを AST から生成します。または (インタープリターの場合には) エバリュエーターが AST を取得し、AST の中で発見したものを実行します 。

一旦 AST に変換することで、結果として生成される構文木に何らかの最適化を行えることを理解できると、上のようなフェーズで構成される理由がより明らかになります。つまりこの電卓の場合であれば、さまざまな式を調べ、大量に式を削除できる場所 (例えば乗算式でオペランドの 1 つが「0」であるような場所など) を探す必要があります。

最初の作業は、この AST を電卓の言語用に定義することです。幸いなことに、Scala には case クラスがあります。case クラスは豊富なデータを扱うことができ、薄くカプセル化されたクラスであり、AST を作成するのに非常に適した便利な機能をいくつか持っています。

case クラス

AST の定義の詳細に入る前に、case クラスとは何かを簡単に説明しましょう。Scala プログラマーにとって case クラスは、想定される何らかのデフォルトの設定を持たせてクラスを作成するための便利な仕組みです。例えば次のようなコードを作成する場合を考えてみてください。

リスト 2. person に case を適用する

```
case class Person(first:String, last:String, age:Int)
{

}
```

こうした場合、Scala のコンパイラーは、期待されるコンストラクターを生成する以上のことをします。つまり Scala のコンパイラーは、一般的な、`equals()` や `toString()`、`hashCode()` の実装も生成するのです。実際、この種の case クラス (つまり追加のメンバーを何も持たないクラス) はあまりにも一般的なため、case クラス宣言の後に中括弧を付けなくてもよいことになっています。

リスト 3. 世界で最も短いクラスのリスト

```
case class Person(first:String, last:String, age:Int)
```

これを検証するためには、昔から使い慣れている `javap` を使えば簡単です。

リスト 4. 神聖なるコード・ジェネレーター

```
C:\Projects\Exploration\Scala>javap Person
Compiled from "case.scala"
public class Person extends java.lang.Object implements scala.ScalaObject,scala.
Product,java.io.Serializable{
    public Person(java.lang.String, java.lang.String, int);
    public java.lang.Object productElement(int);
    public int productArity();
    public java.lang.String productPrefix();
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public int hashCode();
    public int $tag();
    public int age();
    public java.lang.String last();
    public java.lang.String first();
}
```

これを見るとわかるように、従来のクラスでは通常行われないことが case クラスでは大量に行われています。これは case クラスが、Scala のパターン・マッチングと組み合わせて使うように設計されているためです (パターン・マッチングに関しては数回前の記事「コレクション型」の中で簡単に触れました)。

case クラスの使い方は従来のクラスの使い方と少し異なりますが、これは通常は case クラスが従来の「new」構文を使って作成されるわけではないためです。実際、case クラスは通常、そのクラスと同じ名前を持つファクトリー・メソッドを使って作成されます。

リスト 5. 何と new がありません

```
object App
{
  def main(args : Array[String]) : Unit =
  {
    val ted = Person("Ted", "Neward", 37)
  }
}
```

case クラスそのものは、従来のクラス以上に興味深いものにも、従来のクラスと異なるものにも見えないかもしれません。しかし case クラスを使ってみると、重要な違いが明らかになります。case クラス用に生成されるコードは、参照先が等しいかどうかではなく、ビット単位で等しいかどうかに注目するのです。そのため次のコードの結果を見ると、Java プログラマーは少しばかり驚くかもしれません。

リスト 6. これは昔ながらのクラスではありません

```
object App
{
  def main(args : Array[String]) : Unit =
  {
    val ted = Person("Ted", "Neward", 37)
    val ted2 = Person("Ted", "Neward", 37)
    val amanda = Person("Amanda", "Laucher", 27)

    System.out.println("ted == amanda: " +
      (if (ted == amanda) "Yes" else "No"))
    System.out.println("ted == ted: " +
      (if (ted == ted) "Yes" else "No"))
    System.out.println("ted == ted2: " +
      (if (ted == ted2) "Yes" else "No"))
```

```
  }
}

/*
C:\Projects\Exploration\Scala>scala App
ted == amanda: No
ted == ted: Yes
ted == ted2: Yes
 */
```

case クラスの真の価値が発揮されるのは、パターン・マッチングを行う場合です。このシリーズの読者であれば (このシリーズの記事「コレクション型」から)、パターン・マッチングは Java の「switch/case」とは似ているものの、switch/case よりもずっと深い機能と能力を持つものであるということを思い出すはずです。パターン・マッチングで値を突き合わせる場合、単に突き合わせ対象の構成体の値を検証できるだけではありません。その値を一部がワイルドカードの値 (一部が「デフォルト」のようなものと考えてください) と突き合わせることや、突き合わせによる検証に対するガードを case に含めることや、マッチングの基準となる値をローカル変数にバインドすることや、さらにはマッチングの基準となる型そのものに対して突き合わせを行うこともできます。

case クラスによって、パターン・マッチングはまったく新たな処理の能力を持つようになります (リスト 7)。

リスト 7. これも昔ながらの switch ではありません

```
case class Person(first:String, last:String, age:Int);

object App
{
  def main(args : Array[String]) : Unit =
  {
    val ted = Person("Ted", "Neward", 37)
    val amanda = Person("Amanda", "Laucher", 27)

    System.out.println(process(ted))
    System.out.println(process(amanda))
  }
  def process(p : Person) =
  {
    "Processing " + p + " reveals that" +
    (p match
    {
      case Person(_, _, a) if a > 30 =>
        " they're certainly old."
      case Person(_, "Neward", _) =>
        " they come from good genes...."
      case Person(first, last, ageInYears) if ageInYears > 17 =>
        first + " " + last + " is " + ageInYears + " years old."
      case _ =>
        " I have no idea what to do with this person"
    })
  }
}

/*
C:\Projects\Exploration\Scala>scala App
Processing Person(Ted,Neward,37) reveals that they're certainly old.
Processing Person(Amanda,Laucher,27) reveals that Amanda Laucher is 27 years old
.
 */
```

リスト 7 では、非常にさまざまなことがすべて同時に行われています。それらをゆっくりと順次説明し、その後で電卓の例に戻り、リスト 7 の中で行われていることを電卓の例に応用する方法について説明することにします。

まず、`match` 式全体が括弧に囲まれています。これは必ずしもパターン・マッチングの構文ではありませんが、パターン・マッチングの式の前にある接頭文字列にパターン・マッチングの式の結果 (関数型言語ではすべてが式であることを思い出してください) を連結しているため、パターン・マッチングの構文になっています。

そして、最初の `case` 式の中には 2 つのワイルドカードがあります (アンダーバーという文字はワイルドカードです)。つまりこの突き合わせでは、突き合わせ対象の `Person` の中にある 2 つのフィールドに対して任意の値が使われますが、ローカル変数 `a` が導入され、この変数に `p.age` の値がバインドされるということです。この `case` が真になるのは付随するガード式 (`case` の後にある `if` 式) が真になる場合のみです。つまり最初の `Person` では真になりますが、2 番目の `Person` では真になりません。2 番目の `case` 式は `Person` の `firstName` 部分にワイルドカードを使っていますが、`lastName` 部分では定数文字列 `Neward` と突き合わせ、また age 部分に対してはワイルドカードを突き合わせています。

最初の `Person` は既に最初の `case` でマッチしており、また 2 番目の `Person` には `Neward` という苗字 (`last name`) がないので、この 2 番目の `case` はどちらの `Person` に対してもマッチしません。(ただし`Person("Michael", "Neward", 15)` であればマッチします。この場合は最初の case に対するガード節で真にならないため、2 番目の case に進むからです。)

3 番目の case はパターン・マッチングの一般的な使い方を示しています (この使い方は抽出と呼ばれることがあります)。この使い方では、突き合わせ対象のオブジェクト `p` の値が抽出されてローカル変数 (first、last、`ageInYears`) に入れられ、case ブロックの内部で使われます。最後の case 式は、一般的な、case のデフォルトであり、他の case 式がどれも真でなかった場合にのみ実行されます。

以上の説明は case クラスとパターン・マッチングの説明としては簡単すぎるかもしれませんが、この説明を理解した上で電卓用の AST を作成する作業に戻りましょう。

電卓用の AST

まず、おそらく電卓用の ASTには一般的なベース・タイプが必要です。なぜならサブ式から数式が構成されることがよくあるからです。これは「5 + (2 * 10)」という例を見るとよく理解できます。この式では「(2 * 10)」というサブ式は「+」演算の右側のオペランドです。

実際、この式から、AST のタイプとして次の 3 つを得ることができます。

- ベースとなる式 (Expression)
- 定数値を保持する Number 型
- 1 つの演算と 2 つのオペランドを保持する BinaryOperator

少し考えてみて、数学では値を正から負に反転するための否定演算子 (マイナス) などの単項演算子も許されることも思い出してください。そこで基本的な AST として、次のようなものを導入します。

リスト 8. 電卓用の AST (src/calc.scala)

```
package com.tedneward.calcdsl
{
  private[calcdsl] abstract class Expr
  private[calcdsl]  case class Number(value : Double) extends Expr
  private[calcdsl]  case class UnaryOp(operator : String, arg : Expr) extends Expr
  private[calcdsl]  case class BinaryOp(operator : String, left : Expr, right : Expr)
   extends Expr
}
```

このすべてをパッケージ (`com.tedneward.calcdsl`) の中に入れるためのパッケージ宣言と、各クラスの前にあるアクセス修飾子宣言に注目してください (アクセス修飾子宣言は、このパッケージまたはサブ・パッケージの他のメンバーからアクセスできることを示します)。このようにする理由は、一連の JUnit テストによってこのコードを実行したいからです (電卓の実際のクライアントは AST を見る必要はありません)。そこで、`com.tedneward.calcdsl` のサブ・パッケージとなるユニット・テストを作成します。

リスト 9. 電卓用のテスト (testsrc/calctest.scala)

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    import org.junit._, Assert._

    @Test def ASTTest =
    {
      val n1 = Number(5)

      assertEquals(5, n1.value)
    }

    @Test def equalityTest =
    {
      val binop = BinaryOp("+", Number(5), Number(10))

      assertEquals(Number(5), binop.left)
      assertEquals(Number(10), binop.right)
      assertEquals("+", binop.operator)
    }
  }
}
```

ここまでは問題ありません。これで AST が用意できました。

少し考えてみてください。4 行の Scala コードによって、任意の深さを持つ数式のコレクションを表現する型の階層構造を作成することができました (確かに簡単な数式ですが、それでも十分に実用に耐えます)。オブジェクト・プログラミングを容易に、より表現力豊にするという Scala の方向性から見る限り、これはあまり関数型とは言えません。(関数型の側面は後ほど出てきますので心配する必要はありません。)

次に、AST を取得して数値にする評価関数が必要です。これはパターン・マッチングの力を利用すればとても簡単に作成することができます。

リスト 10. calculator (src/calc.scala)

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    def evaluate(e : Expr) : Double =
    {
      e match {
        case Number(x) => x
        case UnaryOp("-", x) => -(evaluate(x))
        case BinaryOp("+", x1, x2) => (evaluate(x1) + evaluate(x2))
        case BinaryOp("-", x1, x2) => (evaluate(x1) - evaluate(x2))
        case BinaryOp("*", x1, x2) => (evaluate(x1) * evaluate(x2))
        case BinaryOp("/", x1, x2) => (evaluate(x1) / evaluate(x2))
      }
    }
  }
}
```

`evaluate()` によって `Double` が返されることに注目してください。これはつまり、パターン・マッチングの中の各 case を評価した結果は `Double` 値でなければならないということです。これは難しくありません。Number の場合は単に、その Number に含まれる値を返せばよいのです。しかしそれ以外の case (2 種類の演算子 (UnaryOp と BinaryOp)) の場合には、オペランドに対して必要な演算 (否定、加算、減算等) を行う前に、オペランドも評価する必要があります。関数型言語では一般的なことですが、ここで再帰が登場し、全体としての演算を行う前に単に各オペランドに対して `evaluate()` を呼び出せばよいのです。

さまざまな演算子の外で評価を実行するというこの発想は、生粋のオブジェクト指向プログラマーの大部分にしてみれば、根本的に誤っているように思えるでしょう。これは明らかにカプセル化とポリモーフィズムの原則に違反しています。正直なところ、議論するまでもありません。少なくとも従来の感覚では、間違いなくカプセル化に違反しています。

しかしここでは、一体このコードを何から分離してカプセル化しようとしているのか、という、より大きな問題を考える必要があります。AST のクラスはこのパッケージの外からはまったく見えず、(最終的には) クライアントは評価して欲しい式のストリング表現を渡すのみ、ということを思い出してください。AST の case クラスを直接扱うのはユニット・テストのみなのです。

ただしこれは、すべてのカプセル化は死んだもの、あるいは過去のものという意味ではありません。実際はまったく逆であり、これはオブジェクトの世界でおなじみの手法をはるかに超える、他の設計手法があることを示すための試みなのです。Scala がオブジェクト型と関数型を融合したものであることを忘れてはなりません。`Expr` とそのサブクラスにさらなる振る舞いを追加しなければならない状況があったとすると (例えば `toString` メソッドを美しく出力するなど)、ほとんど手間をかけずにそれらのメソッドを `Expr` に追加することができます。関数型の世界とオブジェクト指向の世界を組み合わせることによって新たな世界が生まれるのであり、関数型言語のプログラマーもオブジェクト指向のプログラマーも、相手側の設計手法や、2 つを組み合わせることで興味深い効果が得られるという事実を無視すべきではありません。

設計の観点から見ると、他の選択肢には疑問が残ります。例えばストリングを使って演算子を保持する方法では、わずかなタイプミスによって誤った結果が生ずるおそれがあります。実

稼働用コードでは、これをストリングではなく列挙にすることができます (そしておそらく列挙にする必要があります)。しかしストリングのままにしておけば、もっと複雑な関数 (例えば abs、sin、cos、tan など) や、さらに極端な場合にはユーザー定義の関数を呼び出せる演算子を作成できる可能性が広がるかもしれません (こうしたことは列挙ベースの手法では実現が困難です)。

設計や実装に関する判断すべてに言えることですが、正しい方法が 1 つだけあるわけではなく、単にそれぞれの方法には相応の結果が伴うということです。責任は使う側にあるのです。

ただしここでは、実装のための興味深い手法が 1 つあります。ある種の数式は単純化できるため、(潜在的には) 式の評価を次のように最適化することができます (そして偶然ではなく、AST の有用性を示すことができます)。

- 「0」を追加するものはすべて、ゼロではないオペランドにまで単純化することができます。
- 「1」を掛けるものはすべて、ゼロではないオペランドにまで単純化することができます。
- 「0」を掛けるものはすべて、ゼロにまで単純化することができます。

そして他も同様です。そこで、こうした単純化そのものを行う、`simplify()` という事前評価ステップを導入します。

リスト 11. calculator (src/calc.scala)

```
  def simplify(e : Expr) : Expr =
  {
    e match {
      // Double negation returns the original value
      case UnaryOp("-", UnaryOp("-", x)) => x
      // Positive returns the original value
      case UnaryOp("+", x) => x
      // Multiplying x by 1 returns the original value
      case BinaryOp("*", x, Number(1)) => x
      // Multiplying 1 by x returns the original value
      case BinaryOp("*", Number(1), x) => x
      // Multiplying x by 0 returns zero
      case BinaryOp("*", x, Number(0)) => Number(0)
      // Multiplying 0 by x returns zero
      case BinaryOp("*", Number(0), x) => Number(0)
      // Dividing x by 1 returns the original value
      case BinaryOp("/", x, Number(1)) => x
      // Adding x to 0 returns the original value
      case BinaryOp("+", x, Number(0)) => x
      // Adding 0 to x returns the original value
      case BinaryOp("+", Number(0), x) => x
      // Anything else cannot (yet) be simplified
      case _ => e
    }
  }
```

この場合も、これらの式の作成が、パターン・マッチングによる定数との突き合わせ機能と変数のバインディング機能を使うことで、どれほど容易になるかに注目してください。`evaluate()` に対する唯一の変更は、評価の前に単純化のための呼び出しを行っている点です。

リスト 12. calculator (src/calc.scala)

```
    def evaluate(e : Expr) : Double =
    {
      simplify(e) match {
        case Number(x) => x
        case UnaryOp("-", x) => -(evaluate(x))
        case BinaryOp("+", x1, x2) => (evaluate(x1) + evaluate(x2))
        case BinaryOp("-", x1, x2) => (evaluate(x1) - evaluate(x2))
        case BinaryOp("*", x1, x2) => (evaluate(x1) * evaluate(x2))
        case BinaryOp("/", x1, x2) => (evaluate(x1) / evaluate(x2))
      }
    }
```

もっと単純化することもできます。これはツリーの最も下のレベルを単純化しただけであることに気付いたでしょうか。`BinaryOp` が `BinaryOp("*", Number(0), Number(5))` と `Number(5)` を含んでいるとすると、内側にある `BinaryOp` を `Number(0)` に単純化することができますが、そうすると外側にある BinaryOp のオペランドの 1 つがゼロになるので、外側にある `BinaryOp` も `Number(0)` に単純化することができます。

少し面倒になったので、この定義作業は読者の演習とします。いや、これをもっと楽しいものにしましょう。もし読者が自分の実装を私に送ってくれたら、私はその実装を次回の記事のコードと説明の中に (名前を紹介した上で) 含めることにします。(現状では、いくつかのユニット・テストはこの条件をテストしても失敗します。皆さんのミッションは (もしこのミッションを受け入れることを選択した場合には)、これらのテストに (そして`BinaryOp` や `UnaryOp` のネスト深さを任意のレベルにしたテストに) パスすることです。)

まとめ

当然ですが、この記事はまだ終了しておらず、解析の作業が残っています。ただし電卓用の AST はかなり良くできています。大きな変更を行わなくとも演算を追加することができ、(Gang of Four (四人組) の Visitor パターンを使うことで) AST をウォークスルーするために大量のコードは必要なく、また (もしクライアントが評価用に喜んで AST を作成してくれるなら) 計算そのものを行うための実際に動作するコードは用意できています。

もっと重要な点として、この記事では case クラスがパターン・マッチングと非常にうまく協調動作すること、そしてこの組み合わせによって AST の作成と評価が非常に簡単になることを説明しました。case クラスとパターン・マッチングとを組み合わせる設計手法は Scala のコードでは (そして実際、大部分の関数型言語のコードでも) 頻繁に使われ、この環境を本格的に利用するつもりであれば、ぜひこの手法に慣れる必要があります。

著者について

Ted Neward

Ted Neward は、Neward & Associates の代表として、Java や .NET、XML サービスなどのプラットフォームに関するコンサルティング、助言、指導、講演を行っています。彼はワシントン州シアトルの近郊に在住です。