Scala 関数型プログラミング入門

Suguru Hamazaki

1 プログラミング言語 Scala

Scala は、オブジェクト指向と関数型をミックスした、 マルチパラダイム言語です。 Scala には様々な特徴がありますが、今回は、関数型言語としての特徴に焦点をあてます。

2 関数型プログラミング

初めに、関数型プログラミングという言葉について簡単に説明します。 関数型プログラミングの学術的な定義については、専門分野の方々におまかせするとして、 ここでは、「純粋関数のみを使うスタイルのプログラミング」という説明にとどめておきます。
純粋関数 (Pure function) とは、以下の条件が成り立つ関数のことです。

- 関数の評価結果が引数の値のみによって決まり、同じ値を与えると常に同じ値の結果を返す
- 関数の評価によって、観測可能な副作用が発生しない

副作用とは、例えば以下のようなことです。

- 変数に再代入する
- データの構造を破壊的に変更する
- オブジェクトのフィールドに値をセットする
- 例外を投げる、エラー時に終了する
- コンソールに出力する、ユーザー入力を読む
- ファイルを読み書きする
- スクリーンに描画する

Scala はオブジェクト指向プログラミングのための機能も備えており、 関数型言語としては非純粋関数型に分類されます。 そのため、副作用を許容するプログラミングも可能です。
しかし、副作用をできるだけ発生させずにプログラムを書くことにはメリットがあります。そのメリットの例について、 次で説明します。

2.1 参照透過性

ある式の中で、その式の値を変えることなく、等しいもの同士を置換できることを、参照透過性 (Referential Transparency) と言います。もし、プログラムが純粋関数のみで作られていれば、 そのプログラムは参照透過になります。
そして、参照透過性が保たれたプログラムは、置換モデル (Substitution Model) を使うことで、プログラムの動作を正しく理解することができます。

2.2 置換モデルの使用例

Structure and Interpretation of Computer Programs で取り上げられている例を使って、置換モデルをどのように使用するのかを見てみます。

例えば、以下のように sumOfSquares()メソッドと square()メソッドが定義されているとします。

```
def sumOfSquares(i1: Int, i2: Int) = square(i1) + square(i2)
def square(i: Int) = i * i
```

この時、以下のように sumOfSquares(a + 1, a * 2)を評価すると、その結果は 136 になります。

```
scala> val a = 5
a: Int = 5

scala> sumOfSquares(a + 1, a * 2)
res1: Int = 136
```

置換モデルを使って、上の結果を次のように導くことができます。

```
sumOfSquares(a + 1, a * 2)
sumOfSquares(5 + 1, 5 * 2)
sumOfSquares(6, 10)
square(6) + square(10)
(6 * 6) + (10 * 10)
36 + 100
136
```

このように、プログラムの参照透過性が保たれていれば、置換モデルを使って プログラムを解析することができます。

3 コインのモデル

純粋関数を使って、参照透過性を保ちながらプログラムを書くには、 オブジェクト指向プログラミングとは異なるコツが必要になります。
ここでは、以下のコインのモデルを使って、ステートマシンの実装方法について考えます。

- コインは 2 通りの状態を持つ (裏 / 表)
- コインに対して 2 通りの操作を定義する (ひっくり返す / そのままにする)
  - 裏のコインをひっくり返すと表になり、表のコインをひっくり返すと裏になる
  - 裏のコインをそのままにすると裏になり、表のコインをそのままにすると表になる

3.1 オブジェクト指向的な実装

以下は、オブジェクト指向的なアプローチで、コインを実装した例です。

```
case class OoCoin(private var head: Boolean) {
  def flip() = { head = !head }
  def stay() = {} // do nothing
  def get = head
}
```

```
object OoCoin {
  def example() = {
    val c = OoCoin(true)
    c.flip()
    c.stay()
    c.flip()
    println("Showing a head? " + c.get)
  }
}
```

OoCoin クラスの中で、head フィールドを var として宣言しています。 Scala では、var で宣言された変数は再代入できます。実際に、flip()メソッドの中で、head に再代入を行なっています。
flip()と stay()は引数を持たず、これらの結果は head フィールドの値に依存します。すなわち、参照透過ではありません。
参照透過性を持たない OoCoin のこの実装の場合、 最後の c.get が返す値を正しく判断するために、c に対する操作をくまなく追跡して、head フィールドの値を把握している必要があります。

3.2 純粋関数のみを使う

次は、純粋関数のみを使った例です。
まず、フィールドに対する再代入を禁止するために、Coin を次のように定義しなおします。

```
case class Coin(head: Boolean)
```

上の head は val 扱いです。再代入しようとすると、次のようにエラーになります。

```
scala> val c = Coin(true)
c: com.suguruhamazaki.Coin = Coin(true)

scala> c.head
res0: Boolean = true

scala> c.head = false
<console>:9: error: reassignment to val
       c.head = false
              ^
```

そして、flip()と stay()は、Coin クラスとは別に、以下のように実装します。

```
  def flip(c: Coin) = Coin(!c.head)
  def stay(c: Coin) = c
```

上の flip()メソッドでは、受け取った Coin は変更せずに、新しい Coin のインスタンスを適切に作って返しています。
この実装では、Coin のインスタンスは一度作られたら最後、その状態が変わることはありません。flip(),stay()メソッドも、その引数のみによって返り値が決まり、 参照透過性が保たれています。
しかし、この実装が果たして使いやすいでしょうか。以下の使用例を見てみましょう。

```
    val c0 = Coin(true)
    val c1 = flip(c0)
    val c2 = stay(c1)
    val c3 = flip(c2)
    println("Showing a head? " + c3.head)
```

上のプログラムでは、flip()や stay()のような操作と操作の間の繋がりを、 API の使用者側で管理しています。
例えば、1 つ目の flip()で返ってきた値は、 次の操作である stay()の入力として渡さなくてはなりません。同様に、その結果は、3 つ目の操作（2 つ目の flip()）の入力として渡す必要があります。どの結果をどの入力として渡すかを間違えると、 意図した動作になりません。

3.3 コインの操作を表現する

そこで、 コインに対する操作を、単なるメソッドで表現するのをやめます。 代わりに、CoinAction クラスを定義して、コインに対する操作をそれで表わすことにします。CoinAction クラスは、コインの操作を定める action 関数をラップし、 連続したコインの操作を合成する機能（+()メソッド)と、 ある状態のコインに対して操作を行なう機能(apply()メソッド) を提供します。 以下は CoinAction クラスの定義です。

```
  case class CoinAction(action: Coin => Coin) extends (Coin => Coin) {
    def apply(c: Coin) = action(c)
    def +(next: CoinAction): CoinAction = CoinAction { c0 =>
      val c1 = action(c0)
      next(c1)
    }
  }
```

上のコードでは、 Scala の特徴的な機能を幾つか使っているので、以下で詳しく説明します。

3.3.1 CoinAction のコンストラクター

CoinAction のコンストラクターは引数として、Coin => Coin 型の action を受け取ります。Coin => Coin 型は、Coin 型の引数を 1 つ取り Coin 型を値返す「関数」を表わします。Scala では、他の多くの関数型言語と同様に、関数を first class object として扱うことが出来ます。 つまり、関数について以下を行うことが出来ます。

- 関数の引数として渡す
- 関数の戻り値として返す
- 変数に代入する

Scala 言語の内部では、Coin => Coin 型は Function1[Coin, Coin]型と同じものとして扱われます。=>を使った表記の方が直感的に読み易いので、 Scala のコードではしばしばこの表記が使われます。
また、関数は当然、それを呼び出すことも可能です。例えば、CoinAction のコンストラクターが受け取った action を、apply()メソッドの中では action(c)のように呼び出しています。

3.3.2 型の継承と apply() メソッド

CoinAction クラスはそれ自身が Coin => Coin 型、すなわち Function1[Coin, Coin]

型を継承しています。そのため、CoinAction クラスのインスタンスは、関数として扱うことができます。つまり、括弧で呼び出して、 あらかじめ定義された振る舞いを実行することができます。
呼び出し時に実行される振る舞いは、apply()メソッドを実装して定義します。CoinAction クラスの apply()メソッドでは、ラップしている action をそのまま呼び出しています。

3.3.3 +() メソッド

+()メソッドでは、引数で受け取った CoinAction と自身がラップする action を元に、新たな CoinAction を作っています。CoinAction のコンストラクターの引数には Coin => Coin 型の関数を渡さなくてはなりませんので、関数リテラルを作って渡しています。 以下が、Coin => Coin 型の関数リテラルの部分です。

```
{ c0 =>
  val c1 = action(c0)
  next(c1)
}
```

c0 は、型が省略されていますが、Coin 型の引数を表わします。Coin 型の引数を受け取った際に、そのコインに対して action を実行し、 その結果（c1）に対して更に next を実行し、その結果を返します。 ただし、この振る舞いが実行されるのは、+()メソッドが呼び出された時でも、 このコンストラクターが呼ばれた時でもありません。 ここでは単に、そのような振る舞いをする関数を作っているだけです。その関数は action として Coin が保持し、Coin の apply()が呼ばれた時に初めて実行されます。

3.3.4 CoinAction の使い方

CoinAction クラスを使うと、flip と stay は以下のように CoinAction クラスのインスタンスとして定義されます。

```
  val flip = CoinAction(c => Coin(!c.head))
  val stay = CoinAction(c => c)
```

これらは+()メソッドを使って、次のように一つの CoinAction にまとめることが出来ます。

```
    val action = flip + stay + flip
    val c = action(Coin(true))
    println("Showing a head? " + c.head)
```

上では、順に操作を合成して、出来た操作 action に対して、 コインの初期値を与えて結果を取り出しています。操作と操作を合成する際、 コインの状態が適切に引き継がれるようになっているため、先ほどの例のように API の使用者が状態を管理する必要は無くなりました。
しかし、この API にも使いづらい点があります。例えば、上では最後のコインの状態のみ出力していますが、 途中のコインの状態も出力したい時はどうすれば良いでしょうか。この API で途中の状態を出力しようとすると、結果を逐一変数に保存しなくてはならず、 結局、先ほどと同じように状態の受け渡しを自分で行うことになってしまいます。

3.4 途中の結果を利用する

コインの状態を次の操作に渡しつつ、途中の結果を取り出すために、CoinAction クラスの定義を次のように変えます。

```
  case class CoinAction[A](action: Coin => (Coin, A))
    extends (Coin => (Coin, A)) {
    def apply(c: Coin) = action(c)
    def +[B](next: CoinAction[B]): CoinAction[B] =
      flatMap(_ => next)
    def map[B](f: A => B): CoinAction[B] = CoinAction { c0 =>
      val (c1, a) = apply(c0)
      (c1, f(a))
    }
    def flatMap[B](f: A => CoinAction[B]): CoinAction[B] =
      CoinAction { c0 =>
        val (c1, a) = apply(c0)
        f(a)(c1)
      }
  }
```

変更点について、以下で説明します。

3.4.1 ラップする action 関数

まず、ラップする action 関数の型が Coin => Coin から Coin => (Coin, A)に変わりました。つまり、action 関数を呼び出した時の戻り値が、Coin 型から(Coin, A)型になりました。(Coin, A)は Scala におけるタプルの表現で、Coin 型と A 型のペアを表わします。
戻り値を単なる Coin から(Coin, A)に変更した理由は、 ある操作をしてコインの状態が遷移した時に、 遷移先の状態と一緒になんらかの結果を取り出せられるようにするためです。
また、状態が遷移した時の結果というのは、単純に考えれば Boolean 型（Coin の head が Boolean 型なので）になるのですが、Boolean 型から変換する余地を残すために、CoinAction の型パラメーターA として任意の型を指定できるようにしています。
そして、action 関数が Coin => (Coin, A)型になったのに合わせて、CoinAction クラスが継承する型も Coin => (Coin, A)になっています。
これに合わせて書き直した flip と stay は、以下のようになります。

```
  val flip = CoinAction { c =>
    val head = !c.head
    (Coin(head), head)
  }
  val stay = CoinAction(c => (c, c.head))
```

3.4.2 map() メソッド

map()メソッドは、A 型の結果を任意の型に変換するためのメソッドです。状態遷移時に A 型の結果を（Coin 型と共に）返す CoinAction に対し map()メソッドを呼び出すと、 それによって得られる CoinAction は、状態遷移時に B 型の結果を返す CoinAction になります。
具体的にどのように A 型から B 型へ変換するかは、引数 f で渡します。f の型は A => B ですので、A 型の引数を 1 つとり、B 型を返す関数です。

3.4.3 flatMap() メソッド

flatMap()メソッドは、先ほどの+()メソッドの機能拡張版と考えると、理解しやすいです。+()メソッドと同様に、連続する 2 つの CoinAction を組み合わせて、1 つの CoinAction を作ります。 ただし、+()メソッドと異なるのは、次の操作を単なる CoinAction として受け取るのではなく、関数 A => CoinAction[B]として受け取る点です。これにより、今の操作の結果を参照しつつ CoinAction[B] を作り出す余地が生まれます。
ちなみに、+()メソッドの実装が、flatMap()メソッドを使うように書き換えられています。

3.4.4 CoinAction の使い方

新しい CoinAction の flatMap()メソッドを使うと、先ほどの flip + stay + flip を次のように書くことが出来ます。

```
    val action =
      flip.flatMap { _ =>
        stay.flatMap { _ =>
          flip
        }
      }
```

一見すると、先ほどとはかけ離れたコードに見えるかもしれませんが、実はそうではありません。 以下の違いの他は、同一のコードです。

- メソッド名が+から flatMap になった
- 操作（CoinAction 型）を渡していた部分が、操作を生成する関数（Boolean => CoinAction 型）を渡すようになった
- .,{}, 改行の有無など、シンタックス上の違い

そして、+()の代わりに flatMap()を使う利点は、map()と組み合わせて以下のようなコードが書けることです。

```
    val action =
      flip.flatMap { b1 =>
        stay.flatMap { _ =>
          flip.map { b3 =>
            (b1, b3)
          }
        }
      }
    val (_, (b1, b3)) = action(Coin(true))
    println("1st occurrence is a head? " + b1)
    println("3rd occurrence is a head? " + b3)
```

前述の通り、flatMap()メソッドが受け取るのは CoinAction ではなく、A を受け取って CoinAction を返す関数 f です。そして、関数 f が受け取る引数 A というのは、今回の操作の結果です。ですので、関数 f の中で次の結果を生成する際、 引数として受け取った A を利用することができます。
この点を上のコードで具体的に説明すると、3 つ目の操作である flip は、map()関数を使って Boolean 型の結果 b3 を(Boolean, Boolean)型の結果(b1, b3)に変換しています。変数 b1 がここで利用できる理由は、1 つ目の操作である flip の flapMap()メソッドの中にあるからです。つまり、1 つ目の操作の結果を b1 として受け取って、2 つ目以降の操作と合成された CoinAction を生成する関数の一部分であるため、b1 が利用できます。
最終的に、val action の型は CoinAction[(Boolean, Boolean)], すなわち、(Boolean, Boolean)型の結果を生み出す CoinAction となります。この action に Coin

を適用すると、その結果は(Coin, (Boolean, Boolean))型となるので、 destructuring bindings を使って b1,b2 のみ取り出しています。
実は、map()と flatMap()を使った上の書き方には、 for-comprehension と呼ばれる別の書き方が用意されています。 for-comprehension を使って以下のように書き直すと、より簡潔で見通しの良いコードになります。

```
    val action = for {
      b1 <- flip
      _ <- stay
      b3 <- flip
    } yield (b1, b3)
    val (_, (b1, b3)) = action(Coin(true))
    println("1st occurrence is a head? " + b1)
    println("3rd occurrence is a head? " + b3)
```

map()と flatMap(), あるいは for-comprehension の使い方に慣れると、 以下のように書くこともできます。

```
    val action = for {
      s1 <- flip.map(b1 => "1st occurrence is a head? " + b1)
      _ <- stay
      s3 <- flip.map(b3 => "3rd occurrence is a head? " + b3)
    } yield (s1 + "\n" + s3)
    val (_, s) = action(Coin(true))
    println(s)
```

上のコード中で副作用を持つのは、最後の println()の部分のみです。 それ以外は全て純粋関数のみで書かれており、参照透過です。 このように、純粋関数のみを使ってステートマシンを実装することが出来ました。

3.5 CoinAction を抽象化する

ここで、CoinAction クラスの定義をもう一度よく見てみると、クラス定義の中に Coin 特有の処理が全く無いことに気付きます。先ほどの CoinAction は、任意の型 S の状態遷移を表わす State クラスとして抽象化できます。以下は、Functional Programming in Scala で、第 6 章 "Purely functional state" の説明に使われているコードを元に、本ドキュメントの説明に合わせて若干修正したものです。

```
case class State[S, +A](run: S => (S, A)) {
  def map[B](f: A => B): State[S, B] =
    State { s =>
      val (s2, a) = run(s)
      (s2, f(a))
    }
  def flatMap[B](f: A => State[S, B]): State[S, B] =
    State { s =>
      val (s2, a) = run(s)
      f(a).run(s2)
    }
}
```

上の State クラスを使うと、CoinAction と、そのインスタンスである flip,stay は次のよ

うに定義できます。

```
type CoinAction[A] = State[Coin, A]
val flip: CoinAction[Boolean] = State { c =>
  val head = !c.head
  (Coin(head), head)
}
val stay: CoinAction[Boolean] = State(c => (c, c.head))
```

`map()`,`flatMap()`, for-comprehension の使い方は、先ほどの例と同様です。

3.6 Scalaz を利用する

実は、`State` クラスは自分で用意する必要すらありません。Scalaz というライブラリーに含まれる `State` 型を、以下のようにそのまま使うことができます。

```
import scalaz.State
type CoinAction[A] = State[Coin, A]
```

4 まとめ

- 純粋関数のみを使って、簡単なステートマシンを実装する方法を紹介しました。
- その方法を汎用的に利用できる、より抽象的な実装を紹介しました。
- 既存のライブラリーで提供されている、同様の API を紹介しました。