Java.next: カリー化と部分適用

関数の実行方法に強力さと間接的な手段をもたらす

Neal Ford

2014年 1月 23日

カリー化と部分適用は、すべての Java.next 言語に組み込まれている手法ですが、その実装方法は言語によって異なります。今回の記事では、カリー化と部分適用について説明し、この 2 つの手法の違いを明らかにします。そして、Scala、Groovy、Clojure のそれぞれでの実装の詳細 (そして実際の使用例) を紹介します。

このシリーズの他の記事を見る

この連載について

Java の遺産となるのは、プラットフォームであって、言語ではないでしょう。200 を超える言語が JVM 上で実行されている今、最終的にこれらの言語の 1 つが JVM のプログラミングに最適な方法としてJava 言語に取って代わることは避けられません。この連載では、Java 開発者が自分たちの近い将来を垣間見ることができるように、3 つの次世代 JVM 言語 — Groovy、Scala、Clojure — について、新しい機能やパラダイムを比較対照することで、詳しく探ります。

カリー化と部分適用は、(20 世紀の数学者、Haskell Curry らによる研究を基に) 数学から生まれたプログラミング言語の手法です。この 2 つの手法は、さまざまな種類のプログラミング言語に存在します。関数型言語でも、そのいずれか、または両方の手法が偏在しています。カリー化と部分適用は、どちらも通常は特定の引数に対して 1 つ以上のデフォルト値を指定する (引数を「固定する」と表現されます) ことによって、関数またはメソッドの引数の個数を操作できるようにします。すべての Java.next 言語にカリー化と部分適用は組み込まれていますが、その実装方法は言語によって異なります。今回の記事ではこの 2 つの手法の違いを説明し、Scala、Groovy、Clojure での実装の詳細を、実際の使用例と併せて紹介します。

用語についての注記

この記事では、「メソッド」と「関数」という用語は互いに置き換えることができますが、カリー化と部分適用をサポートしているオブジェクト指向言語では、「メソッド」という用語を使用します。同様に、「関数のパラメーター」と「関数の引数」も、この記事では互いに置き換えることができます。カリー化と部分適用の概念は、数学から生まれたものであるため、私は一貫して「関数」と「引数」という用語を使用しますが、だからと言って、カリー化と部分適用の手法がメソッドに対して有効でないことを意味するわけではありません。

IBM SDK for Java 8 オープン・ベータ版プログラム

定義および違い

うわべだけを見ると、カリー化と部分適用には同じ効果があるように思えます。どちらの手法にしても、関数の 1 つのバージョンとして、一部の引数に対してあらかじめ値が指定された関数を作成することができます。

- **「カリー化」**は、複数の引数を取る関数を、1 つの引数のみを取る関数のチェーンに変換する処理を表す用語です。つまり、変換された関数を呼び出すことを表しているのではなく、変換プロセスそのものを表しています。この関数に適用する引数の個数は、呼び出す側が決めることができ、それによって少ない数の引数を取る関数を派生させることができます。
- **「部分適用」**は、複数の引数を取る関数を、それより少ない数の引数を取る関数に変換し、除外された引数には最初から値を指定しておく、という処理を表す用語です。その名前にふさわしく、この手法は関数に対して複数の引数を部分的に適用し、残りの引数で構成されるシグニチャーを持つ別の関数を返します。

カリー化と部分適用では、一部の引数の値を指定すると、値を指定しなかった引数を新たな引数に取る関数が (呼び出し可能な関数として) 返されることは共通しています。その一方で、関数をカリー化した場合には、関数のチェーンで次の関数が返されますが、部分適用では、実行時に引数の値は指定した値に「束縛」され、元の関数よりもアリティーが小さい (引数の個数が少ない) 別の関数が生成されます。この違いは、アリティーが 2 より大きい関数について検討すると、より明確になります。例えば、関数 `process(x, y, z)` を完全にカリー化したバージョンは、`process(x)(y)(z)` です。この場合、`process(x)` と `process(x)(y)` はどちらも 1 つの引数を取る関数です。1 番目の引数のみをカリー化した場合、`process(x)` の戻り値は 1 つの引数を取る関数であり、その関数の戻り値も、1 つの引数を取る関数です。それとは対照的に、部分適用を使用すると、アリティーが小さくなった関数が返されることになります。`process(x, y, z)` に対し、1 つの引数の部分適用を使用した場合は、2 つの引数を取る関数 `process(y, z)` が生成されます。

カリー化と部分適用の結果が同じになることはよくありますが、この 2 つの手法の違いは重要です。しかし多くの場合、その違いについては誤った解釈をされています。さらに事態を複雑化することに、Groovy で実装している部分適用とカリー化は、どちらも `curry` を呼び出します。また Scala には、関数の部分適用と `PartialFunction` という、似たような呼び名でありながらも異なる概念があります。

Scala の場合

Scala では、カリー化と部分適用のほかにも、制約を設けた関数を定義するために使用できるトレイトをサポートしています。

カリー化

Scala の関数では、複数の引数リストを括弧のセットとして定義することができます。関数に定義された引数の個数よりも少ない数の引数を指定して関数を呼び出すと、指定されなかった引数のリストを引数として取る関数が返されます。リスト 1 に記載する、Scala のドキュメントから抜粋した例について検討してみましょう。

リスト 1. Scala での引数のカリー化

```
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 modN(n: Int)(x: Int) = ((x % n) == 0)

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

リスト 1 の `filter()` 関数は、渡されたフィルター基準を再帰的に適用します。`modN()` 関数は、2 つの引数リストを使用して定義されています。ここでは、`filter()` を使って `modN` を呼び出すときに、引数を 1 つだけ渡しています。`filter()` 関数が 2 番目の引数として取るのは、Int 型の引数を 1 つ取って `Boolean` を返す関数です。私が渡しているカリー化された関数のシグニチャーは、この 2 番目の引数として取る関数に該当します。

関数の部分適用

リスト 2 に示すように、Scala では関数の部分適用をすることもできます。

リスト 2. Scala での関数の部分適用

```
def price(product : String) : Double =
  product match {
    case "apples" => 140
    case "oranges" => 223
}

def withTax(cost: Double, state: String) : Double =
  state match {
    case "NY" => cost * 2
    case "FL" => cost * 3
}


val locallyTaxed = withTax(_: Double, "NY")
val costOfApples = locallyTaxed(price("apples"))

assert(Math.round(costOfApples) == 280)
```

リスト 2 では、最初に製品と価格との間のマッピングを返す `price` 関数を作成しています。次に作成しているのは、`cost` と `state` を引数として取る `withTax()` 関数です。ただし、特定のソース・ファイル内では、1 つの州の税金だけを処理することがわかっています。そこで、この関数を呼び出すたびに余分な引数を指定するのではなく、`state` 引数を部分適用し、`state` 値が固定されたバージョンの関数を返すようにします。この `locallyTaxed` 関数は、引数 `cost` だけを取ります。

部分 (制約された) 関数

Scala の `PartialFunction` トレイトは、パターン・マッチングとシームレスに連動するように設計されています (パターン・マッチングについては、連載「関数型の考え方」の記事「Either ツリーとパターン・マッチング」を参照してください)。名前は似ていますが、このトレイトは、部分適用される関数を作成するのではありません。このトレイトを使用することで、定義された値と型のサブセットに対してだけ作用する関数を定義することができます。

部分関数を適用する 1 つの方法は、`case` ブロックを使用することです。リスト 3 では Scala の case を使用していますが、これに対応する従来の `match` 演算子は使用していません。

リスト 3. match のない `case` を使用する

```
val cities = Map("Atlanta" -> "GA", "New York" -> "New York",
  "Chicago" -> "IL", "San Francsico " -> "CA", "Dallas" -> "TX")

cities map { case (k, v) => println(k + " -> " + v) }
```

リスト 3 では、都市と州を対応させたマップを作成した後、このコレクションに対して `map` 関数を呼び出します。すると、`map` がキーと値のペアをばらして出力します。Scala では、コード・ブロックに `case` 文を含めることが、匿名関数を定義する方法の 1 つとなっています。`case` を使用しないほうがより簡潔に匿名関数を定義できますが、`case` 構文を使用すると、リスト 4 に示す付加価値がもたらされます。

リスト 4. `map` と `collect` の違い

```
List(1, 3, 5, "seven") map { case i: Int ? i + 1 } // won't work
// scala.MatchError: seven (of class java.lang.String)

List(1, 3, 5, "seven") collect { case i: Int ? i + 1 }
// verify
assert(List(2, 4, 6) == (List(1, 3, 5, "seven") collect { case i: Int ? i + 1 }))
```

リスト 4 の型が混在するコレクションに対しては、`map` 関数を `case` と一緒に使用することはできません。この関数が文字列 `seven` をインクリメントしようとすると、`MatchError` を受け取ります。一方、`collect` 関数は正常に動作します。この違いと、エラーにならない理由は何なのでしょうか?

case ブロックは部分関数を定義するのであって、関数の部分適用を定義するのではありません。部分関数では、許容値の範囲が限定されます。例えば、数学関数 `1/x` は、`x = 0` の場合は無効です。このように、部分関数には、許容値に対する制約を定義する手段があります。リスト 4 の `collect` の例では、case は `Int` に対して定義されていて、`String` に対しては定義されていません。したがって、文字列 `seven` には collect が適用されないというわけです。

部分関数を定義するには、リスト 5 に示すように、`PartialFunction` トレイトを使用することもできます。

リスト 5. Scala での部分関数の定義

```
val answerUnits = new PartialFunction[Int, Int] {
    def apply(d: Int) = 42 / d
    def isDefinedAt(d: Int) = d != 0
}

assert(answerUnits.isDefinedAt(42))
assert(! answerUnits.isDefinedAt(0))

assert(answerUnits(42) == 1)
//answerUnits(0)
//java.lang.ArithmeticException: / by zero
```

リスト 5 では、`PartialFunction` トレイトから `answerUnits` を派生させて、`apply()` 関数と `isDefinedAt()` 関数の 2 つを提供しています。`apply()` は値を計算する関数です。`isDefinedAt()` (`PartialFunction` の必須メソッド) は、引数の適合性を判別する制約を定義するために使用する関数です。

部分関数は `case` ブロックによっても実装できるため、リスト 5 の `answerUnits` は、さらに簡潔に作成することができます (リスト 6 を参照)。

リスト 6. **`answerUnits`** のもう 1 つの定義方法

```
def pAnswerUnits: PartialFunction[Int, Int] =
    { case d: Int if d != 0 => 42 / d }

assert(pAnswerUnits(42) == 1)
//pAnswerUnits(0)
//scala.MatchError: 0 (of class java.lang.Integer)
```

リスト 6 では、ガード条件と併せて `case` を使用して、値を制約すると同時に結果を提供しています。リスト 5 との顕著な違いは、(`ArithmeticException` ではなく) `MatchError` となっている点です。この違いが生じた理由は、リスト 6 ではパターン・マッチングを使用しているからです。

部分関数を適用できる対象は、数値型に限られているわけではありません。`Any` を含めたすべての型を部分関数で使用することができます。一例として、リスト 7 に示すインクリメンターの実装について検討してみましょう。

リスト 7. Scala でのインクリメンターの定義

```
def inc: PartialFunction[Any, Int] =
    { case i: Int => i + 1 }

assert(inc(41) == 42)
//inc("Forty-one")
//scala.MatchError: Forty-one (of class java.lang.String)

assert(inc.isDefinedAt(41))
assert(! inc.isDefinedAt("Forty-one"))

assert(List(42) == (List(41, "cat") collect inc))
```

リスト 7 では、任意の型 (`Any`) を入力として受け入れるように部分関数を定義していますが、部分関数が作用する対象を型のサブセットに制限しました。ただし、この部分関数では `isDefinedAt()` 関数を呼び出せることにも注意してください。`case` を使用して `PartialFunction`

トレイトを実装する場合、暗黙的に定義されている `isDefinedAt()` を呼び出すことができます。リスト 4 に `map` と `collect` では振る舞いが異なることを示しましたが、その違いは、部分関数の振る舞いから説明がつきます。つまり、`collect` は部分関数を受け入れて、一致しない要素を無視し、該当する要素に対して `isDefinedAt()` を呼び出すように作られています。

Scala の部分関数と関数の部分適用は、呼び名は似ているものの、まったく異なる機能セットを提供しています。例えば、部分関数を部分適用してはならない理由はどこにもありません。

Groovy の場合

Groovy のカリー化と部分適用については、連載「関数型の考え方」の記事「関数型の観点で考える、第 3 回」で、一部の詳細を説明しました。Groovy では、`Closure` クラスに由来する `curry()` 関数を使用してカリー化を実装しています。その名前に反して、`curry()` は基礎となるクロージャー・ブロックを操作して、実際には部分適用を実装します。ただし、部分適用を使用してカリー化を模倣することはできます。その方法は、1 つの関数を、引数を 1 つだけ取るように部分適用された一連の関数に分けるというものです (リスト 8 を参照)。

リスト 8. Groovy での部分適用とカリー化

```
def volume = { h, w, l -> return h * w * l }
def area = volume.curry(1)
def lengthPA = volume.curry(1, 1) //partial application
def lengthC = volume.curry(1).curry(1) // currying

println "The volume of the 2x3x4 rectangular solid is ${volume(2, 3, 4)}"
println "The area of the 3x4 rectangle is ${area(3, 4)}"
println "The length of the 6 line is ${lengthPA(6)}"
println "The length of the 6 line via curried function is ${lengthC(6)}"
```

リスト 8 では、`lengthPA` と `lengthC` で `curry()` 関数を使用して引数の一部を適用しています。ただし、`lengthC` の場合は、1 つの引数を取る関数のチェーンになるまで引数の一部を適用することによって、カリー化に見えるようにしています。

Clojure の場合

Clojure に組み込まれている `(partial f a1 a2 ...)` 関数は、関数 `f` と必要な個数より少ない引数を取って、部分適用された関数を返します。この関数は、残りの引数を指定して呼び出すことができます。リスト 9 に例を 2 つ記載します。

リスト 9. Clojure での部分適用

```
(def subtract-from-hundred (partial - 100))

(subtract-from-hundred 10)      ; same as (- 100 10)
; 90

(subtract-from-hundred 10 20)   ; same as (- 100 10 20)
; 70
```

リスト 9 では、部分適用された「`-`」演算子 (Clojure では演算子も関数です) として `subtract-from-hundred` 関数を定義し、部分適用された引数として 100 を指定しています。リスト 9 の 2 つの例に示されているように、Clojure の部分適用は、引数を 1 つ取る関数にも、複数の引数を取る関数にも有効です。

Clojure は動的に型付けされ、可変引数リストをサポートするため、カリー化は言語の機能として実装されていません。必要なケースは、部分適用が処理します。ただし、Clojure が reducers ライブラリー (「参考文献」を参照) に追加した名前空間のプライベート関数 `(defcurried ...)` によって、このライブラリーに含まれるいくつかの関数を遥かに簡単に定義できるようになっています。Clojure は Lisp から柔軟性を受け継いでいるため、`(defcurried ...)` の適用範囲をさらに広げるのは簡単なことです。

一般的な使用例

定義するのが厄介で、無数の実装詳細があるにも関わらず、カリー化と部分適用は実際のプログラミングに用いられています。

関数ファクトリー

カリー化 (および部分適用) は、従来のオブジェクト指向言語でファクトリー関数を実装するような場面で使用するのに適しています。一例として、リスト 10 に Groovy での単純な `adder` 関数の実装を示します。

リスト 10. Groovy でのアダー (加算関数) とインクリメンター (増分関数)

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

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

リスト 10 では、`adder` 関数から `incrementer` 関数を派生させています。リスト 2 でも同様に、部分適用を使用して、局所的により簡潔なバージョンの関数を作成しています。

Template Method デザイン・パターン

Gang of Four デザイン・パターンのうちの 1 つに、Template Method (テンプレート・メソッド) というパターンがあります。このパターンの目的は、アルゴリズムの大枠をその中で使用する抽象メソッドとともに定義しておくことで、その後の実装に柔軟性を持たせられるようにすることです。部分適用とカリー化は、これと同じ問題を解決することができます。部分適用を使用して既知の振る舞いを指定し、他の引数を実装の詳細で使用するために残しておくことで、このオブジェクト指向のデザイン・パターンの実装を模倣することができます。

暗黙の値

リスト 2 と同様の一般的なケースとして、同じような引数の値を使用して一連の関数を呼び出すことはよくあります。例えば、パーシスタンス・フレームワークを操作するときには、1 番目の引数としてデータ・ソースを渡さなければなりません。この場合、部分適用を使用すれば、値を暗黙的に指定することができます (リスト 11 を参照)。

リスト 11. 部分適用による暗黙の値の指定

```
(defn db-connect [data-source query params]
      ...)

(def dbc (partial db-connect "db/some-data-source"))

(dbc "select * from %1" "cust")
```

リスト 11 では、データ関数にアクセスするためにコンビニエンス関数 `dbc` を使用しています。この場合、データ・ソースを指定する必要はありません。データ・ソースは自動的に指定されるからです。オブジェクト指向プログラミングの核心でもある、あらゆる関数に魔法のように現れる暗黙的 `this` コンテキストの概念は、カリー化を使用して実装することができ、それによって使用者には見えないようにして `this` をすべての関数に指定することができます。

まとめ

すべての Java.next 言語には、さまざまな形でカリー化と部分適用が出現します。このいずれかの手法を使用すれば、より簡潔な関数定義をすることも、暗黙の値を指定することも、関数ファクトリーを作成することもできます。

次回の記事では、すべての Java.next 言語の関数型プログラミングの機能に共通する意外な共通点を、言語によって大幅に異なることもある実装の詳細と併せて紹介します。

関連トピック

- Scala: Scala は JVM 上で実行される最近の関数型言語です。
- Groovy は、Java 言語の動的バージョンとして Java の構文と機能が更新されたものです。
- Clojure: Clojure は JVM 上で実行される最近の関数型 Lisp です。
- 「What's the difference between Currying and Partial Application?」: 人気のある開発者ブログである Raganwald では、カリー化と部分適用との違いを明確に示しています。
- reducers: reducers ライブラリーは、Clojure の強力な拡張機能であり、このライブラリーを使用することで map などの処理が高度な同時アクセスを実現できるようになります。
- developerWorks Java technology ゾーン: Java プログラミングのあらゆる側面を網羅した豊富な記事を調べてください。