Java.next: 継承を伴わない拡張、第 1 回

Groovy、Scala、Clojure でクラスに振る舞いを追加する方法を探る

Neal Ford
Director / Software Architect / Meme Wrangler
ThoughtWorks Inc.

2013年 7月 11日

Groovy、Scala、および Clojure には多くの拡張メカニズムが用意されている一方、継承となると、事実上 Java 言語の継承が唯一の手段となります。今回の記事では、Java.next 言語での Java クラス拡張手段としてのカテゴリー・クラス、`ExpandoMetaClass`、暗黙の型変換、プロトコルについて見ていきます。

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。

Java 言語の設計では、その前身となる言語で認識されていた問題を回避するために、意図的に排除されたものがあります。例えば、Java 言語の設計者達は、C++ での多重継承があまりにも複雑であると感じていたことから、多重継承は設計に含めないようにしました。実際、Java 言語には、拡張するための手段がわずかしか組み込まれておらず、拡張する場合は単一継承とインターフェースに頼ることになります。

一方、Java.next 言語を含む他の言語には、豊富な拡張手段があります。今回の記事から 3 回にわたり、継承を伴わない Java クラスの拡張方法を探ります。今回の記事では、既存のクラスに直接、または構文糖を使用してメソッドを追加する方法について見ていきます。

Expression Problem

Expression Problem とは、ベル研究所の Philip Wadler 氏が書いた未発表の論文 (「参考文献」を参照) に端を発する、最近のコンピューター・サイエンスの歴史から引き出された有名な見解です (これについては、Stuart Sierra 氏が developerWorks に寄稿した記事「Solving the Expression Problem with Clojure 1.2」の中でその詳細をわかりやすく説明しています)。Wadler 氏はその論文のなかでこう述べています。

© Copyright IBM Corporation 2013

> Expression Problem とは、昔からある問題に付けられた新しい名前です。この問題で目指していることは、データ型を場合分けして定義することで、そのデータ型には新しい場合を追加することや、新しい関数を追加することができる (しかも、コードを再コンパイルすることなく行える) 一方で、静的な型の安全性は保たれる (例えば、キャストは使用されません) ようにすることです。

別の言葉に置き換えると、型変換や `if` 文を使わずに、階層内のクラスに機能を追加するにはどうすればよいのか、という問題です。

現実の世界で Expression Problem がどのように生じるかを簡単な例で示します。例えば、皆さんの会社では、アプリケーションで使われる長さの単位をメートルで表すことを前提にしていて、他の単位向けの機能は一切クラスに組み込んでいないとします。ある日、会社がライバル会社と合併することになりました。そのライバル会社では常に、長さの単位としてフィートを前提としています。

この問題を解決する 1 つの方法は、`Integer` を変換メソッドで増補することによって、2 つの単位形式を簡単に切り替えられるようにすることです。最近の言語には、この方法を実現するソリューションがいくつか用意されています。それらのソリューションのうち、この記事では次の 3 つに焦点を当てます。

- オープン・クラス
- ラッパー・クラス
- プロトコル

Groovy のカテゴリーと `ExpandoMetaClass`

Groovy には、オープン・クラスを使用して既存のクラスを拡張するための手段が 2 つ含まれています。オープン・クラスでは、クラス定義を「再オープン」して、変更 (例えば、メソッドの追加、変更、または削除) を加えることができます。

カテゴリー・クラス

カテゴリー・クラスは、Objective-C から取り入れられた概念であり、静的メソッドを持つ通常のクラスのことです。各メソッドは、そのメソッドが増補する型を表す 1 つ以上の引数を取ります。例えば、Integer に対するメソッドを追加するには、その型 (`Integer`) を第 1 引数として取る静的メソッドが必要になります (リスト 1 を参照)。

リスト 1. Groovy のカテゴリー・クラス

```
class IntegerConv {
  static Double getAsMeters(Integer self) {
    self * 0.30480
  }

  static Double getAsFeet(Integer self) {
    self * 3.2808
  }
}
```

リスト 1 の `IntegerConv` クラスには、このクラスを増補する 2 つのメソッドが含まれています。それぞれのメソッドは、`self` (一般的な命名規則による名前) という名前の `Integer` 型の引数を

取ります。これらのメソッドを使用するには、`use` ブロックに、各メソッドを参照するコードをラップする必要があります (リスト 2 を参照)。

リスト 2. カテゴリー・クラスを使用する

```
@Test void test_conversion_with_category() {
  use(IntegerConv) {
    assertEquals(1 * 3.2808, 1.asFeet, 0.1)
    assertEquals(1 * 0.30480, 1.asMeters, 0.1)
  }
}
```

リスト 2 には、特に興味深い点が 2 つあります。まず、リスト 1 での拡張メソッドの名前は `getAsMeters()` ですが、このメソッドを `1.asMeters` のように呼び出しています。Java のプロパティーに対する Groovy の構文糖により、`getAsMeters()` メソッドを `asMeters` という名前のクラス・フィールドであるかのようにして実行することができます。拡張メソッドの `as` を省略する場合は、拡張メソッドの呼び出しには `1.asMeters()` のように空の括弧が必要になります。概して、プロパティーの構文は簡潔であるほうが私の好みです。そしてこれが、ドメイン特化言語 (DSL) を作成する際の一般的な手法となっています。

リスト 2 でもう 1 つ注目する点は、`asFeet` および `asMeters` の呼び出しです。use ブロック内では、組み込みメソッドと同じように新規メソッドを呼び出しています。拡張は、`use` ブロックのレキシカル・スコープの中では透過的です。このブロックは増補のスコープをクラス (場合によってはコアのクラス) に制限するので便利です。

`ExpandoMetaClass`

カテゴリーは、Groovy が追加した最初の拡張メカニズムです。けれども、カテゴリーのレキシカル・スコープでは、Groovy ベースの Web フレームワークである Grails を作成するには制約がありすぎることが判明しました。カテゴリーの制約に不満を感じていた Grails 作成者の一人、Graeme Rocher 氏は、別の拡張メカニズムを Groovy に追加しました。それが、`ExpandoMetaClass` です。

`ExpandoMetaClass` は、あらゆるクラスから「派生」することができ、必要になってからインスタンス化される拡張ホルダーです。リスト 3 に、`ExpandoMetaClass` を使用して `Integer` クラスの拡張を実装する方法を示します。

リスト 3. `ExpandoMetaClass` を使用して `Integer` を拡張する

```
class IntegerConvTest{

  static {
    Integer.metaClass.getAsM { ->
      delegate * 0.30480
    }

    Integer.metaClass.getAsFt { ->
      delegate * 3.2808
    }
  }

  @Test void conversion_with_expando() {
    assertTrue 1.asM == 0.30480
    assertTrue 1.asFt == 3.2808
  }
}
```

リスト 3 では、`metaClass` ホルダーを使用して `asM` および `asFt` プロパティーを追加しています。ここで使用している命名規則はリスト 2 と同じです。メタクラスの呼び出しは、テスト・クラスの静的イニシャライザー内で行われます。これは、拡張メソッドが検出される前に、増補されることを確実にしなければならないためです。

カテゴリー・クラスと `ExpandoMetaClass` は両方とも、組み込みメソッドの前に拡張クラスのメソッドを呼び出します。これにより、既存のメソッドの追加、変更、または削除が可能になります。リスト 4 に一例を記載します。

リスト 4. 既存のメソッドに優先する拡張クラス

```
@Test void expando_order() {
  try {
    1.decode()
  } catch(NullPointerException ex) {
    println("can't decode with no parameters")
  }
  Integer.metaClass.decode { ->
    delegate * Math.PI;
  }
  assertEquals(1.decode(), Math.PI, 0.1)
}
```

リスト 4 で最初に呼び出している `decode()` メソッドは、整数のエンコード方法を変更することを目的とした Groovy の組み込み静的メソッドです。一般に、このメソッドは単一の引数を取ります。引数なしで呼び出された場合、このメソッドは `NullPointerException` をスローします。ただし、`Integer` クラスを私独自の `decode()` メソッドで増補すると、このメソッドが元のメソッドに取って代わるようになります。

Scala の暗黙の型変換

Scala では、Expression Problem が持つ、既存のクラスを簡潔に拡張するという課題に、ラッパー・クラスを使用するという方法で対処します。クラスにメソッドを追加するには、そのメソッドをヘルパー・クラスに追加し、元のクラスからヘルパー・クラスへの暗黙の型変換を記述します。型変換が行われると、見た目にはわかりませんが、元のクラスではなくヘルパー・クラスのメソッドが呼び出されるようになります。リスト 5 の例では、この手法を使用しています。

リスト 5. Scala の暗黙の型変換

```
class UnitWrapper(i: Int) {
  def asFt = {
    i * 3.2808
  }

  def asM = {
    i * 0.30480
  }
}

implicit def unitWrapper(i:Int) = new UnitWrapper(i)

println("1 foot = " + 1.asM + " meters");
println("1 meter = " + 1.asFt + "foot")
```

リスト 5 では、`UnitWrapper` という名前のヘルパー・クラスを定義しています。コンストラクターの引数を 1 つ取るこのクラスには、`asFt` と `asM` という 2 つのメソッドがあります。値を変換するヘルパー・クラスを用意した後、`implicit def` を作成して、新しい `UnitWrapper` をインスタンス化します。メソッドを呼び出すには、元のクラスのメソッドであるかのようにメソッド (例えば、`1.asM`) を呼び出します。Scala は `Integer` クラスの `asM` メソッドを検出できない場合には、呼び出し側のクラスを、ターゲット・メソッドを持つクラスへと変換するための暗黙の変換が存在していないかどうかをチェックします。Groovy と同じように、Scala にもメソッド呼び出しの括弧を省略できるようにする構文糖がありますが、これは命名規則ではなく、言語の機能として存在します。

Scala での変換ヘルパーは、一般にクラスではなく `object` ですが、私はクラスを使用しています。これは、値をコンストラクターの引数として渡したいためです (`object` では、コンストラクターの引数として値を渡すことはできません)。

Scala での暗黙の型変換は、既存のクラスを増補するための簡潔でタイプ・セーフな方法ですが、このメカニズムでは、オープン・クラスでのように既存のメソッドを変更または削除することはできません。

Clojure のプロトコル

Clojure では、Expression Problem が持つ、既存のクラスを簡潔に拡張するという課題に、これまで説明したのとはまた別の手法で対処しており、`extend` 関数と Clojure プロトコル抽象化の組み合わせを使用します。プロトコルとは、概念的には Java インターフェースのようなもので、実装のないメソッド・シグニチャーのコレクションです。Clojure は本来、オブジェクト指向ではなく関数型の言語ですが、クラスを操作 (および拡張) したり、メソッドを関数にマッピングしたりすることができます。

数値を拡張して変換を追加するために、2 つの関数 (`asF` と `asM`) を含めたプロトコルを定義します。このプロトコルを使用すれば、`extend` 関数によって既存のクラス (例えば、`Number`) を拡張することができます。`extend` 関数は、ターゲット・クラスを最初の引数に取り、プロトコルを、キー (関数名) と値 (関数の実装 (匿名関数としての実装)) のマップとセットにしたものを 2 番目の引数に取ります。リスト 6 に、Clojure による単位の変換を示します。

リスト 6. Clojure のプロトコルによる拡張

```
(defprotocol UnitConversions
  (asF [this])
  (asM [this]))

(extend Number
  UnitConversions
  {:asF (fn [this] (* this 3.2808))
   :asM #(* % 0.30480)})
```

この新しい拡張を以下のように Clojure の REPL (インタラクティブな Read-Eval-Print Loop) で使用して、変換を確認することができます。

```
user=> (println "1 foot is " (asM 1) " meters")
1 foot is  0.3048  meters
```

リスト 6 では、2 つの変換関数の実装に、匿名関数を宣言する際の 2 つのバージョンの構文が示されています。それぞれの関数は引数を 1 つ取ります (`asF` 関数では `this`)。引数を 1 つ取る関数は一般的であるため、Clojure にはこのような関数を作成するための構文糖があります。この構文糖は、`%` がパラメーターのプレースホルダーとなっている `asM` 関数に示されています。

プロトコルは、既存のクラスにメソッドを (関数として) 追加するための単純なソリューションとなります。さらに、Clojure には、複数の拡張を 1 つのグループに統合するための便利なマクロもいくつかあります。例えば、Compojure Web フレームワーク (「参考文献」を参照) では、プロトコルを使用して各種の型を拡張することによって、それぞれの型にそのレンダリング方法を「把握」させています。リスト 7 に、Compojure の `Renderable` 定義からの抜粋を記載します。

リスト 7. プロトコルを使用して多数の型を拡張する

```
(defprotocol Renderable
  (render [this request]
    "Render the object into a form suitable for the given request map."))

(extend-protocol Renderable
  nil
  (render [_ _] nil)
  String
  (render [body _]
    (-> (response body)
        (content-type "text/html; charset=utf-8")))
  APersistentMap
  (render [resp-map _]
    (merge (with-meta (response "") (meta resp-map))
           resp-map))
  IFn
  (render [func request]
    (render (func request)
  ; . . .
```

リスト 7 の `Renderable` プロトコルは、値とリクエスト・マップを引数に取る 1 つの `render` 関数とともに定義されています。Clojure の `extend-protocol` マクロでは、複数のプロトコル定義を 1 つのグループにまとめることができます。このマクロが引数に取るのは、型と実装のペアです。Clojure では、関心のない引数をアンダースコアで置き換えることができます。このプロトコル定義のリスト 7 に記載されている部分では、`nil`、`String`、`APersistentMap`、および `IFn` (Clojure の関数にとってコアとなるインターフェース) に対するレンダリング命令が指定されています (このフレームワークには、他にも多くの型がありますが、スペースを節約するために上記のリストでは省略しています)。このように、レンダリングする必要があるすべての型に対して、意味と拡張を一緒に定義できると、実際にコードを作成する際に大いに役立ちます。

まとめ

今回の記事では、Expression Problem について紹介し、この問題が持つ既存のクラスを簡潔に拡張するという課題に Java.next 言語ではどのように対処しているかを詳しく探りました。3 つの言語が使用する手法はそれぞれに異なりますが (Groovy ではオープン・クラスを、Scale ではラッパー・クラスを使用し、Clojure ではプロトコルを実装します)、その結果は同様です。

ただし、Expression Problem は型を増補するだけにとどまりません。次回の記事では引き続き拡張について探り、他のプロトコルの機能、特徴、およびミックスインを取り上げます。

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 ThoughtWorks のディレクターであり、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、最新の著作は『Presentation Patterns』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。

© Copyright IBM Corporation 2013
(www.ibm.com/legal/copytrade.shtml)
商標
(www.ibm.com/developerworks/jp/ibm/trademarks/)