Java.next: ミックスインとトレイト

Groovy と Scala のクラスに新しい振る舞いをミックスインする

Neal Ford

2013年 10月 24日

Java 言語の主要なパラダイムである「単一継承を使用したオブジェクト指向」は、プログラミング問題の大半を効果的にモデル化しますが、すべての問題をカバーするわけではありません。Java.next 言語は、このパラダイムをさまざまな方法で拡張します。その方法には、ミックスインとトレイトも含まれます。連載「Java.next」の今回の記事では、ミックインとトレイトに共通するメカニズムを明らかにし、Groovy のミックスインと Scala のトレイトとの微妙な違いについて探ります。

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 言語の開発者たちは、「多重継承」を使用する C++ などの言語に精通していました。多重継承では、クラスは任意の数の親を継承することができますが、継承された機能がどちらの親から派生されたものなのかを判別できないという問題があります。この問題は、「菱形継承問題」と呼ばれています (「参考文献」を参照)。Java 言語の設計者が単一継承とインターフェースという手段を選択する動機となったのは、多重継承に内在するこの菱形継承問題やその他の複雑さです。

インターフェースはセマンティクスを定義しますが、振る舞いは定義しません。インターフェースはメソッドのシグニチャーとデータの抽象化を定義するのに適しており、すべての Java.next 言語が Java のインターフェースを本質的な変更なしでサポートしています。しかし、横断的関心事となると、単一継承とインターフェースのモデルには適合しないため、Java 言語にはアスペクト指向プログラミングなどの外部メカニズムが必要となりました。Java.next 言語のうち、Groovy と Scala は、ミックスインまたはトレイトと呼ばれる言語構成体を使用して、別の拡張レベルで横断的関心事に対処しています。この記事では Groovy のミックスインと Scala のトレイトについて説明し、それぞれの使用方法を紹介します (Clojure では、これとほとんど同じような機能に対処す

るためにプロトコルを使用しています。それについては、「Java.next: 継承を伴わない拡張、第 2 回」で取り上げています)。

ミックスイン

アイスクリームから得た発想

Flavors 言語 (「参考文献」を参照) で初めて登場したミックスインの概念は、この言語の開発が行われたオフィスの近くにあるアイスクリーム店から発想を得たものです。そのアイスクリーム・パーラーでは、プレーンのアイスクリームにお客が望む「ミックスイン」(チョコバーを砕いたもの、スプリンクル、ナッツなど) をトッピングとして追加して提供していました。

初期のオブジェクト指向言語のなかには、クラスの属性とメソッドをまとめて 1 つのコード・ブロックで定義した時点でクラス定義が完成するものもありましたが、開発者が複数の属性を 1 箇所で定義し、メソッドの定義は後に遅らせて、適切なタイミングでそれらをクラスに「ミックスインする」(混ぜ合わせる) ことができるものもあります。オブジェクト指向言語が進化するにつれ、最新の言語でのミックスインの機能の詳細も同じく進化してきました。

Ruby、Groovy、およびこの 2 つと同様の言語では、ミックスインがインターフェースと親クラスの中間にある物として、既存のクラス階層を増補します。インターフェースと同じく、ミックスインも `instanceof` チェックの型としての役割を果たし、インターフェースと同じ拡張ルールに従います。1 つのクラスに適用できるミックスインの数に制限はありません。一方、インターフェースとは異なり、ミックスインはメソッドのシグニチャーを指定するだけでなく、シグニチャーの振る舞いを実装することもできます。

ミックスインを導入した当初の言語では、ミックスインに含めることができるのはメソッドのみであり、メンバー変数などの状態を含めることはできませんでしたが、今では、Groovy をはじめとする多くの言語でステートフルなミックスインを導入しています。Scala のトレイトもステートフルに振る舞います。

Groovy のミックスイン

Groovy でミックスインを実装するには、`metaClass.mixin()` メソッドまたは `@Mixin` アノテーションのいずれかを使用します (`@Mixin` アノテーションの場合、さらに Groovy の AST (Abstract Syntax Tree: 抽象構文木) 変換を使用して、必要なメタプログラミングを関連付けます)。リスト 1 に、`File` クラスで圧縮 ZIP ファイルを作成できるようにするために `metaClass.mixin()` を使用する例を示します。

リスト 1. `File` クラスへの `zip()` メソッドのミックスイン

```
class Zipper {

  def zip(dest) {
      new ZipOutputStream(new FileOutputStream(dest))
          .withStream { ZipOutputStream zos ->
            eachFileRecurse { f ->
              if (!f.isDirectory()) {
                zos.putNextEntry(new ZipEntry(f.getPath()))
                new FileInputStream(f).withStream { s ->
                    zos << s
                    zos.closeEntry()
                }
```

```
              }
            }
          }
  }

  static {
    File.metaClass.mixin(Zipper)
  }

}
```

リスト 1 では `Zipper` クラスを作成し、このクラスに新規 `zip()` メソッドと、この新規メソッドを既存の `File` クラスに追加するための接続の両方を含めています。`zip()` メソッドの (平凡な) Groovy コードは、再帰的に ZIP ファイルを作成します。新規メソッドを既存の `File` クラスに組み込むために上記リストの最後の部分では、static イニシャライザーを使用しています。Java 言語の場合と同じく、static クラス・イニシャライザーは、クラスのロード時に実行されます。static イニシャライザーは、拡張部分を利用するコードよりも前に実行されることが保証されるため、増補したコードの組み込みを行うにはうってつけの場所です。リスト 1 では、この static イニシャライザーの中で `mixin()` メソッドによって `File` クラスに `zip()` メソッドを追加しています。

「Java.next: 継承を伴わない拡張、第 1 回」で、`ExpandoMetaClass` とカテゴリー・クラスという Groovy のメカニズムを取り上げました。この 2 つのメカニズムを使用することで、既存のクラスのメソッドを追加、変更、削除することができます。`mixin()` を使用してメソッドを追加しても `ExpandoMetaClass` またはカテゴリー・クラスでメソッドを追加する場合と同じ結果になりますが、その実装は異なります。リスト 2 のミックスインの例を検討してみましょう。

リスト 2. 継承階層を操作するミックスイン

```
import groovy.transform.ToString

class DebugInfo {
  def getWhoAmI() {
    println "${this.class} <- ${super.class.name}
    <<-- ${this.getClass().getSuperclass().name}"
  }
}

@ToString class Person {
  def name, age
}

@ToString class Employee extends Person {
  def id, role
}

@ToString class Manager extends Employee {
  def suiteNo
}


Person.mixin(DebugInfo)

def p = new Person(name: "Pete", age: 33)
def e = new Employee(name: "Fred", age: 25, id:"FRE", role:"Manager")
def m = new Manager(name: "Burns", id: "001", suiteNo: "1A")

p.whoAmI
e.whoAmI
m.whoAmI
```

リスト 2 では、`DebugInfo` というクラスを作成して、そこに単一の `getWhoAmI` プロパティー定義を含めています。このプロパティー内で、クラスの詳細 (現行のクラス、および `super` プロパティーと `getClass().getSuperClass()` プロパティー両方の親子関係に関するパースペクティブ) を出力します。その次に作成しているのは、`Person`、`Employee`、`Manager` からなる単純なクラス階層です。

続いて、階層の最上位にある `Person` クラスに `DebugInfo` クラスをミックスインします。`whoAmI` プロパティーは `Person` クラスのために存在することから、その子クラスにもこのプロパティーは存在します。

出力を見ると、(驚くかもしれませんが) `DebugInfo` クラスが継承階層に入り込んでいることがわかります。

```
class Person <- DebugInfo <<-- java.lang.Object
class Employee <- DebugInfo <<-- Person
class Manager <- DebugInfo <<-- Employee
```

ミックスイン・メソッドは、Groovy がすでに持つ、メソッド解決のための複雑な関係の中に収まらなければなりません。リスト 2 の親クラスへのさまざまな戻り値に、その複雑な関係が反映されています。この記事では、メソッド解決の詳細については説明しませんが、ミックスイン・メソッド内で `this` と `super` の値を (さまざまな形で) 使用する際には慎重に行ってください。

カテゴリー・クラスまたは `ExpandoMetaClass` を使用しても、継承には影響しません。その場合、クラスに対して変更を加えるのであって、新しい異なる振る舞いを追加するわけではないからです。ただし、これらの変更は、個別のカテゴリー成果物として識別できないという欠点があります。カテゴリー・クラスまたは `ExpandoMetaClass` を使用して、複数のクラスに共通で 3 つのメソッドを追加した場合、どの特定のコード成果物 (インターフェースやクラス・シグニチャーなど) も、追加によって新たに存在するようになった共通性を識別しません。ミックスインの利点は、Groovy ではミックスインを使用するすべてのものをカテゴリーとして扱うことです。

カテゴリー・クラスを実装する際の頭痛の種の 1 つは、その厳格なクラス構造です。このクラス構造では、すべての静的メソッド (そのそれぞれが引数を少なくとも 1 つとります) を使用して、増補される型を表さなければなりません。メタプログラミングは、このようなボイラープレート・コードを排除するのにとりわけ役立ちます。`@Mixin` アションの出現により、極めて簡単にカテゴリーを作成してクラスにミックスインすることができるようになっています。リスト 3 (Groovy のドキュメントからの抜粋) に、カテゴリーとミックスインの相乗効果を示します。

リスト 3. カテゴリーとミックスインとの組み合わせ

```
interface Vehicle {
    String getName()
}

@Category(Vehicle) class Flying {
    def fly() { "I'm the ${name} and I fly!" }
}

@Category(Vehicle) class Diving {
    def dive() { "I'm the ${name} and I dive!" }
}
```

```
@Mixin([Diving, Flying])
class JamesBondVehicle implements Vehicle {
    String getName() { "James Bond's vehicle" }
}

assert new JamesBondVehicle().fly() ==
       "I'm the James Bond's vehicle and I fly!"
assert new JamesBondVehicle().dive() ==
       "I'm the James Bond's vehicle and I dive!"
```

リスト 3 では、単純な `Vehicle` インターフェースと 2 つのカテゴリー・クラス (`Flying` および `Diving`) を作成しました。ボイラープレート・コードの要件には、`@Category` アノテーションが対処します。カテゴリーを定義した後、これらのカテゴリーを `JamesBondVehicle` にミックスインすることによって、両方のカテゴリーの振る舞いを追加します。

Groovy のカテゴリー、`ExpandoMetaClass`、およびミックスインのすべてに共通するのは、言語の活発な進化によってもたらされた必然的な結果であるという点です。この 3 つの手法には、重複する部分がかなりありますが、それぞれに、その手法でなければ最適に処理することができない得意分野があります。Groovy が初めから作り直されるとしたら、その作成者たちはこの 3 つの手法の機能の多くを 1 つのメカニズムに統合することでしょう。

Scala のトレイト

Scala では、コードの再利用を実装する手段としてトレイトを使用します。トレイトは、ミックスインと同様のコアとなる言語機能です。Scala のトレイトはステートフルであり (メソッドとフィールドの両方を含めることができます)、Java 言語でインターフェースが果たすのと同じ `instanceof` の役割を果たします。トレイトとミックスインが解決する問題の多くは共通していますが、トレイトにはミックスインよりも高いレベルの言語的厳密性があります。

「Java.next: Groovy、Scala、Clojure の共通点、第 1 回」では、Scala での演算子の多重定義を説明するために複素数クラスを使用しました。その複素数クラスにはブール比較演算子を実装しませんでしたが、それは、Scala の組み込み `Ordered` トレイトによって簡単に実装できるためです。リスト 4 に、`Ordered` トレイトを利用して改善した複素数クラスを記載します。

リスト 4. 比較可能な複素数

```
final class Complex(val real: Int, val imaginary: Int) extends Ordered[Complex] {
  require (real != 0 || imaginary != 0)

  def +(operand: Complex) =
      new Complex(real + operand.real, imaginary + operand.imaginary)

  def +(operand: Int) =
    new Complex(real + operand, imaginary)

  def -(operand: Complex) =
    new Complex(real - operand.real, imaginary - operand.imaginary)

  def -(operand: Int) =
    new Complex(real - operand, imaginary)


  def *(operand: Complex) =
      new Complex(real * operand.real - imaginary * operand.imaginary,
          real * operand.imaginary + imaginary * operand.real)
```

```
  override def toString() =
      real + (if (imaginary < 0) "" else "+") + imaginary + "i"

  override def equals(that: Any) = that match {
    case other : Complex => (real == other.real) && (imaginary == other.imaginary)
    case _ => false
  }

  override def hashCode(): Int =
    41 * ((41 + real) + imaginary)

  def compare(that: Complex) : Int = {
    def myMagnitude = Math.sqrt(this.real ^ 2 + this.imaginary ^ 2)
    def thatMagnitude = Math.sqrt(that.real ^ 2 + that.imaginary ^ 2)
    (myMagnitude - thatMagnitude).round.toInt
  }
}
```

リスト 4 では `>`、`<`、`<=`、および `>=` 演算子を実装していませんが、リスト 5 に記載するように、これらの演算子は複素数のインスタンスで呼び出すことができます。

リスト 5. 比較のテスト

```
class ComplexTest extends FunSuite {

  test("comparison") {
    assert(new Complex(1, 2) >= new Complex(3, 4))
    assert(new Complex(1, 1) < new Complex(2,2))
    assert(new Complex(-10, -10) > new Complex(1, 1))
    assert(new Complex(1, 2) >= new Complex(1, 2))
    assert(new Complex(1, 2) <= new Complex(1, 2))
  }

}
```

複素数を比較するために数学上定義されている手法は 1 つも存在しないため、リスト 4 では一般に受け入れられているアルゴリズムを使用して数値の大きさを比較しています。クラス定義を継承するために使用している `Ordered[Complex]` トレイトは、パラメーター化されたクラスを対象としたブール演算子をミックスインします。このトレイトが機能するには、ミックスインされた演算子が 2 つの複素数を比較する必要があります。それが、`compare()` メソッドの目的です。`Ordered` トレイトを継承しようとして、必要なメソッドを提供しなかったとすると、必要なメソッドが欠落しているため、クラスを `abstract` として宣言しなければならないことが、コンパイラー・メッセージによって通知されます。

Scala では、トレイトに明確な 2 つの役割が定義されています。それは、インターフェースを強化する役割と、スタック可能な変更を実行する役割です。

インターフェースの強化

Java 開発者はインターフェースを設計するときに、利便性に依存する難問に直面します。その難問とは、多数のメソッドを含めたリッチ・インターフェースを作成するか、それとも少数のメソッドだけを含めたシン・インターフェースを作成するかの決定です。リッチ・インターフェースは多彩なメソッドを提供することから、使用者にとってはリッチ・インターフェースのほうが便利ですが、メソッドの数が多ければ、インターフェースを実装するのが難しくなります。シン・インターフェースには、それとは逆の問題があります。

このリッチかシンかのジレンマを解決するのが、トレイトです。シン・インターフェースでコア機能を作成した後、さらにリッチな機能を提供するために、トレイトを使用してインターフェースを増補することができます。例えば、Scala では `Set` トレイトによってセットの共有機能が実装され、開発者が選択したサブトレイト — `mutable` または `immutable` — によってセットが可変であるかそうでないかが決まります。

スタック可能な変更

Scala におけるトレイトのもう 1 つの一般的な使い方は、スタック可能な変更です。トレイトでは、既存のメソッドを変更することも、新しいメソッドを追加することも可能であり、さらには前のトレイトの実装にチェーンバックするためのアクセス手段が `super` によって提供されています。

リスト 6 に、数値のキューを使用することによる、スタック可能な変更を示します。

リスト 6. スタック可能な変更の作成

```
abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}

import scala.collection.mutable.ArrayBuffer

class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}

trait Squaring extends IntQueue {
  abstract override def put(x: Int) { super.put(x * x) }
}
```

リスト 6 では、単純な `IntQueue` クラスを作成した後、`ArrayBuffer` が含まれる可変バージョンを作成しています。`Squaring` トレイトは `IntQueue` を継承し、値がキューに挿入されると自動的にその値を二乗します。スタック内で前のトレイトにアクセスするには、`Squaring` トレイト内で `super` を呼び出します。オーバーライドされたメソッド (最初のメソッドを除く) が `super` を呼び出す限り、変更が次々とスタックされていきます (リスト 7 を参照)。

リスト 7. スタックされるインスタンスを作成する

```
object Test {
  def main(args: Array[String]) {
    val queue = (new BasicIntQueue with Squaring)
    queue.put(10)
    queue.put(20)
    println(queue.get()) // 100
    println(queue.get()) // 400
  }
}
```

リスト 6 での `super` の使い方は、トレイトとミックスインとの重要な違いを明らかにしています。ミックスインは — 元のクラスを作成した後に (文字通り) ミックスインするため — クラス階

層内での現在の位置が曖昧になる可能性に対処しなければなりません。トレイトは、クラスが作成されるときに線形化されます。したがって、コンパイラーは曖昧さなしに `super` が何であるのかという疑問を解決します。Scala での線形化の仕組みは、厳密に定義された複雑なルール (このルールについては、この記事では説明しません) によって制御されます。トレイトはまた、Scala での菱形継承問題も解決します。Scala がメソッドの継承元と解決を追跡する際に、曖昧さは許されません。なぜなら、この言語では解決を扱うための明示的なルールを定義しているためです。

まとめ

今回の記事では、(Groovy における) ミックスインと (Scala における) トレイトの共通点と相違点を探りました。ミックスインとトレイトは同様の機能を数多く提供しますが、実装の詳細は異なります。その違いは、言語によって異なる設計思想を表す重要な点での違いです。Groovy では、ミックスインはアノテーションとして存在し、AST 変換が提供する強力なメタプログラミング機能を使用します。ミックスイン、カテゴリー・クラス、`ExpandoMetaClass` はいずれも機能の点で重複していますが、微妙な (そして重要な) 違いがあります。Scala でのトレイト (`Ordered` など) は、Scala の組み込み機能のほとんどが依存するコアの言語機能を形成します。

次回の記事では、Java.next でのカリー化と部分適用について取り上げます。

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。
- Clojure: Clojure は JVM 上で実行される最近の関数型 Lisp です。
- 多重継承: この Wikipedia の記事で、多重継承言語で菱形継承問題が生じる理由を学んでください。
- ミックスイン: ミックスインについて (この言葉の起源も含めて) の詳細と Flavors 言語についての詳細を Wikipedia で読んでください。
- 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。