多忙な Java 開発者のための Scala ガイド: trait と振る舞い

Scala バージョンの Java インターフェースを使う

Ted Neward
Principal
Neward & Associates

2008年 4月 29日

Scala は単に関数型の概念を JVM にもたらすだけではなく、オブジェクト指向言語の設計に対する新しい観点を提供してくれます。「多忙な Java 開発者のための Scala ガイド」シリーズの今回の記事では、Scala では trait を活用することでオブジェクトを単純かつ容易に作成できることを Ted Neward が解説します。この記事で学ぶように、trait には Java™ のインターフェースと C++ の多重継承という、従来提供されている両極端のものと似ている面、異なっている面があります。

このシリーズの他の記事を見る

有名な科学者であり研究者でもある Isaac Newton 卿は、「もし私が他の人よりも遠くを見ているとしたら、それは巨人の肩 (先人の業績) の上に立っているからだ」という言葉で知られています。私は熱心な歴史学者、政治科学者として、この偉人の言葉を少し変え、「もし私が他の人よりも遠くを見ているとしたら、それは私が歴史という肩の上に立っているためである」と言いたいと思います。またこの言葉は、歴史学者である George Santayana による別の言葉、「過去を忘れる者はその過去を繰り返す運命を負わされる」を反映しています。言い換えれば、もし私達が歴史を振り返らず、先人 (私達自身を含みます) の過ちから学ぶことができないとしたら、改善の可能性はほとんどありません。

さて皆さんは、こうした哲学的な議論と Scala とがどう関係するのか不思議に思うことでしょう。1 つは継承の問題です。Java 言語は約 20 年前、「オブジェクト指向」の全盛期に作られたという事実を考えてください。Java 言語は、当時最も優勢な言語であった C++ の開発者達を Java プラットフォームに引きつけようという露骨な試みとして、C++ を真似て作られました。当時は明白で必要と思われたいくつかの決定がなされましたが、振り返ってみると、そうした決定の一部は、当時それを作成した人達が信じていたほど有効ではなかったことがわかりました。

例えば、20 年前には Java 言語の作成者達が C++ スタイルの private 継承と多重継承のどちらも拒否することは適切でした。しかしその頃から後、多くの Java 開発者達は、そうした決定を後悔する事態を経験してきました。Scala へのガイドの今回は、Java 言語での多重継承と private 継承の歴史を再検証します。そして、そうした歴史を Scala がどのように書き換え、私達全員の利益のために何をしようとしているのかを説明します。

C++ と Java 言語での継承

> 歴史とは人々が同意した過去の出来事に対する見解である
> — Napoleon Bonaparte

C++ を使って作業したことのある人であれば、private 継承は is-a 関係を明示的に受け入れずに基本クラスの振る舞いを吸収するための 1 つの方法であることを覚えているでしょう。基本クラスを「private」とすることで、派生クラスは実際に基本クラスの 1 つにならずに基本クラスから継承することができました。しかし private 継承そのものは、よくありがちな、あまり有効に使われない機能の 1 つでした。基本クラスへのダウンキャストや派生元へのアップキャストができないにもかかわらず基本クラスから継承するという考えは、ばかげたものに思えたのです。

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。developerWorks の、この新しいシリーズでは、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 プログラミング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 コードで可能ならば、手間をかけて Scala を学ぶ必要はないのです。

一方、多重継承はオブジェクト指向プログラミングに必要な要素と一般的に考えられていました。乗り物の階層構造をモデリングする際には、`SeaPlane` (水上飛行機) は明らかに、`Boat` (`startEngine()` メソッドと `sail()` メソッドと共に) と `Plane` (`startEngine()` メソッドと `fly()` メソッドと共に) の両方から継承する必要があります。`SeaPlane` は `Boat` としても `Plane` としても動作するからです。

いずれにせよ、これが C++ の全盛期の考え方でした。さて、「早送り」して Java 言語の時代まで来ると、多重継承には private 継承とまったく同じように欠陥があると見なされています。しかしどの Java 開発者も語るように、`SeaPlane` は `Floatable` インターフェースと `Flyable` インターフェースから (そしておそらく、インターフェースまたは基本クラスである `EnginePowered` からも) 継承する必要があるのです。インターフェースから継承するということは、仮想多重継承という恐怖に遭遇することなく、そのクラスに必要なすべてのメソッドを実装できるということです (ここでは、`SeaPlane` の `startEngine()` メソッドが呼び出された場合に、どちらの基本クラスの `startEngine()` を呼び出すのかという問題を解決しようとしています)。

残念ながら、private 継承と多重継承を放棄したことで、コードの再利用の面で大きな代償を払う羽目になりました。Java を開発した人達は仮想多重継承がなくなったことを喜ぶかもしれませんが、その代償としてプログラマーが行う作業は骨の折れる (そして間違いやすい) 作業になりがちなのです。

再利用可能な振る舞いを再検証する

> 出来事 (event) は大まかに言って、おそらく実際には起きなかった出来事と、どうでもよい出来事に分けることができる。
> — William Ralph Inge

JavaBeans 仕様は Java プラットフォームの土台であり、Java のエコシステムが大きく依存する POJO が生まれる元となりました。私達は皆、Java コードのプロパティーは `get()` と `set()` のペアによって操作されるという概念を理解しています (リスト 1)。

リスト 1. Person POJO

```
//This is Java
public class Person
{
    private String lastName;
    private String firstName;
    private int age;

    public Person(String fn, String ln, int a)
    {
        lastName = ln; firstName = fn; age = a;
    }

    public String getFirstName() { return firstName; }
    public void setFirstName(String v) { firstName = v; }
    public String getLastName() { return lastName; }
    public void setLastName(String v) { lastName = v; }
    public int getAge() { return age; }
    public void setAge(int v) { age = v; }
}
```

これは非常に単純に見え、あまり苦労せずに行えそうです。しかし、通知をサポートしなければならない場合 (例えばサードパーティーが POJO を使って登録し、プロパティーが変更されたらコールバックを受け取れるようにしたい場合) はどうすればよいのでしょう。JavaBeans の仕様によれば、そうした場合には `PropertyChangeListener` インターフェースと、このインターフェースの 1 つのメソッド、`propertyChange()` を実装する必要があります。任意の POJO の `PropertyChangeListener` がプロパティーへの変更に対して「賛否を示す」ことができるようにしたい場合には、その POJO は `VetoableChangeListener` インターフェースも実装する必要があり、そのためには `vetoableChange()` メソッドを実装する必要があります。

少なくとも、それが本来の姿なのです。

実際には、プロパティーの変更通知を受信するものが `PropertyChangeListener` インターフェースを実装する必要があり、また送信側 (この場合は `Person` クラス) が、このインターフェースのインスタンスを引数に取る public メソッドと、リスナーがリッスンしようとするプロパティーの名前を提供する必要があります。その結果、`Person` はずっと複雑になります (リスト 2)。

リスト 2. Person POJO の第 2 版

```
//This is Java
public class Person
{
    // rest as before, except that inside each setter we have to do something
    // like:
    // public setFoo(T newValue)
    // {
    //     T oldValue = foo;
    //     foo = newValue;
    //     pcs.firePropertyChange("foo", oldValue, newValue);
    // }

    public void addPropertyChangeListener(PropertyChangeListener pcl)
    {
        // keep a reference to pcl
    }
    public void removePropertyChangeListener(PropertyChangeListener pcl)
    {
        // find the reference to pcl and remove it
```

```
    }
}
```

プロパティー変更のリスナーへの参照を保持するということは、Person POJO が、すべての参照を含むために何らかのコレクション・クラス (例えば `ArrayList`) を保持する必要があるということです。すると、この POJO をインスタンス化する必要があり、挿入削除できる必要があり、しかもそうしたアクションはアトミックではないため、この POJO には同期に対する適切な保護も含まれている必要があります。

最後に、あるプロパティーが変更された場合には、そのプロパティーのリスナーすべてに通知する必要があります (通常は `PropertyChangeListeners` のコレクションの各リスナーに対して繰り返し `propertyChange()` を呼び出すことによって行います)。そしてこのプロセスには、(`PropertyChangeEvent` クラスと JavaBeans 仕様で要求されているように) そのプロパティーと古い値、そして新しい値を記述する新しい `PropertyChangeEvent` を渡す動作が含まれている必要があります。

これまでに作成された POJO のうち、リスナー通知をサポートしているものが非常に少ないことは驚くにあたりません。それをサポートするためには膨大な作業が必要であり、しかも作成されるすべての JavaBean/POJO に対して、手動で作業を繰り返す必要があるのです。

大量の作業に対する回避策

興味深いことに、もし C++ の private 継承サポートが Java 言語に引き継がれていたとしたら、今それを利用することで、JavaBeans 仕様による難問の一部を解決することができます。そうすれば、基本クラスは POJO の基本的な `add()` メソッドと `remove()` メソッド、コレクション・クラス、そして「`firePropertyChanged()`」メソッドを提供することができ、プロパティーの変更をリスナーに通知することができるはずです。

Java クラスでも同じことをできなくはありませんが、Java には private 継承がないため、`Person` クラスは基本 `Bean` クラスから継承する必要があり、従って `Bean` にアップキャストできる必要があります。そうすると `Person` は他のどのクラスからも継承できなくなります。後者の問題は多重継承で解決できるかもしれませんが、今度は仮想継承の問題に戻ることにもなります。それは絶対に避けなければなりません。

この問題に対する Java 言語でのソリューションは、よく知られているイディオムであるサポート・クラス (この場合は `PropertyChangeSupport`) です。つまり POJO の中にある、サポート・クラスの 1 つをインスタンス化し、必要な public メソッドをその POJO 自体に置き、各 public メソッドがそのサポート・クラスを呼び出して面倒な作業をさせるのです。下記は `PropertyChangeSupport` を使うように Person POJO を更新したものです。

リスト 3. Person POJO の第 3 版

```
//This is Java
import java.beans.*;

public class Person
{
    private String lastName;
    private String firstName;
    private int age;
```

```
    private PropertyChangeSupport propChgSupport =
        new PropertyChangeSupport(this);

    public Person(String fn, String ln, int a)
    {
        lastName = ln; firstName = fn; age = a;
    }

    public String getFirstName() { return firstName; }
    public void setFirstName(String newValue)
    {
        String old = firstName;
        firstName = newValue;
        propChgSupport.firePropertyChange("firstName", old, newValue);
    }

    public String getLastName() { return lastName; }
    public void setLastName(String newValue)
    {
        String old = lastName;
        lastName = newValue;
        propChgSupport.firePropertyChange("lastName", old, newValue);
    }

    public int getAge() { return age; }
    public void setAge(int newValue)
    {
        int old = age;
        age = newValue;
        propChgSupport.firePropertyChange("age", old, newValue);
    }

    public void addPropertyChangeListener(PropertyChangeListener pcl)
    {
        propChgSupport.addPropertyChangeListener(pcl);
    }
    public void removePropertyChangeListener(PropertyChangeListener pcl)
    {
        propChgSupport.removePropertyChangeListener(pcl);
    }
}
```

皆さんはどう思われるかわかりませんが、私はこうした複雑なコードを見ると、再度アセンブリー言語を使おうかと思ってしまうほどです。しかも悪いことに、作成されるすべての POJO の中で、これとまったく同じコード・シーケンスを繰り返す必要があるのです。リスト 3 の作業の半分は POJO そのものの中にあり、そのため再利用することができません (もちろん、伝統的な「カット・アンド・ペースト」プログラミングによる場合は別です)。

では、もっと適切な回避策として Scala は何を提供するのかを調べてみましょう。

Scala での trait、そして振る舞いの再利用

> すべての人は自分自身の性格のはっきりした特徴 (trait) をじっくり見つめる義務がある。また、そうした個性を適切に統制する必要があり、そして他の誰かの個性の方が自分にとってふさわしいのでは、などと考え迷うべきではない。
> — Cicero

Scala では、インターフェースとクラスの中間にある、trait と呼ばれる新しい構成体を定義することができます。trait が風変わりな点は、クラスは必要に応じて trait を (インターフェースと同じように) いくつでも組み入れることができ、しかも trait は (クラスと同じように) 振る舞いを含む

ことができることです。また、クラスとインターフェースの両方と同じように、trait は新しいメソッドを導入することができます。しかしクラスともインターフェースとも異なり、trait が実際にクラスの一部として組み込まれるまで trait の振る舞いの定義はチェックされません。あるいは別の言い方をすれば、trait を使用するクラス定義の中に組み込まれるまで適切さをチェックされないメソッドを定義することができます。

trait は複雑に思えるかもしれませんが、実際の使い方を見れば容易に理解することができます。まず、Person POJO を Scala で再定義したものが次のリストです。

リスト 4. Scala による Person POJO

```
//This is Scala
class Person(var firstName:String, var lastName:String, var age:Int)
{
}
```

また、単純に `scala.reflect.BeanProperty` アノテーションをクラス・パラメーター、`firstName`、`lastName`、`age` に対して使うだけで、Scala の POJO が Java POJO ベースの環境で想定されている `get()/set()` メソッドを持つことが保証されます。ここではとりあえず簡単のため、`get()/set()` メソッドは別にしておきます。

`Person` クラスが `PropertyChangeListeners` を受け付けられるようにしたい場合には、リスト 5 の方法で行うことができます。

リスト 5. Scala による、リスナーを持つ Person POJO

```
//This is Scala
object PCL
    extends java.beans.PropertyChangeListener
{
    override def propertyChange(pce:java.beans.PropertyChangeEvent):Unit =
    {
        System.out.println("Bean changed its " + pce.getPropertyName() +
            " from " + pce.getOldValue() +
            " to " + pce.getNewValue())
    }
}
object App
{
    def main(args:Array[String]):Unit =
    {
        val p = new Person("Jennifer", "Aloi", 28)

        p.addPropertyChangeListener(PCL)

        p.setFirstName("Jenni")
        p.setAge(29)

        System.out.println(p)
    }
}
```

リスト 5 では `object` を使うことで静的メソッドをリスナーとして登録できていることに注意してください。Java コードでは、明示的に `Singleton` クラスを作成してインスタンス化しない限り、こうすることはできません。このことからも、Scala が Java 開発に関する歴史的な苦労から学んだ理論が実証されています。

Person に対する次のステップは、addPropertyChangeListener() メソッドを提供し、さらにプロパティーが変更された時に各リスナーで propertyChange() メソッドの呼び出しを起動することです。Scala では、これを単に trait を定義して使用するだけの再利用可能な方法で容易に行うことができます (リスト 6)。ここではこの trait を BoundPropertyBean と呼ぶことにします。JavaBeans の仕様では「通知を受ける」プロパティーは正式にはバウンド・プロパティーと呼ばれるからです。

リスト 6. これが振る舞いの再利用です

```
//This is Scala
trait BoundPropertyBean
{
    import java.beans._

    val pcs = new PropertyChangeSupport(this)

    def addPropertyChangeListener(pcl : PropertyChangeListener) =
        pcs.addPropertyChangeListener(pcl)

    def removePropertyChangeListener(pcl : PropertyChangeListener) =
        pcs.removePropertyChangeListener(pcl)

    def firePropertyChange(name : String, oldVal : _, newVal : _) : Unit =
        pcs.firePropertyChange(new PropertyChangeEvent(this, name, oldVal, newVal))
}
```

この場合も相変わらず java.beans パッケージの PropertyChangeSupport クラスを使用しています。その理由は必要な実装の詳細の約 60% をこのクラスが提供してくれるからですが、それだけではなく、このクラスを使用することで、このクラスを直接使用する JavaBeans/POJO と同じ振る舞いを得られるからです。この「サポート」クラスにどのような機能を追加しても、追加された機能は trait によって伝達されるのです。唯一の違いは、この方法では Person POJO が PropertyChangeSupport を直接使用する必要がないことです (リスト 7)

リスト 7. Scala による Person POJO 第 2 版

```
//This is Scala
class Person(var firstName:String, var lastName:String, var age:Int)
    extends Object
    with BoundPropertyBean
{
    override def toString = "[Person: firstName=" + firstName +
        " lastName=" + lastName + " age=" + age + "]"
}
```

コンパイルした後で Person の定義を少し見てみると、Person には public メソッド、addPropertyChangeListener()、removePropertyChangeListener()、firePropertyChange() があることがわかります。これは Java 版の Person の場合とまったく同じです。Scala 版の Person は実質的に、クラス宣言の中の with 節という 1 行のコードの追加だけで、こうした新しいメソッドを得ています (この with 節は Person クラスが BoundPropertyBean という trait から継承していることを示しています)。

残念ながら、まだ終わりではありません。これで Person クラスはリスナーの受け付け、削除、そしてリスナーへの通知をサポートできるようになりましたが、Scala が firstName メンバーに対して生成するデフォルトのメソッドはリスナーを利用していません。また同じく残念なことに、

この記事の執筆時点では、Scala には `PropertyChangeSupport` インスタンスを使用する get/set メソッドを魔法のように自動的に生成する便利なアノテーションがありません。そのため get/set メソッドを自分で作成する必要があります (リスト 8)。

リスト 8. Scala による Person POJO 第 3 版

```
//This is Scala
class Person(var firstName:String, var lastName:String, var age:Int)
    extends Object
    with BoundPropertyBean
{
    def setFirstName(newvalue:String) =
    {
        val oldvalue = firstName
        firstName = newvalue
        firePropertyChange("firstName", oldvalue, newvalue)
    }

    def setLastName(newvalue:String) =
    {
        val oldvalue = lastName
        lastName = newvalue
        firePropertyChange("lastName", oldvalue, newvalue)
    }

    def setAge(newvalue:Int) =
    {
        val oldvalue = age
        age = newvalue
        firePropertyChange("age", oldvalue, newvalue)
    }

    override def toString = "[Person: firstName=" + firstName +
        " lastName=" + lastName + " age=" + age + "]"
}
```

便利な trait

trait は決して関数型の概念ではありません。むしろオブジェクト・プログラミングに関する 10 年間にわたる反省の結果なのです。実際、簡単な Scala プログラムでは気付かずに次のような trait を使ってしまうかもしれません。

リスト 9. main() よ、立ち去れ！

```
//This is Scala
object App extends Application
{
    val p = new Person("Jennifer", "Aloi", 29)

    p.addPropertyChangeListener(PCL)

    p.setFirstName("Jenni")
    p.setAge(30)

    System.out.println(p)
}
```

この `Application` の trait は、これまでずっと手動で定義されてきたものと同じ `main()` メソッドを定義しています。実際、この trait には、もう 1 つ便利なもの、タイマーが含まれています。タイマーは、`Application` を実装するコードに `scala.time` というシステム・プロパティーが渡されると、このアプリケーションの実行時間を計測します (リスト 10)。

リスト 10. 時間計測がすべてです

```
$ scala -Dscala.time App
Bean changed its firstName from Jennifer to Jenni
Bean changed its age from 29 to 30
[Person: firstName=Jenni lastName=Aloi age=30]
[total 15ms]
```

JVM での trait

> 非常に高度な技術は、どれも魔法と区別できない。
> — Arthur C Clarke

ここまで来ると、この一見魔法のようなメソッド兼インターフェースという構成体 (つまり trait) が、どのように JVM にマッピングされるのかを尋ねても不適切ではないでしょう。リスト 11 は、この魔法のカーテンの背後で何が起きているのかを、おなじみの `javap` が見せてくれます。

リスト 11. Person の中身を見る

```
$ javap -classpath C:\Prg\scala-2.7.0-final\lib\scala-library.jar;classes Person
Compiled from "Person.scala"
public class Person extends java.lang.Object implements BoundPropertyBean,scala.
ScalaObject{
    public Person(java.lang.String, java.lang.String, int);
    public java.lang.String toString();
    public void setAge(int);
    public void setLastName(java.lang.String);
    public void setFirstName(java.lang.String);
    public void age_$eq(int);
    public int age();
    public void lastName_$eq(java.lang.String);
    public java.lang.String lastName();
    public void firstName_$eq(java.lang.String);
    public java.lang.String firstName();
    public int $tag();
    public void firePropertyChange(java.lang.String, java.lang.Object, java.lang
.Object);
    public void removePropertyChangeListener(java.beans.PropertyChangeListener);

    public void addPropertyChangeListener(java.beans.PropertyChangeListener);
    public final void pcs_$eq(java.beans.PropertyChangeSupport);
    public final java.beans.PropertyChangeSupport pcs();
}
```

`Person` に対するクラス宣言に注目してください。この POJO は `BoundPropertyBean` というインターフェースを実装しており、これによって trait が (インターフェースとして) JVM そのものにマッピングされています。では trait のメソッドの実装に関してはどうなのでしょう。ここで思い出して欲しいのですが、コンパイラーは最終的な結果が Scala 言語の意味体系に従う限り、どんな種類の細工もできるのです。この場合コンパイラーは、trait の中で定義されているメソッドの実装とフィールドの宣言を、この trait を実装する `Person` クラスの中に入れています。`-private` を付けて `javap` を実行すると、これが非常に明白にわかります (もし `javap` 出力の最後の 2 行 (trait の中で定義される pcs の値を参照しています) を見ても明白にわからなければの話です)。

リスト 12. Person の中身を見る、第 2 版

```
$ javap -private -classpath C:\Prg\scala-2.7.0-final\lib\scala-library.jar;classes Person
Compiled from "Person.scala"
public class Person extends java.lang.Object implements BoundPropertyBean,scala.
ScalaObject{
    private final java.beans.PropertyChangeSupport pcs;
```

```
    private int age;
    private java.lang.String lastName;
    private java.lang.String firstName;
    public Person(java.lang.String, java.lang.String, int);
    public java.lang.String toString();
    public void setAge(int);
    public void setLastName(java.lang.String);
    public void setFirstName(java.lang.String);
    public void age_$eq(int);
    public int age();
    public void lastName_$eq(java.lang.String);
    public java.lang.String lastName();
    public void firstName_$eq(java.lang.String);
    public java.lang.String firstName();
    public int $tag();
    public void firePropertyChange(java.lang.String, java.lang.Object, java.lang.Object);
    public void removePropertyChangeListener(java.beans.PropertyChangeListener);

    public void addPropertyChangeListener(java.beans.PropertyChangeListener);
    public final void pcs_$eq(java.beans.PropertyChangeSupport);
    public final java.beans.PropertyChangeSupport pcs();
}
```

実際、この説明は、チェック用に使われるまで trait のメソッドの実行を遅延する方法についての答えにもなっています。trait のメソッドは、(あるクラスがその trait を実装するまで) 実際にはどのクラスの「部分」でもないため、コンパイラーは、これらのメソッドのロジックの一部の側面をチェックしないままにしておくことができます。これは非常に便利です。これによって trait は、その trait が実装するクラスの基本クラスが実際にはどんなものかを知らなくても `super()` を呼び出すことができるからです。

trait に関する注意点

ここでは `BoundPropertyBean` の中で、`PropertyChangeSupport` インスタンスを作成する際に trait の機能を使っています。このコンストラクターはプロパティーを通知する対象となる `Bean` を必要とし、また先ほど定義した trait の中で、私は「`this`」を渡しました。`Person` に対して実装されるまで trait は実際には定義されないので、「`this`」は `Person` インスタンスを参照することになり、`BoundPropertyBean` trait そのものを参照するわけではありません。trait の持つ、この特定の側面、つまり定義の解決を遅らせるという側面は、小さなことですが、この種の「遅延バインディング」に対して非常に強力です。

この `Application` の trait の場合には、この魔法は 2 ヵ所で起こります。`Application` の trait の `main()` メソッドは Java アプリケーションに対する汎用の入力ポイントを提供し、また実行時間を追跡する必要があるかどうか判断するために `-Dscala.time` システム・プロパティーもチェックします。しかし `Application` は trait であるため、このメソッドは実際にサブクラス (`App`) 上に「現れ」ます。このメソッドを実行するためには `App` シングルトンを作成する必要があります。これは `App` のインスタンスを作成するということであり、またそのクラスの本体を「いじる」ということであり、実質的にそれによってアプリケーションが実行されます。それが完了した後で初めて trait の `main()` が呼び出され、実行にかかった時間が表示されます。

これはあまりスマートではありませんが、動作はします。ただし、アプリケーションは `main()` に渡されるどのコマンドライン・パラメーターにもアクセスできない、という点に注意する必要があります。またこの方法は trait の振る舞いをその trait が実装するクラスにまで「遅延」させる方法の説明でもあります。

trait とコレクション

> あなたが溶液の一部でないなら、あなたは沈殿物である。
> — Henry J Tillman

trait の強力さが特に発揮されるのは、trait によって具体的な振る舞いと抽象的な宣言とを組み合わせ、実装者に便利なものを提供する場合です。例えば、Java の古典的なコレクション・インターフェース兼クラスである、`List` と `ArrayList` を考えてみてください。`List` インターフェースでは、`List` に内容を挿入した順序と同じ順序でコレクションの内容をトラバースできることが保証されています。あるいはもっと正式な言い方をすれば、「場所のセマンティクスが尊重されます」。

`ArrayList` は特別なタイプの `List` であり、割り当てられた配列にリストの内容を保存しますが、一方 `LinkedList` はリンク・リストの実装を使います。ArrayLists はリストの内容にランダムにアクセスする場合に適しており、`LinkedList` はリストの最後以外の任意の場所での挿入や削除に適しています。いずれにせよ、この 2 つのクラスの振る舞いの大部分は同じであり、その結果どちらも、共通の基本クラスである `AbstractList` を継承します。

もし trait が Java プログラミングでサポートされていたとしたら、この「共通の基本クラスを継承するという手段に訴える必要のない、再利用可能な振る舞い」といったような面倒な問題に対する、はるかに優れた構成体になっていたことでしょう。trait は C++ の「private 継承」機構のように動作することができます。そのため、新しい `List` サブタイプは直接 `List` を実装するべきなのか (そして、もしかすると `RandomAccess` インターフェースの実装も忘れてよいのか)、あるいは `AbstractList` 基本クラスを継承するべきなのかの判断に迷うおそれがなくなります。これは C++ で「mixin」と呼ばれたりもしていましたが、Ruby の mixin と混同してはいけません (あるいは Scala の mixin と間違えるかもしれませんが、Scala の mixin については今後の記事で説明する予定です) 。

Scala のドキュメンテーション・セットの中には古典的な例として `Ordered` trait があり、この trait は「奇妙な名前を持つメソッド」を定義することで比較機能 (従って順序付け) 機能を提供します (リスト 13)。

リスト 13. 順序に従って並べる

```
//This is Scala
trait Ordered[A] {
  def compare(that: A): Int

  def <  (that: A): Boolean = (this compare that) <  0
  def >  (that: A): Boolean = (this compare that) >  0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
  def compareTo(that: A): Int = compare(that)
}
```

ここでは `Ordered` trait (パラメーター化された型、つまり Java 5 のGenerics) が 1 つの抽象メソッド、`compare` を定義しています。`compare` はパラメーターとして A を取り、`this` が that 「よりも小」ならば 1 よりも小さい値を返し、`this` が that 「よりも大」ならば 1 よりも大きな値を返し、両者が等しければ 0 を返します。それに続いて、`compare()` メソッドに関する比較演算子 (`<` や `>` など) と、java.util.Comparable インターフェースでも使用されるおなじみの `compareTo()` メソッドを定義しています。

Scala と Java の互換性

> 1 つの画像には千個の単語の価値があり、1 つのインターフェースには千個の画像の価値がある。
> — Ben Shneiderman

実際には、「擬似的な実装と継承」は Scala における trait の最も強力な使い方でもなく、最も一般的な使い方でもありません。むしろ Scala での trait は、Java のインターフェースに対する基本的な置き換えなのです。Scala を呼び出して使いたい Java プログラマーは、Scala を使うための機構として、tarit にも慣れる必要があります。

私がこのシリーズで今まで指摘してきたように、コンパイルされた Scala コードは必ずしも Java 言語に非常に似ているわけでありません。例えば、Scala の「奇妙な名前を持つメソッド」(「`+`」や「`\`」など) が、Java 言語の構文では直接使用できない文字でエンコードされることがよくあることを思い出してください (「`$`」は大いに気になるものの 1 つです)。そのため、「Java から呼び出せる」インターフェースを作成することで Scala コードへの呼び出しを単純化することができます。

ここに挙げる例は少し不自然であり、ここで使用されている Scala の方式では trait による間接化レイヤーを実際には必要としません (ここでは「奇妙な名前を持つメソッド」を使っていないため)。しかし少し我慢してください。ここでは考え方が重要なのです。リスト 14 では、`Student` インスタンスを生成する、(さまざまな Java オブジェクト・モデルに通常見られるような) 従来の Java スタイルのファクトリーを作ろうとしています。まず、`Student` に対する Java 互換のインターフェースが必要です。

リスト 14. 私は学生です

```
//This is Scala
trait Student
{
    def getFirstName : String;
    def getLastName : String;
    def setFirstName(fn : String) : Unit;
    def setLastName(fn : String) : Unit;

    def teach(subject : String)
}
```

これをコンパイルすると、POJI、つまり Plain Old Java Interface になります (`javap` を使って少し見てみるとわかります)。

リスト 15. これは POJI です

```
$ javap Student
Compiled from "Student.scala"
public interface Student extends scala.ScalaObject{
    public abstract void setLastName(java.lang.String);
    public abstract void setFirstName(java.lang.String);
    public abstract java.lang.String getLastName();
    public abstract java.lang.String getFirstName();
    public abstract void teach(java.lang.String);
}
```

次に、ファクトリーそのものとなるクラスが必要です。通常の Java コードでは、これはクラスに対する (「StudentFactory」といったような) 静的なメソッドです。しかし Scala には静的メソッドというものがないことを思い出してください。その代わり Scala にはオブジェクトがあり、オブジェクトはインスタンス・メソッドを持つシングルトンです。これこそ、まさにここで探し求めているものです。そこで StudentFactory オブジェクトを作成し、そこに Factory メソッドを置きます。

リスト 16. Student を作成する

```
//This is Java
object StudentFactory
{
    class StudentImpl(var first:String, var last:String, var subject:String)
        extends Student
    {
        def getFirstName : String = first
        def setFirstName(fn: String) : Unit = first = fn
        def getLastName : String = last
        def setLastName(ln: String) : Unit = last = ln

        def teach(subject : String) =
            System.out.println("I know " + subject)
    }

    def getStudent(firstName: String, lastName: String) : Student =
    {
        new StudentImpl(firstName, lastName, "Scala")
    }
}
```

ネストされたクラス、StudentImpl は Student の trait の実装であり、従ってこの実装はこの trait に必要な get()/set() メソッドのペアを提供します。思い出して欲しいのですが、trait は振る舞いを持てるものの、trait は JVM に対してインターフェースとしてモデル化されているため、trait をインスタンス化しようとすると Student は抽象化されているというエラーになってしまいます。

この些細な例から得られる重要な成果として、当然のことながら、Scala によって作成されたこのような新しいオブジェクトを利用した Java アプリケーションを作成できるのです。

リスト 17. 学生 Neo

```
//This is Java
public class App
{
    public static void main(String[] args)
    {
        Student s = StudentFactory.getStudent("Neo", "Anderson");
        s.teach("Kung fu");
    }
}
```

これを実行すると、「I know Kung fu」(カンフーを知っています) が表示されるはずです (これまでの設定が単に映画を参照するだけにしては長すぎることは承知しています。

まとめ

> 人は考えることを嫌います。考えると、結論を出さなければなりません。結論は必ずしも楽しいものではありません。
> — Helen Keller

trait は Scala でのカテゴリー分けと定義のための強力な機構であり、クライアントが使うための (従来の Java インターフェース式の) インターフェースを定義する機構であると同時に、trait の中で定義される他の振る舞いを元に振る舞いを継承するための機構でもあります。おそらく私達に必要なものは、trait と trait が実装するクラスとの間の関係を記述する、継承のための新しい表現、IN-TERMS-OF (・・の観点では) なのかもしれません。

この記事で説明した使い方以外にも、さまざまな方法で trait を使うことができます。しかしこのシリーズの目標の 1 つは、皆さん自身が自習できるように Scala 言語に関する十分な情報を提供することです。Scala の実装をダウンロードして実験し、現在の Java システムのどの部分に Scala を活用できるかを調べてみてください。そして毎度のことですが、Scala が便利であるとわかったり、この記事に関するコメントがあったり、あるいはコードや文章の中にバグを見つけたら、メールで私に連絡してください。

では関数型のファンの皆さん、次回をお楽しみに。

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。