多忙な Java 開発者のための Scala ガイド: パッケージとアクセス修飾子

Scala で public、private、そしてその中間のさまざまなスコープを指定する

Ted Neward
Principal
Neward & Associates

2008年 7月 29日

実際に作成されたコードは、他のコードから参照できるようにパッケージ化されます。「多忙な Java 開発者のための Scala ガイド」シリーズ第 7 回目の今回は、Ted Neward が、これまでの説明で大きく抜け落ちていた、Scala のパッケージとアクセス修飾子について説明します。その後で、Scala の関数型の側面についての説明を続け、今回は apply の仕組みについて簡単に説明します。

このシリーズの他の記事を見る

私は最近、読者からの指摘によって、このシリーズを進める中で Scala 言語の重要な側面である、パッケージとアクセス修飾子の機能についての説明を忘れていたことに気付きました。そこで今回はこれらの機能について説明し、その後で Scala 言語の関数型要素の 1 つである `apply` の仕組みを説明することにします。

パッケージ化

Java™ コードには、コード同士が競合しないようにコードをグループ分けするために package キーワードが用意されています。この `package` キーワードによって語彙の名前空間が作成され、その内部でクラスの宣言が行われます。要するに、com.tedneward.util という名前のパッケージの中に `Foo` というクラスを配置すると、正式なクラス名は `com.tedneward.util.Foo` に変更されるため、このクラスを参照するには、この正式なクラス名を使う必要があるということです。しかし Java プログラマーは即座に、自分たちはそんなことはしない、と言うでしょう。彼らはパッケージをインポートすることで正式な名前をコーディングしなくてすむようにします。確かにそうすることで正式な名前のコーディングを省いていますが、それは実際のところは正式な名前でクラスを参照するという作業が単にコンパイラーとバイトコードに移っただけににすぎません。これは javap で出力した内容をちょっと見てみればわかります。

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。このとても楽しみな developerWorks のシリーズでは、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 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 で可能ならば、手間をかけて Scala を学ぶ必要はないのです。

ただし、Java 言語でのパッケージには、いくつか奇妙な性質があります。例えば、パッケージ宣言は .java ファイルの先頭に置き、そのパッケージのスコープを持つクラスをそのファイル内に記述する必要があります (このため、そのパッケージにアノテーションを適用しようとすると言語上の深刻な混乱が生じます)。そして、パッケージ宣言のスコープは、そのファイル全体にわたります。これはつまり、2 つのクラスがパッケージの境界をまたがって密に結合されているという稀なケースで、ファイルを分割しなければならない場合に、2 つのクラスが密に結合されていることに気付かないという問題が起きやすくなるということです。

Scala ではパッケージ化に関して少し異なる手法をとっており、Java 言語での宣言による手法と C# でのスコープを指定する手法の組み合わせとしてパッケージを扱います。これを念頭に置いた上で、従来の Java の方法に従って通常の Java のクラスの場合と同じように `package` 宣言を .scala ファイルの先頭に記述することができます。この場合、パッケージ宣言のスコープは Java コードの場合と同じようにファイル全体に適用されます。もう 1 つの方法として、Scala のパッケージの「スコープを指定する」方法を使うこともできます。この方法では中括弧によって `package` 文のスコープを区切ります (リスト 1)。

リスト 1. パッケージ化が単純に

```
package com
{
  package tedneward
  {
    package scala
    {
      package demonstration
      {
        object App
        {
          def main(args : Array[String]) : Unit =
          {
            System.out.println("Howdy, from packaged code!")
            args.foreach((i) => System.out.println("Got " + i) )
          }
        }
      }
    }
  }
}
```

このコードは実質的に、`App` という 1 つのクラス、あるいはもっと正確に言えば `com.tedneward.scala.demonstration.App` という 1 つのクラスを宣言しています。Scala ではパッケージ名をドットで区切ることもできることに注意してください。この方法で記述するとリスト 1 はもっと簡潔になります (リスト 2)。

リスト 2. パッケージ化が単純に (簡潔バージョン)

```
package com.tedneward.scala.demonstration
{
  object App
  {
      def main(args : Array[String]) : Unit =
      {
        System.out.println("Howdy, from packaged code!")
        args.foreach((i) => System.out.println("Got " + i) )
      }
  }
}
```

どちらをコンパイルしてもコードの構成はまったく同じになるため、どちらでも適切と思われるスタイルを使うことができます。 (scalac は javac と同様、パッケージ宣言をしたサブディレクトリーの中に .class ファイルを生成します。)

インポート

パッケージ化したものを利用するために行う操作は、もちろんインポートです。Scala では、インポートは現在の語彙の名前空間に名前を追加するための仕組みです。このシリーズの読者は既にこれまで `import` を使った例をいくつか見てきていますが、今回は Java 開発者が驚くような `import` の使い方をいくつか紹介します。

まず第 1 に、`import` はクライアントの Scala ファイルの先頭だけではなく、このファイル内の任意の場所で使うことができるため、スコープを限定することができます。つまりリスト 3 では、`java.math.BigInteger` のインポートが有効なスコープは `App` オブジェクト内部で定義されるメソッドに完全に限定され、それ以外のところでは有効ではありません。`mathfun` 内の `App` 外部のクラスあるいはオブジェクトで `java.math.BigInteger` を使おうとする場合には、`App` でこのクラスをインポートしているのと同様に、その当該のクラスあるいはオブジェクトで、この `java.math.BigInteger` クラスをインポートする必要があります。あるいは、`mathfun` の中のすべてのクラスで `java.math.BigInteger` を使おうとする場合には、`App` の定義の外のパッケージ・レベルでインポートを行う必要があり、そうすることによって、このパッケージのスコープ内のすべてのクラスに `BigInteger` がインポートされます。

リスト 3. import のスコープを指定する

```
package com
{
  package tedneward
  {
    package scala
    {
        // ...

      package mathfun
      {
        object App
        {
          import java.math.BigInteger

          def factorial(arg : BigInteger) : BigInteger =
          {
            if (arg == BigInteger.ZERO) BigInteger.ONE
            else arg multiply (factorial (arg subtract BigInteger.ONE))
          }
```

```
          def main(args : Array[String]) : Unit =
          {
            if (args.length > 0)
              System.out.println("factorial " + args(0) +
                " = " + factorial(new BigInteger(args(0))))
            else
              System.out.println("factorial 0 = 1")
          }
        }
      }
    }
  }
}
```

しかし `import` の使い方はこれだけではありません。Scala では、最上位レベルのメンバーとネストされたメンバーを区別する理由は何もないと考えます。そのため、import を使うことによって、ネストされたメンバーだけではなく、任意のメンバーをレキシカル・スコープに入れることができます。例えばリスト 4 のように `java.math.BigInteger` の中にあるすべての名前をインポートすることによって、スコープをつけて ZERO や ONE を指定しなくても、スコープを付けない名前のみで ZERO や ONE を指定することができます。

リスト 4. (static を使わない) 静的なインポート

```
package com
{
  package tedneward
  {
    package scala
    {
        // ...

      package mathfun
      {
        object App
        {
          import java.math.BigInteger
          import BigInteger._

          def factorial(arg : BigInteger) : BigInteger =
          {
            if (arg == ZERO) ONE
            else arg multiply (factorial (arg subtract ONE))
          }

          def main(args : Array[String]) : Unit =
          {
            if (args.length > 0)
              System.out.println("factorial " + args(0) +
                " = " + factorial(new BigInteger(args(0))))
            else
              System.out.println("factorial 0 = 1")
          }
        }
      }
    }
  }
}
```

アンダーバー (Scala でのワイルドカード文字を覚えているでしょうか) を使うことで、実質的に、`BigInteger` の中のすべてのメンバーをスコープ内に入れるように、Scala のコンパイラーに指示することができます。また `BigInteger` はその前の import 文によって既にスコープ内に入って

いるため、クラス名を明示的にパッケージ名で修飾する必要はありません。その上、これらを組み合わせて 1 つの import 文にすることができます。なぜなら import 文はカンマで区切られた複数のターゲットをインポートすることができるからです (リスト 5)。

リスト 5. まとめてインポートする

```
package com
{
  package tedneward
  {
    package scala
    {
        // ...

      package mathfun
      {
        object App
        {
          import java.math.BigInteger, BigInteger._

          def factorial(arg : BigInteger) : BigInteger =
          {
            if (arg == ZERO) ONE
            else arg multiply (factorial (arg subtract ONE))
          }

          def main(args : Array[String]) : Unit =
          {
            if (args.length > 0)
              System.out.println("factorial " + args(0) +
                " = " + factorial(new BigInteger(args(0))))
            else
              System.out.println("factorial 0 = 1")
          }
        }
      }
    }
  }
}
```

こうすることによって 1 行か 2 行、節約することができます。ただし、次の 2 つを組み合わせることはできないことに注意してください。つまり最初に `BigInteger` クラス自体をインポートし、次にその最初のクラス内のさまざまなメンバーをインポートする、ということはできません。

また `import` を使って、定数ではない他のメンバーを導入することもできます。例えば、リスト 6 に示す (おそらく計算結果に問題がある) 数学のユーティリティー・ライブラリーを考えてみてください (訳注: リスト 6 からリスト 9 のタイトルにある「エンロン」は 2001年に粉飾決算が問題となった米国の大企業のエンロンのことで、著者はライブラリーの計算に問題があることをエンロンの粉飾決算にかけているようです))。

リスト 6. エンロンの決算用のコード

```
package com
{
  package tedneward
  {
    package scala
    {
        // ...

      package mathfun
```

```
      {
        object BizarroMath
        {
          def bizplus(a : Int, b : Int) = { a - b }
          def bizminus(a : Int, b : Int) = { a + b }
          def bizmultiply(a : Int, b : Int) = { a / b }
          def bizdivide(a : Int, b : Int) = { a * b }
        }
      }
    }
  }
}
```

このライブラリーをしばらく使うと、非常に面倒に思えてきます。なぜなら `BizarroMath` のメンバーの 1 つが要求されるたびに `BizarroMath` と入力しなければならないからです。しかし Scala では、`BizarroMath` の各メンバーがあたかもグローバル関数であるかのように、最上位レベルの語彙の名前空間にそれらのメンバーをインポートすることができます (リスト 7)。

リスト 7. エンロンの経費を計算する

```
package com
{
  package tedneward
  {
    package scala
    {
      package demonstration
      {
        object App2
        {
          def main(args : Array[String]) : Unit =
          {
            import com.tedneward.scala.mathfun.BizarroMath._

            System.out.println("2 + 2 = " + bizplus(2,2))
          }
        }
      }
    }
  }
}
```

他にも興味深い構成体があり、それらを利用すると `2 bizplus 2` のようにもっと自然に書けるのですが、それについての説明は別の機会に譲ります。(興味のある読者のために、Odersky と Spoon、Venners の共著による『Programming in Scala』では、非常に乱用される可能性のある機能である Scala の `implicit` 構成体について解説しています。)

アクセス制限

パッケージ化 (そしてインポート) は Scala でのカプセル化とパッケージ化の機能の一部ですが、Java コードの場合と同様、カプセル化とパッケージ化の機能の中でも主要な機能として、特定のメンバーへのアクセスを選択的な方法で制限する機能があります。つまり Scala では、一部のメンバーを「public」、「private」、あるいはその中間の任意のアクセス制限の指定をすることができます。

Java 言語のアクセス･レベルには、public、private、protected、そしてパッケージ・レベル (苛立たしいことにキーワードがありません) の 4 つがありますが、Scala でのアクセス・レベルは以下のようになっています。

- (ある方法では) パッケージ・レベルの修飾子を使う必要がありません。
- デフォルトは public です。
- 「このスコープ内でのみアクセス可能」という意味で private を指定します。

そして protected は、Java コードでの protected とは明らかに違う使い方をします。Java の protected メンバーには、サブクラスからも、そのメンバーが定義されているパッケージからもアクセス可能ですが、Scala ではサブクラスからのアクセスしか許可していません。これはつまり、Scala での protected は Java の protected よりも (間違いなく直感的だとは言え) 制限が厳しいということです。

しかし Scala が Java コードと明確に異なる点は、Scala でのアクセス修飾子はさらにパッケージ名を使って修飾できるという点で、これによってメンバーがアクセス可能な範囲のアクセス・レベルを示すことができます。例えば `BizarroMath` パッケージのメンバーが同じパッケージの他のメンバーにアクセスすることを許可する (ただしサブクラスへのアクセスは許可しない) 場合、リスト 8 のコードを使うことができます。

リスト 8. エンロンの決算のコード

```
package com
{
  package tedneward
  {
    package scala
    {
        // ...

      package mathfun
      {
        object BizarroMath
        {
          def bizplus(a : Int, b : Int) = { a - b }
          def bizminus(a : Int, b : Int) = { a + b }
          def bizmultiply(a : Int, b : Int) = { a / b }
          def bizdivide(a : Int, b : Int) = { a * b }

              private[mathfun] def bizexp(a : Int, b: Int) = 0
        }
      }
    }
  }
}
```

この場合の `private[mathfun]` という表現に注意してください。このアクセス修飾子は要するに、このメンバーは `mathfun` というパッケージの範囲までが private であると言っています。これはつまり、`mathfun` パッケージのすべてのメンバーは `bizexp` にアクセスすることはできますが、サブクラスや `mathfun` パッケージ外部のメンバーは `bizexp` ににアクセスできないということです。

これがどのように強力な意味を持っているかというと、private 宣言または protected 宣言で指定する宣言の対象範囲には、任意のパッケージを指定することが可能であり、その範囲は上記の例で言うと `com` にまで及ぼすことができるということです (あるいは、ルート名前空間のエイリアス

である `_root_` にまでも及ぼすことすらできるので、`private[_root_]` を実質的に public と同じにすることができるのです)。これによって、Java 言語よりもはるかに柔軟にアクセス可能範囲の指定をすることができます。

さらに Scala には、アクセス可能範囲を指定する方法がもう 1 つあります。それは、`private[this]` と指定することでオブジェクトを private の範囲に指定する方法です。このように指定すると、対象のメンバーはこの同じオブジェクトから呼び出されているメンバーからしか見えず、別のオブジェクトからは (たとえそのオブジェクトが同じタイプのオブジェクトであったとしても) 見えません。(これによって、面接で Java プログラミングに関する質問をするときぐらいにしか使われなかった、Java におけるアクセス可能範囲の仕様が抱える小さな欠陥を補うことができます。)

注意する点として、アクセス修飾子はいずれかの段階で JVM へとマッピングする必要があるため、コンパイルや通常の Java コードからの呼び出しが行われると、定義の詳細の一部が失われてしまいます。例えば上記の (`private[mathfun]` と宣言されたメンバー `bizexp` を持つ) BizarroMath の例からは、リスト 9 に示すクラス定義が生成されます (これは javap を実行した結果を示しています)。

リスト 9. エンロンの会計ライブラリーの JVM

```
Compiled from "packaging.scala"
public final class com.tedneward.scala.mathfun.BizarroMath
   extends java.lang.Object
{
    public static final int $tag();
    public static final int bizexp(int, int);
    public static final int bizdivide(int, int);
    public static final int bizmultiply(int, int);
    public static final int bizminus(int, int);
    public static final int bizplus(int, int);
}
```

コンパイルされた `BizarroMath` クラスの 2 行目から明らかなように、`bizexp()` メソッドには JVM レベルの `public` というアクセス修飾子が与えられています。これはつまり、Scala のコンパイラーでアクセス可能範囲のチェックを行うと、`private[mathfun]` による微妙な違いが失われているということです。そのため、Java コードから使われることを意図した Scala コードでは、私であれば従来からの private と public による定義を使うようにするでしょう。(protected でさえ、JVM レベルの public にマッピングされてしまう場合があります。そのため、疑わしい場合は実際にコンパイルされたバイトコードに対して javap を実行し、アクセス・レベルを確認する必要があります。)

apply の仕組み

このシリーズの前回の記事 (「コレクション型」) の中で Scala での配列 (正確には `Array[T]`) を説明する際、私は「配列の i 番目の要素を取得するもの」が実際には「例の奇妙な名前を持つメソッドの 1 つ」だと説明しました。実は、その際には詳細に触れたくなかったのですが、その説明は完全には正確ではありませんでした。

そうです。私は嘘を言いました。

技術的に言うと、`Array[T]` クラスで使う小括弧は、単なる「奇妙な名前を持つメソッド」というよりももう少し複雑です。Scala では、例の特定の文字シーケンス (左括弧と右括弧のシーケンス) のために、特別な呼び方が予約されています。その理由は、このシーケンスが、ある特定の意図、つまり何かを「行う」(あるいは関数型の言い方をすれば、何かに何かを「適用する」) 意図を持って使われることが非常に多いためです。

言い換えると、Scala には、「適用」演算子「()」に相当する特別な構文 (もっと正確に言えば、特別な構文上の関係) があります。正確に言うと、Scala では、指定されたオブジェクトをメソッド呼び出しのように `()` を使って呼び出す場合、`apply()` メソッドがそのオブジェクトに対して呼び出されたと認識します。例えば、ファンクター (`functor`: 関数として動作するオブジェクト) として動作しようとするクラスは、以下のように `apply` メソッドを定義することで、関数風、あるいはメソッド風のセマンティクスを提供することができます。

リスト 10. ファンクターを使ったコードの例

```
class ApplyTest
{
  import org.junit._, Assert._

  @Test def simpleApply =
  {
    class Functor
    {
      def apply() : String =
      {
        "Doing something without arguments"
      }

      def apply(i : Int) : String =
      {
        if (i == 0)
          "Done"
        else
          "Applying... " + apply(i - 1)
      }
    }

    val f = new Functor
    assertEquals("Doing something without arguments", f() )
    assertEquals("Applying... Applying... Applying... Done", f(3))
  }
}
```

ここで読者の中には、ファンクターと匿名関数やクロージャーとではどのように違うのか、興味を持った方もいるでしょう。実は、この関係は非常に明白です。標準の Scala ライブラリーの Function1 型 (つまり 1 つのパラメーターをとる関数) の定義の中には `apply` メソッドがあります。Scala の匿名関数用に生成された Scala の匿名クラスをいくつか眺めてみると、生成されたクラスが Function1 (またはその関数がパラメーターをいくつ取るかにより、Function2 や Function3 など) の子孫であることがわかります。

これはつまり、求められている設計手法で匿名関数や名前付き関数を使うことが必ずしも適切でない場合、Scala では `functor` クラスを作成し、フィールドに初期化データを入れてこのクラスを提供し、一般的な基底クラスを使わずに `()` によってこのクラスを実行することができます (これは従来の Strategy パターンの実装の場合と同様です)。

リスト 11. ファンクターを使ったコードの例、その 2

```
class ApplyTest
{
  import org.junit._, Assert._

  // ...

  @Test def functorStrategy =
  {
    class GoodAdder
    {
      def apply(lhs : Int, rhs : Int) : Int = lhs + rhs
    }
    class BadAdder(inflateResults : Int)
    {
      def apply(lhs : Int, rhs : Int) : Int = lhs + rhs * inflateResults
    }

    val calculator = new GoodAdder
    assertEquals(4, calculator(2, 2))
    val enronAccountant = new BadAdder(50)
    assertEquals(102, enronAccountant(2, 2))
  }
}
```

適切な引数を持つ `apply` メソッドを提供するクラスであれば、どんなクラスでも引数の数と型が一致する限り、呼び出されれば正常に動作します。

まとめ

Scala でのパッケージ化やインポート、アクセス修飾子などの仕組みによって、従来の Java プログラムでは不可能だった詳細な制御やカプセル化を行うことができます。例えば、あるオブジェクトの一部のメソッドのみをインポートし、それらをグローバルなメソッドのように見せることで、グローバルなメソッドが従来より抱えていた欠点を持たないメソッドとして扱うことができます。こうした仕組みのおかげでこれらのメソッドを非常に容易に扱えるようになりますが、それが顕著に表れるのは、メソッドが、このシリーズで以前紹介した架空の `tryWithLogging` 関数 (「Don't get thrown for a loop!」を参照) のような高階関数を提供する場合です。

同様に、Scala では apply の仕組みによって、関数型の前面から実行の詳細が見えないようにすることができます。そのため、プログラマーは、呼び出している対象が実際には関数ではなく、非常に複雑になっているオブジェクトであるということすら知らなくてもよい (あるいは、気にする必要すらない) のです。この仕組みは、Scala の関数型の性質のさらなる特徴となっており、Java 言語 (あるいは C# や C++) でも確かに同様のことを行えましたが、Scala のように簡潔な構文で実現したものではありませんでした。

これで今回は終わりです。では次回をお楽しみに。

ダウンロード

内容	ファイル名	サイズ
Sample Scala code for this article	j-scala07298.zip	95KB

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。