Java.next: Groovy、Scala、Clojure の共通点、第 3 回

例外、式、値の不在についての再考

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

2013年 6月 20日

Clojure、Scala、Groovy の間にある共通点を話題にしたこの 3 回からなる記事の最終回では、これら 3 つの言語が、例外、式、そして `null` (いずれも Java 言語で問題となっている領域) をどのように扱っているのかを探ります。これらの Java.next 言語では、それぞれの言語の特性を際立たせる独自の実装によって、Java 言語が持つ欠点に対処しています。

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、それぞれの言語は Java 言語の機能を超える新たな興味深い機能をもたらしています。この連載では、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。この連載の目的は、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した上で、新しい言語の学習にどれだけの時間をかけるかの選択を十分な知識に基づいて行えるようにすることです。

前回の記事では、Java 言語を悩ませている定型的な処理や複雑さを軽減させる、Java.next 言語の革新的な方法を説明しました。今回の記事では、Java の弱点となっている例外、文と式の比較、そして `null` に関するエッジ・ケースが、Java.next 言語ではどのように改善されているのかを明らかにします。

式

Java 言語が C 言語から引き継いでいる 1 つの遺産は、文のプログラミングと式のプログラミングの区別です。Java での文の例には、`if` または `while` を使用するコード行、`void` を使用して戻り値のないメソッドを宣言するコード行などがあります。`1 + 2` のような式は、値に評価されます。

このような文と式の分裂は、最も初期のプログラミング言語で始まったものです。例えば Fortran では、ハードウェアに関する考慮事項とプログラミング言語設計に関する初期段階の理解に基づいて文と式が区別されていました。この区別は、アクション (文) と評価 (式) との違いを示す指標として多くの言語で引き継がれましたが、言語の設計者たちはそれぞれ独自に、言語を式だけ

で構成して、式の結果が興味の対象でなければ、それを無視できることに気付き始めました。現在、ほぼすべての関数型言語では文と式との区別を完全に排除し、式だけを使用しています。

Groovy の `if` と `?:`

Java.next 言語では、従来の命令型言語 (Groovy) と関数型言語 (Clojure と Scala) との分裂が、式への進化を示しています。Groovy には今でも Java 構文に基づく文があるとは言え、さらに多くの式が追加されています。Scala と Clojure については、完全に式だけが使用されています。

文と式の両方を組み込むと、言語の構文が煩雑になります。例えば、Groovy が Java から継承している `if` 文を考えてみましょう。リスト 1 で比較しているように、Groovy には判定を目的とした `if` 文と、三項演算子 `?:` という 2 つの形があります。

リスト 1. Groovy が Java から継承した `if` 文の 2 つの形

```
def x = 5
def y = 0
if (x % 2 == 0)
  y = x * 2
else
  y = x - 1
println y   // 4

y = x % 2 == 0 ? (x *= 2) : (x -= 1)
println y   // 4
```

リスト 1 の `if` 文には、副次作用として `x` の値を設定する必要があります。それは、`if` 文には戻り値がないためです。判定と代入を同時に行う場合には、リスト 1 の 2 番目の代入に示されているように、三項演算子の代入を使用する必要があります。

Scala の式ベースの `if`

Scala では三項演算子を使用する必要はありません。それは、`if` 式で両方のケースを処理できるようになっているためです。リスト 2 に示されているように、if 式は、Java コードでの `if` 文と同じように使用することも (つまり、戻り値を無視します)、代入式で使用することもできます。

リスト 2. Scala の式ベースの `if`

```
val x = 5
val y = if (x % 2 == 0)
          x * 2
 else
   x - 1
println(y)
```

Scala では、他の 2 つの Java.next 言語と同じく、メソッドに明示的な `return` 文は必要ありません。この文が存在しなければ、メソッドの最後の行が戻り値になります。この点が、これらの言語のメソッドは式をベースにするという特質を強調しています。

Java や Groovy のコードで値を操作して設定する場合と同じように、各応答をコード・ブロックとしてカプセル化して (リスト 3 を参照)、必要なあらゆる副次作用をそこに含めることができます。

リスト 3. Scala の `if` + 副次作用

```
val z = if (x % 2 == 0) {
              println("divisible by 2")
       x * 2
     } else {
              println("not divisible by 2; odd")
       x - 1
     }
println(z)
```

リスト 3 では、新しく計算された値を返すだけでなく、各ケースのステータス・メッセージを出力します。ブロック内でのコード行の順番は重要です。ブロックの最後の行が、その条件に対する戻り値を表すためです。したがって、式ベースの `if` を使用して評価と副次作用の両方を含める場合には、特に注意を払う必要があります。

Clojure の式と副次作用

Clojure も同じく式だけで構成されますが、副次作用のコードを評価から区別するという点に関してはさらに一歩先を行っています。前の 2 つの例は、Clojure では `let` ブロックで表現されます (リスト 4 を参照)。このブロックを使用することで、ローカルにスコープを設定した変数を定義することができます。

リスト 4. Clojure の式ベースの `if`

```
(let [x 5
      y (if (= 0 (rem x 2)) (* x 2) (- x 1))]
  (println y))
```

リスト 4 では、`x` に値 `5` を代入した後、`if` を使用して 2 つの条件を計算する式を作成しています。具体的には、`(rem x 2)` が余りを求める関数 (Java の `%` 演算子と同様) を呼び出し、結果をゼロと比較して、2 で割ったときに余りがゼロになるかどうかをチェックします。Clojure の `if` 式では、最初の引数が条件で、2 番目の引数が真の場合の分岐、そして 3 番目の引数がそれ以外の場合のオプションの分岐です。`if` 式の結果が `y` に代入された後、出力されます。

Clojure でも条件ごとに (副次効果を組み込める) ブロックを使用できるようになっていますが、それには `(do ...)` のようなラッパーが必要です。ラッパーは、最後の行をブロックの戻り値として使用して、ブロック内の各式を評価します。リスト 5 に、条件と副次作用を評価する方法を示します。

リスト 5. Clojure での明示的な副次作用

```
(let [x 5
      a (if (= 0 (rem x 2))
          (do
            (println "divisible by 2")
            (* x 2))
          (do
            (println "not divisible by 2; odd")
            (- x 1)))]
  (println a))
```

リスト 5 では、`a` に `if` 式の戻り値を代入します。それぞれの条件ごとに `(do ...)` ラッパーを作成することで、任意の数の文を使用できます。リスト 3 の Scala の例と同じように、ブロックの最後の行は、`(do...)` ブロックの戻り値です。`(do...)` ブロックをこの方法で使用することはよくあ

るため、Clojure の構成体 (`(let [])` など) の多くには、あらかじめ暗黙的 `(do ...)` ブロックが組み込まれています。そのため多くの場合、このブロックを組み込む必要はありません。

Java/Groovy コードと Scala/Clojure コードとで対照的な式の扱い方は、文と式の無用な二分化を排除しようというプログラミング言語の一般的傾向を示しています。

例外

私にとって、Java プログラミングで「その当時は名案だと思えた」一番の機能が、チェック例外と、例外に対する認識をブロードキャスト (および強制) できる機能です。けれども実際に使ってみると、コンテキストから切り離された不必要な例外処理 (および誤った処理) を強いるチェック例外は、悪夢と化します。

すべての Java.next 言語は共通して、JVM にすでに組み込まれている例外メカニズムと、Java 構文をベースに各言語に固有の構文に合わせて変更された構文を使用します。さらに、チェック例外を排除し、Java との相互運用中に検出されたチェック例外は `RuntimeExceptions` に変換するという点でも、これらの言語は共通しています。

式ベースの世界で機能するように Java 例外処理メカニズムを変換する上で、Scala はいくつかの興味深い振る舞いを見せています。まず、リスト 6 に示すように、式の値として例外を返すことができます。

リスト 6. 戻り値としての例外

```
val quarter =
  if (n % 4 == 0)
    n / 4
  else
    throw new RuntimeException("n must be quarterable")
```

リスト 6 では、`n` の 1/4 の値または例外のいずれかを代入します。例外は戻り値が評価される前に伝播するので、例外がトリガーされた場合の戻り値には何の意味もありません。Scala が型付き言語であることを考えると、この代入が正当であることは奇妙に思えるかもしれません。Scala の例外の型は数値型ではないことから、開発者は throw 式の戻り値を考慮することに慣れていません。Scala はこの問題を巧妙な方法で解決しています。それは、`throw` の戻り値の型として特殊な `Nothing` 型を使用するという方法です。`Any` は、(Java での Object のように) Scala の継承階層の最上位にあり、すべてのクラスがこれを継承することを意味します。逆に、`Nothing` は階層の最下位にあるため、自動的に他のすべてのクラスのサブクラスになります。したがって、リスト 6 のコードをコンパイルすると、数値が返されるか、あるいは戻り値が設定される前に例外がトリガーされるかのどちらかとなります。`Nothing` は `Int` のサブクラスであるため、コンパイラーがエラーを報告することはありません。

式ベースの世界での 2 つ目の興味深い振る舞いは、`finally` ブロックにあります。Scala の `finally` ブロックは他のブロックと同じように機能しますが、戻り値に関しては微妙に異なる振る舞いを見せます。例えば、リスト 7 のコードについて検討してみます。

リスト 7. Scala の `finally` からの戻り値

```
def testReturn(): Int =
  try {
    return 1
  } finally {
    return -1
  }
```

リスト 7 の全体としての戻り値は `-1` になります。ここでは、`finally` ブロックの戻り値が、`try` 文の本体による戻り値を上書きするためです。このような意外な結果となるのは、`finally` ブロックに明示的な `return` 文が含まれている場合のみであり、暗黙のリターンについては無視されます (リスト 8 を参照)。

リスト 8. Scala の暗黙のリターンn

```
def testImplicitReturn(): Int =
  try {
    1
  } finally {
   -1
  }
```

リスト 8 では、関数の戻り値は `1` となります。これは、`finally` ブロックに意図された用途は、式を解決することではなく、副次作用をクリーンアップする場所であることを明らかに示しています。

Clojure も同じく完全に式ベースです。`(try ...)` の戻り値は、必ず以下のいずれかになります。

- 例外が発生しなかった場合は、`try` ブロックの最後の行。
- 例外がキャッチされた場合は `catch` ブロックの最後の行。

リスト 9 に、Clojure での例外の構文を記載します。

リスト 9. Clojure の `(try...catch...finally)` ブロック

```
(try
  (do-work)
  (do-more-work)
  (catch SomeException e
    (println "Caught" (.getMessage e)) "exception message")
  (finally
    (do-clean-up)))
```

リスト 9 では、例外が発生しなかった場合の戻り値は、`(do-more-work)` から返されます。

Java.next 言語では、Java の例外メカニズムの最も優れた部分を継承し、厄介な部分は破棄しています。さらに、実装にいくつかの相違点があるとは言え、これらの言語は例外を式ベースの見方に取り込むことに成功しています。

値の不在

2009年に開催された QCon London カンファレンスでの有名なプレゼンテーションで、Tony Hoare 氏は ALGOL W (1965年に発表された実験的なオブジェクト指向言語) のために自らが考案した

「null」の概念を「10 億ドルにも相当する過ち」と称しました。この概念に感化されたプログラミング言語で、null によってそれだけの問題が発生しているというのがその理由です。Java 言語自体も null に関するエッジ・ケースに悩まされていますが、Java.next 言語では `null` に対処しています。

例えば、Java プログラミングでの一般的なイディオムでは、メソッド呼び出しをする前に、`NullPointerException` が発生するのを防ぐコードを記述します。

```
if (obj != null) {
    obj.someMethod();
}
```

Groovy はこのパターンを安全なナビゲーション演算子 `?.` にカプセル化します。この演算子は左側の項の `null` チェックを自動的に行い、`null` でない場合に限り、メソッド呼び出しを試行します。それ以外の場合は、`null` を返します。

```
obj?.someMethod();
def streetName = user?.address?.street
```

安全なナビゲーション演算子の呼び出しは、ネストすることもできます。

Groovy でこれに密接に関連するエルビス演算子 `?:` は、デフォルト値の場合に使用する Java の三項演算子を短縮します。例えば、以下の 2 つのコード行は等価です。

```
def name = user.name ? user.name : "Unknown" //traditional ternary operator usage

def name = user.name ?: "Unknown"  // more-compact Elvis operator
```

エルビス演算子は、演算子の左側の変数にすでに値が格納されている場合は (通常、これがデフォルトです)、それを維持しますが、値が格納されていない場合には新規の値を設定します。エルビス演算子は、より簡潔な式指向の三項演算子です。

Scala は `null` の概念を拡張し、関連するクラス `scala.Nothing` と併せて `null` をクラス (`scala.Null`) にしました。`Null` と `Nothing` は、Scala のクラス階層で最下位にあります。`Null` はすべての参照クラスのサブクラスであり、`Nothing` は他のすべての型のサブクラスです。

Scala には、値の不在を示すための `null` と例外の両方に代わる手段があります。コレクションでの Scala による処理の多くは (例えば、`Map` での `get` 処理など)、`Option` インスタンスを返します。このインスタンスには、`Some` または `None` のいずれか一方が含まれます (両方が含まれることはありません)。一例として、リスト 10 に REPL による対話を示します。

リスト 10. Scala で返される `Option`

```
scala> val designers=Map("Java" -> "Gosling", "c" -> "K&R", "Pascal" -> "Wirth")
designers: scala.collection.immutable.Map[java.lang.String,java.lang.String] =
    Map(Java -> Gosling, c -> K&R, Pascal -> Wirth)

scala> designers get "c"
res0: Option[java.lang.String] = Some(K&R)

scala> designers get "Scala"
res1: Option[java.lang.String] = None
```

リスト 10 では、`get` 処理が成功した場合には value に値が格納されて `Option[java.lang.String] = Some(value)` が返され、`get` 処理で対象が見つからなかった場合には `None` が返されます。コレクションから値をアンラップする手法では、パターン・マッチングが使用されます。パターン・マッチング自体が式であり、これを使用することにより、1 つの簡潔な式でアクセスとアンラップを実行することができます。

```
println(designers get "Pascal" match { case Some(s) => s; case None => "?"})
```

特に構文上のサポートを考えると、`Option` を使用したほうが、`null` を単独で使用する場合に比べ、値の不在をより適切に表現することができます。

まとめ

今回の記事では、Java 言語で問題となっている領域として、式、例外、`null` の 3 つを深く掘り下げました。Java.next 言語は Java から継承された欠点に対処していますが、その方法はそれぞれの言語の特徴を表しています。式の有無によって、一見無関係に思える概念 (例えば、例外など) のイディオムとオプションが変わります。これは、言語の機能が互いに高度に結合されていることを一層明確に示しています。

Java 開発者は習慣的に、継承が振る舞いを拡張する唯一の方法であると考えがちです。次回の記事では、Java.next 言語には継承よりも強力な数々の方法が用意されていることを明らかにします。

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。