実用的なGroovy: スムースな演算子

Javaプラットフォームで演算子多重定義？　GroovyならYesです！

Andrew Glover

2005年 10月 25日

Java™言語では演算子の多重定義（operator overloading）が禁止されていますが、若いGroovyは、「大歓迎」と言っています。長年使えずにいたものが、使えるようになりました。今回はAndrew Gloverが『実用的なGroovy』シリーズの最終回として、多重定義可能な演算子の3つのカテゴリーについて、ごく一般的な使い方を、手順を示しながら解説します。

C++から使い始めた開発者の多くは、+ や - のような多重定義演算子（overload operator）に懐かしさを感じているでしょう。多重定義演算子は便利なのですが、その多様性のために混乱を招く恐れがあり、Java言語では禁止されてきました。この制限を加えることによる利点は、明確さです。Java開発者は、2つのオブジェクトに対する + が、両者を加算するという意味なのか、一方を他方に付加するという意味なのかを迷う必要がありません。

このシリーズについて

どのようなツールであれ、開発作業の中に採り入れるためには、どういう場合に使うべきか、または使うべきではないかをよく知る必要があります。スクリプト言語（あるいは動的言語）は非常に強力なツールですが、その強力さは、適切なシナリオで適切な使い方をした場合にのみ発揮されます。『実用的なGroovy』シリーズはそういう点を念頭に置き、Groovyの実用的な使い方に焦点を絞って、どういう場合に、どのように使うのかを解説して行きます。

そこに、自由放任主義のGroovyが、ショートカットを持ち込んでくれたのです。今回の『実用的なGroovy』では、Groovyがサポートしており、演算子多重定義としても知られている、ad-hocポリモーフィズム演算子（operator ad-hoc polymorphism）を紹介します。C++の開発者なら誰でも口を揃えて言うように、これは手軽で楽しい手法なのですが、健全な注意をもって扱うべきものです。

スムースになった3種類

私はGroovyにある、多重定義可能な演算子を3つの論理グループ、つまり比較演算子、数学的演算子、配列的演算子、というグループに分類しました。これらのグループは、通常のJavaプログラミングで使用できる演算子のサブセットにすぎません。例えば & や ^ などの論理演算子は、現在のGroovyでは使用できません。

表1は、Groovyで使用できる、こうした3つのカテゴリーで多重定義可能な演算子を示しています。

表1. Groovyでの多重定義可能な演算子

1.	比較演算子は、通常のJava の**equals**や**compareTo**などの実装にマップされます
2.	Javaでの数学的演算子（ `+` や `-` 、 `*` など ）
3.	配列アクセス的な演算子（ `[]` ）

比較演算子

比較演算子は、Java言語に見られるequalsやcompareToなどの実装にマップされ、通常は集合でのソートのための簡略表現として使われます。表2は、Groovyにある、2つの比較演算子を示しています。

表2. 比較演算子

演算子	メソッド
a == b	`a.equals(b)`
a <=> b	`a.compareTo(b)`

演算子 == は、Java言語ではオブジェクトが等価なことを示す簡略表現ですが、オブジェクト参照のIDが等価なことを示すわけではありません。つまりGroovyで2つのオブジェクトの間に == が置かれると、両者はプロパティーが同じなので同じオブジェクトなのですが、それぞれのオブジェクトは別々の参照を指します。

そのJavadocによると、Java言語のcompareTo() メソッドは、そのオブジェクトが指定されたオブジェクトよりも小さいか、等しいか、あるいは大きい場合に、負の整数、ゼロ、あるいは正の整数を返します。このメソッドが返す値は、これら3つの値の内、どれか1つなので、Groovyは表3に示す4つの値で <=> 構文を補強しています。

表3. さらに4つの値

演算子	意味
a > b	a.compareTo(b) がゼロよりも大きな値を返す場合、この条件はtrueとなる
a >= b	a.compareTo(b) がゼロか、ゼロよりも大きな値を返す場合、この条件はtrueとなる
a < b	if a.compareTo(b) がゼロよりも小さな場合、この条件はtrueとなる
a <= b	a.compareTo(b) がゼロか、ゼロよりも小さな場合、この条件はtrueとなる

比較演算子の選択

皆さんは、私が「Feeling Groovy」の中で定義したディスコ狂いのLavaLampクラスを覚えているでしょうか。「Groovyの成長が加速」の中でも、JSR構文に移行する場合の例として、このクラスを

使用しました。比較演算子の持つ賢いトリックを説明するために、ここでも再度、このクラスを使うことにします。

リスト1では、通常のJavaのequals() メソッド (と、その犯罪パートナー、hashCode) を実装することによって、LavaLampクラスを機能強化しています。さらに、LavaLampがJava言語のComparableインターフェースを実装するようにし、compareTo() メソッド用の実装を作りました。

リスト1. LavaLampが帰ってきた！

```
package com.vanward.groovy

import org.apache.commons.lang.builder.CompareToBuilder
import org.apache.commons.lang.builder.EqualsBuilder
import org.apache.commons.lang.builder.HashCodeBuilder
import org.apache.commons.lang.builder.ToStringBuilder

class LavaLamp implements Comparable{
 @Property model
 @Property baseColor
 @Property liquidColor
 @Property lavaColor

  def String toString() {
    return new ToStringBuilder(this).
        append(this.model).
     append(this.baseColor).
     append(this.liquidColor).
     append(this.lavaColor).
     toString()
  }

  def boolean equals(obj) {
      if (!(obj instanceof LavaLamp)) {
         return false
      }
      LavaLamp rhs = (LavaLamp) obj
         return new EqualsBuilder().
             append(this.model, rhs.model).
             append(this.baseColor, rhs.baseColor).
             append(this.liquidColor, rhs.liquidColor).
             append(this.lavaColor, rhs.lavaColor).
             isEquals()
   }

   def int hashCode() {
      return new HashCodeBuilder(17, 37).
           append(this.model).
           append(this.baseColor).
           append(this.liquidColor).
           append(this.lavaColor).
           toHashCode()
   }

   def int compareTo(obj) {
      LavaLamp lmp = (LavaLamp)obj
         return new CompareToBuilder().
             append(lmp.model, this.model).
             append(lmp.lavaColor, this.lavaColor).
             append(lmp.baseColor, this.baseColor).
             append(lmp.liquidColor, this.liquidColor).
             toComparison()
   }
}
```

注記: 私は再利用の大ファンなので、これらの実装に関して（toString() も含めて）Jakartaの Commons Langプロジェクトに大きく頼っています。もし皆さんが相変わらず独自のequals() メソッドをコーディングしているのであれば、ぜひ一度、このライブラリーをチェックしてみるようにお勧めします（参考文献を参照のこと）。

リスト2では、リスト1で設定した演算子多重定義の実際を見ることができます。私は5つのLavaLampインスタンスを作りました（まるでパーティーをしているようです）。そして、それらをお互いに区別するために、Groovyの比較演算子を使っています。

リスト2. 比較演算子の実際

```
lamp1 = new LavaLamp(model:"1341", baseColor:"Black",
         liquidColor:"Clear", lavaColor:"Red")
lamp2 = new LavaLamp(model:"1341", baseColor:"Blue",
         liquidColor:"Clear", lavaColor:"Red")
lamp3 = new LavaLamp(model:"1341", baseColor:"Black",
         liquidColor:"Clear", lavaColor:"Blue")
lamp4 = new LavaLamp(model:"1342", baseColor:"Blue",
         liquidColor:"Clear", lavaColor:"DarkGreen")
lamp5 = new LavaLamp(model:"1342", baseColor:"Blue",
         liquidColor:"Clear", lavaColor:"DarkGreen")

println lamp1 <=> lamp2  //  1
println lamp1 <=> lamp3  // -1
println lamp1 < lamp3    //  true
println lamp4 <=> lamp5  //  0

assert lamp4 == lamp5
assert lamp3 != lamp4
```

lamp4とlamp5が同じであること、そして、他はお互いに微妙に異なることに注意してください。lamp1のbaseColorはBlackであり、lamp2のbaseColorはBlueなので、<=>は1を返しています（「black」の「a」は「blue」のuよりも前に来ます）。同様に、lamp3のlavaColorはlamp1のRedとは異なり、Blueです。条件、lamp1 <=>lamp3 は -1 を返しているので、ステートメント、lamp1 < lamp3 はtrueを返します。llamp4とllamp5は等しいので、<=>は0 を返します。

また、オブジェクトが等しいことに対して == がどのように動作するかが分かるでしょうか。lamp4とlamp5は同じです。もちろん、これをassert lamp4.equals(lamp5) を使って表現することもできますが、== の方がずっと速くできます。

参照IDをください

さて、もしオブジェクトの参照IDを本当にテストしたいとしたら、どうするのでしょう。== を使うことは当然できませんが、Groovyでは正にこうしたことのために、is() メソッドが用意されているのです。これをリスト3に示します。

リスト3.　is() ならうまく行きます！

```
lamp6 = new LavaLamp(model:"1344", baseColor:"Black",
             liquidColor:"Clear", lavaColor:"Purple")

lamp7 = lamp6
assert lamp7.is(lamp6)
```

リスト3を見ると分かるように、lamp6とlamp7は同じ参照なので、isはtrueを返します。

ところで、皆さんは少しめまいを感じたとしても、驚くことはありません。多重定義演算子を使うということは、Groovy言語をほとんど逆行させているのです。ただし、非常に楽しい方法で後戻りしているのだと私は思います。

数学的な演算子

Groovyでは、多重定義のために、次のような数学的演算子をサポートしています。

表3. Groovyでの数学的演算子

演算子	メソッド
a + b	`a.plus(b)`
a - b	`a.minus(b)`
a * b	`a.multiply(b)`
a / b	`a.divide(b)`
a++ or ++a	`a.next()`
a-- or --a	`a.previous()`
a << b	`a.leftShift(b)`

皆さんは、Groovyでの + 演算子が、特に集合に関して、既に幾つか他の領域でも多重定義されていることに気がついたかも知れません。なぜこんなことが可能なのか、あるいは、自分独自のクラスでも同じことをしてみたい、と思ったのではないでしょうか。では、それを調べてみましょう。

演算子を加え合わせる

皆さんは、「JavaアプリケーションにGroovyを混ぜ込む」の中でのSongクラスを覚えているでしょうか。これらのSongを再生するために、jukeboxオブジェクトを作ったらどうなるかを見てみましょう。リスト4では、MP3を実際に再生するための詳細は無視し、JukeboxにSongを追加したり削除したりすることに焦点を当てています。

リスト4. これがJukeboxだ！

```
package com.vanward.groovy

import com.vanward.groovy.Song


class JukeBox {

  def songs

  JukeBox(){
    songs = []
  }

  def plus(song){
    this.songs << song
```

```
  }

  def minus(song){
    def val = this.songs.lastIndexOf(song)
    this.songs.remove(val)
  }

  def printPlayList(){
     songs.each{ song -> println "${song.getTitle()}" }
  }
}
```

plus() メソッドとminus() メソッドを実装することによって + と - の両方を多重定義し、それらをスクリプトの中で使います。リスト5は、プレイリストに曲を追加削除する場合の、多重定義した + と - の振る舞いを示しています。

リスト5. ごきげんな、あの曲をかけてくれ

```
sng1 = new Song("SpanishEyes.mp3")
sng2 = new Song("RaceWithDevilSpanishHighway.mp3")
sng3 = new Song("Nena.mp3")


jbox = new JukeBox()
jbox + sng1
jbox + sng2
jbox + sng3

jbox.printPlayList() //prints Spanish Eyes, Race with the Devil.., Nena

jbox - sng2

jbox.printPlayList() //prints Spanish Eyes, Nena
```

多重定義された多重定義を多重定義する

この、多重定義というテーマを追い続けている中で、皆さんは表3の中で、多重定義可能な数学的演算子の1つが << であること、そしてそれがGroovyの集合に対しても多重定義されていることに気がついたかも知れません。集合の場合、<< は集合の最後に値を付加することによって、多重定義されて通常のJavaのadd() メソッドのように動作します（これはRubyと非常に似ています）。リスト6では、この振る舞いを真似て、Jukeboxのユーザーが、+ 演算子だけではなく << 演算子を使ってもSongを追加できるようにしています。

リスト6. その曲を左シフトしてくれ

```
def leftShift(song){
   this.plus(song)
}
```

リスト6では、プレイリストにSongを追加するためにplusメソッドを呼ぶleftShiftメソッドを実装しています。リスト7は、<< 演算子の実際を示しています。

リスト7. レースが始まった

```
jbox << sng2 //re-adds Race with the Devil...
```

ご覧の通り、Groovyの数学的な多重定義演算子は非常に多くを定義できるだけではなく、それを非常に速く行うことができるのです。

配列的な演算子

Groovyでは、Javaでの標準配列アクセス構文、 [] の多重定義をサポートしています。これを表4に示します。

表4. 配列演算子

演算子	メソッド
a[b]	`a.getAt(b)`
a[b] = c	`a.putAt(b, c)`

配列アクセスのための構文は、非常によく集合にマップできるので、私はJukeBoxクラスを更新して、両方のケースを多重定義するようにしました。これをリスト8に示します。

リスト8. 音楽を多重定義

```
def getAt(position){
  return songs[position]
}

def putAt(position, song){
  songs[position] = song
}
```

これでgetAtとputAtの両方を実装したので、リスト9に示すように [] 構文を使うことができます。

リスト9. これよりも速くできるものがあるでしょうか

```
println jbox[0] //prints Spanish Eyes

jbox[0] = sng2  //placed Race w/the Devil in first slot
println jbox[0] //prints Race w/the Devil
```

よりGroovyなJDKメソッド

演算子多重定義の概念と、それがGroovyの中でどのように行われるかを理解できると、通常のJavaオブジェクトの多くが、Groovyを書いた人達によって既に機能強化されていることに気がつきます。

例えば、CharacterクラスはcompareTo() をサポートしています。これをリスト10に示します。

リスト10. Characterを比較する

```
def a = Character.valueOf('a' as char)
def b = Character.valueOf('b' as char)
def c = Character.valueOf('c' as char)
def g = Character.valueOf('g' as char)

println a < b  //prints true
println g < c  //prints false
```

同様に、StringBufferにも <<演算子を付加することができます。これをリスト11に示します。

リスト11. バッファーにあるストリング

```
def strbuf = new StringBuffer()
strbuf.append("Error message: ")
strbuf << "NullPointerException on line ..."

println strbuf.toString() //prints Error message: NullPointerException on line ...
```

そして最後に、リスト12は、Dateが + と - で操作できることを示しています。

リスト12. 今日は何日でしょう

```
def today = new Date()

println today   //prints Tue Oct 11 21:15:21 EDT 2005
println "tomorrow: " + (today + 1)  //Wed Oct 12 21:15:21 EDT 2005
println "yesterday: " + (today - 1) //Mon Oct 10 21:15:21 EDT 2005
```

まとめ

ad-hocポリモーフィズム演算子（operator ad-hoc polymorphism）、つまり私達にとっての演算子多重定義は、注意深く使用し、適切に文書化する限り、非常に強力なものです。ただし、この機能を乱用すべきではありません。何か変わったことをするために演算子を多重定義しようとする場合には、その意図を明確に文書化すべきです。多重定義をサポートするようにGroovyクラスを機能強化するのは、驚くほど単純です。豊富で手軽なショートカットが利用でき、それから得られる結果を考えれば、使い方に注意することや、使い方の意図を適切に文書化する手間は当然な代償と言うべきでしょう。

関連トピック

- 演算子の多重定義が素晴らしいと思うのでしたら、「実用的なGroovy: MOPとミニ言語について」（ Andrew Glover著, developerWorks, 2005年9月 ）を読んで、Meta Object Protocolについて調べてください。
- 演算子の多重定義という柔軟性を持つ、もう一つのJavaベースの言語、Jythonについて学ぶために、「alt.lang.jre: Jythonを知る」（ Barry Feigenbaum著, developerWorks, 2004年7月 ）を読んでください。
- Andrewは「Interview with Guillaume Laforge」の中で、JSRについて、誰がGroovyを使っているのか、またGroovyの将来などについて、Groovyのプロジェクト・マネージャーと語り合っています。
- 『実用的なGroovy』シリーズは、Groovyプログラマーのためのヒントと手法を解説したシリーズです。
- Java technologyゾーンには、Javaプログラミングのあらゆる面を網羅した記事が豊富に用意されています。
- Commons Langには、String操作方法を含めて、Javaプラットフォームでの中核クラスを処理するための様々なメソッドが置かれています。