

Scalaらしいライブラリ の作り方(仮)

水島 宏太(みずしま)

自己紹介

- 氏名：水島宏太
 - <http://d.hatena.ne.jp/kmizushima/>
 - 筑波大学の大学院生
 - 構文解析の研究とかしてます
 - プログラミング言語が好き
 - Onion: <http://www.onion-lang.org/>
 - Scala布教中
 - Scala関係の日記を巡回してツッコミ入れたり
-

はじめに

- 「Scalaらしい」ライブラリを作るための基本的なテクニックについて紹介
 - ファーストクラスの関数
 - implicit conversion
 - 演算子多重定義
 - Java 7のクロージャを使ったコードとの比較も行います
 - Scalaの概要についても軽く紹介します
 - 時間の都合上文法の説明ははしょります
-

Scalaってなんて呼ぶの？

- スカラ？
 - スケーラ？
 - Google Tech talkでの作者の講演映像
 - <http://video.google.com/videoplay?docid=553859542692229789>
 - 作者の発音： スカラ…と聞こえる
 - とりあえず、スカラで良さそう
-

Scalaってなに？（その1）

- ポストJavaの候補として注目を集めている
 - …かもしれないプログラミング言語
 - 「純粋な」オブジェクト指向言語
 - 全ての値はオブジェクト
 - オブジェクト指向と関数型の融合
 - クロージャ、パターンマッチングなど
 - Martin Odersky教授により開発
 - スイスローザンヌ工科大学
 - Java Genericsの貢献者の一人
-

Scalaってなに？（その2）

- 高速な処理系
 - Javaとほぼ同等の実行速度（書き方による）
 - 他のJVM用スクリプト言語よりもかなり高速
 - 静的型付けでありながら簡潔な記述
 - 型推論、文末の;不要、()の省略などなど
 - Javaのライブラリをシームレスに使える
 - JVM上で動作する処理系
 - Javaと親和性の高い静的型付け言語
-

Scalaってなに？（その3）

hello, world in Scala

```
object HelloWorld {  
  def main(args :Array[String]) :Unit = {  
    println("hello, world")  
  }  
}
```

- **object**でシングルトンオブジェクトを定義
 - **def**でメソッドを定義
 - mainメソッドがエントリポイント=Javaと同じ
 - `Array[String] ≐ String[]`
 - `Unit ≐ void`
-

Scalaの問題点

- コンパイルが遅い
 - javacと比べて2倍以上遅い(環境による)
 - fscである程度改善可能
 - プロセスとしてScalaコンパイラが常駐
 - 開発環境のサポートがやや弱い
 - EclipseやNetBeansのプラグインが存在するが、安定性などの面で問題あり
 - 標準ライブラリが偏っている
-

標準ライブラリの問題点

- とにかく偏っている
 - 充実してるもの
 - コレクション、XML、構文解析、Actorなど
 - 無いか、あっても貧弱なもの
 - IO、ネットワーク、GUIなど
 - 無いものはJavaのライブラリを利用する必要
-

Javaのライブラリの利用

- 特別な作業無しに可能
 - JRuby, Jythonなどと比べても容易
 - JavaのライブラリはScalaの機能を前提に作られていない
 - Javaの言語仕様に利便性が縛られる
 - Scalaの便利な機能を活用できない
- 「Scalaらしい」ラッパーライブラリを
-

Scalaらしい(Javaと比較して)機能

- implicit conversion
 - 既存クラスにメソッドを追加できる
 - かのように見せかけることができる機能
 - ファーストクラスの関数
 - 俗に言う「クロージャ」
 - trait
 - 実装を持てるinterfaceのようなもの
 - 演算子多重定義
 - 無数の演算子を定義できる(<->とか)
-

Case 1: java.io.File

- 問題：あるディレクトリの中で指定した拡張子を持つファイルのみを抽出せよ（再帰的にディレクトリをたどる必要は無いものとする）

Case 1: Javaによる解

```
File[] files = new File(args[0]).listFiles(  
    new FileFilter() {  
        public boolean accept(File file) {  
            return file.getName().endsWith(args[1]);  
        }  
    }  
);
```

- File#listFiles(FileFilter) を利用
 - FileFilterの無名サブクラスを生成
-

Case 1: Scalaによる解その1 (ライブラリをそのまま利用)

```
val files = new File(args(0)).listFiles(  
  new FileFilter {  
    def accept(file: File) = file.getName.endsWith(args(1))  
  }  
)
```

- あまり簡潔になっていない
 - 型宣言が減った分くらい
- FileFilterの無名サブクラスを生成
 - Javaと同様

Case 1: Scalaによる解その2 (ラッパーライブラリを作成)

```
val files =
```

```
new File(args(0)).filter {_.getName.endsWith(args(1))}
```

- かなり簡潔になった
 - Fileクラスにfilterメソッドを追加
 - Fileを引数に取りBooleanを返すメソッド
 - インタフェース型を関数型で置き換える
 - 無名関数を利用
-

Case 1: Scalaによる解その2 - 実装

```
class RichFile(src: File) {  
  def filter(f: File => Boolean) = src.listFiles(  
    new FileFilter { def accept(file: File) = f(file) }  
  )  
}
```

```
implicit def enrichFile(file: File) = new RichFile(file)
```

- 追加したいメソッドを適当なクラスに定義
- File => Boolean
 - Fileを受け取ってBooleanを返す関数型
 - function type
- implicit defによってメソッドの追加を定義
 - implicit conversion

Case 1: Java 7 (BGGGA) による解

```
File[] files = new File(args[0]).listFiles(  
    {File file => file.getName().endsWith(args[1])}  
);
```

■ Scalaと比較すると…

- 無名関数→インタフェース型の自動変換
 - Closure Conversion
- 無名関数の仮引数の型を省略できない
- ローカル変数の型を省略できない

Case 2: java.io.Reader

- 問題：指定されたファイルの中で、指定された文字列にマッチする行のみを表示せよ
-

Case 2: Javaによる解

```
BufferedReader reader =
    new BufferedReader(new FileReader(args[0]));
try {
    for(String line; (line = reader.readLine()) != null;) {
        if(line.indexOf(args[1]) >= 0) {
            System.out.println(line);
        }
    }
} finally {
    try { reader.close(); } catch(IOException e) {}
}
```

Case 2: Scalaによる解その1 (ライブラリをそのまま使用)

```
val reader = new BufferedReader(new FileReader(args(0)))
try {
  var line: String = null
  while({line = reader.readLine; line != null}) {
    if(line.indexOf(args(1)) >= 0) {
      println(line)
    }
  }
} finally {
  try { reader.close } catch { case e: IOException => }
}
```

Case 2: Scalaによる解その2 (ラッパーライブラリを作成)

```
new File(args(0)).openReader {reader =>
  reader.eachLine {line =>
    if(line.indexOf(args(1)) >= 0) {
      println(line)
    }
  }
}
```

- readerのclose不要
 - openReaderの終了後、自動でcloseされる
 - eachLineで各行ごとに繰り返し
 - 各行の内容がlineに代入されて呼び出される
-

Case 2: Scalaによる解その2 - 実装

```
class RichBufferedReader(src: BufferedReader) {  
  def eachLine(f: String => Unit) { ... }  
}  
class RichFile(src: File) {  
  def openReader[T](f: BufferedReader => T) = { ... }  
}  
implicit def enrichBufferedReader(src: BufferedReader) = {  
  new RichBufferedReader(src)  
}  
implicit def enrichFile(src: File) = new RichFile(src)
```

- 基本的にCase 1と同じ手法
 - implicit conversion + function type
-

Case 2: Java 7 (BGGA) による解

```
openReader(BufferedReader reader:new File(args[0])) {  
  eachLine(String line:reader) {  
    if(line.indexOf(args[1]) >= 0) {  
      System.out.println(line);  
    }  
  }  
}
```

- Scala版と同様にreaderのclose不要
 - function typeを最後の引数に取るメソッドのための専用構文
-

Case 2: Java 7 (BGGGA) による解 - 実装 -

```
public static <T, throws E> T openReader(  
    File file, {BufferedReader ==> T throws E} f  
) throws E, FileNotFoundException { .. }  
public static <throws E> void eachLine(  
    BufferedReader reader, {String ==> void throws E} f  
) throws E, IOException { .. }
```

- シグネチャの宣言が煩雑
 - throwsの嵐
 - 上位層に例外を丸投げするために必要な処理
 - staticメソッドとして実装
-

Case 3: `java.math.BigInteger`

- 問題：指定された数の階乗を計算せよ

Case 3: Javaによる解

```
static BigInteger fact(BigInteger n) {  
    if (n.compareTo(BigInteger.valueOf(2)) < 0)  
        return BigInteger.ONE;  
    else  
        return n.multiply(fact(n.subtract(BigInteger.ONE)));  
}
```

...

```
System.out.println(fact(new BigInteger(args[0])));
```

- メソッド呼び出しがネストして読みづらい
 - 演算子多重定義があれば...
-

Case 3: Scalaによる解その1 (ライブラリをそのまま使用)

```
def fact(n: BigInteger): BigInteger = {  
  if((n compareTo BigInteger.valueOf(2)) < 0)  
    BigInteger.ONE  
  else  
    n.multiply(fact(n.subtract(BigInteger.ONE)))  
}  
println(fact(new BigInteger(args(0))))
```

- Java版とほとんど変わり無し
-

Case 3: Scalaによる解その2 (ラッパーライブラリを作成)

```
def fact(n: BigInteger): BigInteger = {  
  if(n < 2) 1 else n * fact(n - 1)  
}  
println(fact(args(0).b))
```

- Intと同じようにBigIntegerを扱えている
 - 演算子の多重定義
 - implicit conversionによる暗黙の型変換
 - IntからBigInteger
 - Java 7でも真似できない
 - 実はscala.BigIntがあるので不要だったり
-

Case 3: Scalaによる解その2 - 実装

```
class RichBigInteger(l: BigInteger) extends
  Ordered[BigInteger] {
  def +(r: BigInteger) = l add r
  def compare(that: BigInteger) = (l compareTo that)
  ...
}
```

```
class RichString(src: String) { ... }
implicit def enrichBigInteger(src: BigInteger) = ...
implicit def enrichString(src: String) = ...
implicit def int2BigInteger(src: Int): BigInteger = ...
```

- 通常の方法と同様に演算子を定義可能
- Ordered traitの利用
 - compareを実装すれば、<, >, ... が利用可能に

まとめ

- Scalaの概要について駆け足で紹介
 - 「Scalaらしい」ライブラリを作るための基本的なテクニックについて説明
 - implicit conversionによるメソッドの追加
 - ファーストクラスの関数の活用
 - 演算子多重定義
 - より快適なScala Lifeを！
-