

Scala

関数型言語

- Scalaはオブジェクト指向+関数型なので、関数型の特徴を復習

オブジェクト指向プログラミング

- クラス、継承、メソッド、カプセル化、ポリモフィズム

⇒Javaはみんな知っているので省略

関数型プログラミング

- 参照透過性、高階関数、関数のカリー化、部分適応、遅延評価

関数型言語

- ・参照透過性

⇒状態を持たない、変数に再代入しない(副作用を起させない)

⇒同じ関数の評価は必ず同じ値になる

⇒ループを使わない

⇒ループは副作用を前提としているため使わない

⇒(末尾)再帰を使う

⇒いつどんなタイミングで実行しても結果が変わらない

⇒並列計算に向く

フィールド変数、スタティック変数、入出力、DBアクセス・・・禁止

関数型言語

- ・ 第1級関数

⇒ 第1級オブジェクト(基本的な操作に制約を受けない)に関数が属する

⇒ 関数を関数の引数や戻り値に使える

- ・ 関数を引数や戻り値にする関数を高階関数

関数型言語

- ・JSで高階関数の例

```
// 関数を変数に入れる
```

```
var foo = function{} {console.log('foo')};  
foo();
```

```
// 関数を引数として受け取る
```

```
function bar(foo){  
  foo();  
}
```

関数型言語

・関数のカリー化

引数が複数ある関数を引数が 1 つの関数のチェーンで表す

数学的 : $f(x, y) = z \Rightarrow f(x) \circ g(y) = z$

ラムダ式 : $\lambda x y. x + y \Rightarrow \lambda x. (\lambda y. x)$

非カリー関数

```
function max(x, y){  
    return x > y ? x : y;  
}
```

カリー化関数

```
function _max(x){  
    return function(y){  
        return x > y ? x : y;  
    }  
}
```

関数型言語

- ・まとめて実行

```
var plus = function (x) {  
  return function (y) {  
    return x + y;  
  }  
}
```

plus(1)(2); \Rightarrow 3

$x = 1$ として、内部の関数が $1+y$ の関数を返すイメージ

関数型言語

- ・部分的に分けて行うことも可能

```
function powBaseBy(x) {  
  return function(y) {  
    return Math.pow(x, y);  
  };  
}
```

部分適用という

```
var pow2 = powBaseBy(2); // 2 のべき乗を求める関数  
console.log(pow2(1)); // -> 2
```

```
var pow3 = powBaseBy(3); // 3 のべき乗を求める関数  
console.log(pow3(2)); // -> 9
```

関数型言語

「1つの関数が1つの引数を確定させ、残りの部分を別の関数に任せる」
という作業を組み合わせて全体を作っていく
→関数型プログラミングの基本的な考え方

メリット

- ・1つの関数の担当が分かりやすくなる
⇒ 处理が段階的になる
- ・部分実行がしやすくなる
- ・1つの関数を再利用しやすい
- ・より数学の合成関数の考えに近くなる
⇒ 小さいの関数の集合で表現できる

関数型言語

- ・ラムダ計算 \Rightarrow 関数の別の記法

引数のパターンと結果の計算式からなる関数名を含まない式

$\text{add}(x,y) = x + y$ //名前 入力 = 出力

↓ 同じ意味

$\text{add} = \lambda xy. x+y$ //名前 = λ 入力.出力

【左辺】 add という関数

【右辺】 x と y という引数を取り、 $x+y$ をする関数

右辺に全ての情報があり名前が消えた(=無名関数で規則だけが残る)ことが大切

言い方の違いで同じことを言っている

関数型言語

- 遅延評価

⇒使われるまで評価をして値に落とさない

⇒意図しない実行時例外を回避できることも多い

⇒使わない部分を評価する無駄を省ける

```
def square(n: => Int): Int = {
```

lazy m = n // mが初めて現れるところで評価され、キャッシュされる

m + m

```
}
```

関数型言語

- 末尾再帰

末尾再帰は、再帰処理を関数の末尾に持ってくること。

これにより、呼び出し先の処理の終了 = 呼び出し元の処理の終了のため、戻り先を保存する必要がない。

この手法により、スタックの累積を軽減又は不要とし、効率の向上する。

```
public int fact(int n){  
    if(n == 1) return 1;  
    else return n* fact(n-1);  
}  
  
public int fact(int n,int acc){  
    int(n == 1) return acc;  
    else return fact(n-1,acc*n);  
}
```

関数型言語

- ・モナド 文脈を持つ計算を扱う仕掛けのこと

倍返し: 数値 => 変換(2倍)=> 数値 があるとする

しかし、現実的には色々な要求が舞い込んでくる。

1. エラーが発生したら、それをきちんと知らせて欲しい
2. 何回「倍返し」したのか記録しておいて欲しい
3. 倍返しする度にログを吐いて欲しい

倍返し1:数値 => 変換(2倍)=> 数値, エラー

倍返し2:数値 => 変換(2倍)=> 数値, 回数

倍返し3:数値 => 変換(2倍)=> 数値, ログ

関数型言語

「倍返し」の例を一般化すると、

倍返し:数値 => 変換(2倍) => 数値, おまけ

モナドとは、

[本来の入力, おまけ] => 本来の入力

出力値 => [本来の出力, おまけ]

を実現する仕掛けをいう

関数型言語

JavaにおけるOptionalがMaybeモナドに相当する

数値(値がある/無い) => 変換(2倍)=> 数値(値がある/無い)

様々なモナドがある

- Maybeモナド(失敗の可能性を持つ)
- IOモナド(副作用を伴う)
- リストモナド(複数の候補の可能性を持つ)
- Stateモナド(状態の引継ぎをする)
- Writer(並列計算)

これにより、元々の関数は純粹性を維持できる

関数型言語

JavaにおけるOptionalがMaybeモナドに相当する

Scala

Scalaの特徴

- ・マルチパラダイム言語(オブジェクト指向+関数型)
- ・静的型付けなので、コンパイル時のエラー検出が強い
- ・クラスを簡単に作れ、クラスに関する機能が豊富なので、意味を持たせやすい
- ・型推論がある
- ・関数型の特徴を持つ(カリー化、クロージャ、遅延評価)
- ・Mix-inやクラス拡張の仕組みがある
- ・Javaと極めて高い互換性を持つ

Scala

変数宣言

- val (Immutable)、var(mutable)を意識する
- 基本的に valのみで構成するようにする
⇒副作用で状態を変えない。

全てが式

- 全てがなんらかの値を返す

```
val flag = if(x > y) { · · · } なものもOK
```

Scala

条件分岐

if(条件) 真の場合 [else 偽の場合]

ループ

while(条件式) 真の場合

for(ジェネレータ1;ジェネレータ2;ジェネレータn)

ジェネレータは xが1から10までのようなものに相当

ジェネレータが2つだと2重ループ、 3つだと3重ループ

Scala

```
for(x <- 1 to 5; y <- 1 until 5)
```

↓

```
for(x = 1 ; x <= 5; x++){  
  for(y = 1 ; y < 5 ;y++){  
  }  
}
```

とおなじ

```
for(x <- 1 to 5; y <- 1 until 5 if x != y)
```

等しくないものだけ処理する

Scala

```
val l = List("A", "B", "C", "D", "E")
val prel = for(e <- l) yield { | "Pre" + e | } // 全ての要素にPreを追加
println(l) // A,B,C,D 元々のオブジェクトは変わらない
println(prel) // PreA,PreB,PreC,PreD
```

Scala

```
マッチ対象の式 match {  
    case パターン1 [if ガード1] => 式1  
    case ...  
    case パターンN [if ガードN] => 式N  
}
```

- ・ガードは、パターンにマッチするが、除外したいものを付ける
 - ・フォールスルー(次のケースに移る) ことはしない
- まとめときは「case パターン1 | パターン2」のように書く
- ・case _ XXX とかくとdefaultとなる

Scala

- ・パターンマッチには様々なものが使える

正規表現 ⇒ 特定の文字列にマッチ

コレクション ⇒ 特定の要素にマッチ(ネストも可能)

クラスの継承 ⇒ String, Integer や継承関係でマッチ

Scala

```
class ClassName(parameter1: Type1, parameter2: Type2, ...) {
```

フィールド、メソッドの定義

```
}
```

```
class Point(_x:Int, _y:Int){
```

```
  println("hello")
```

```
  val x = _x
```

```
  val y = _y
```

<メソッドの定義>

```
}
```

クラス定義直後からコンストラクタの定義

Javaのようにクラス名と同名のメソッドではない

Scala

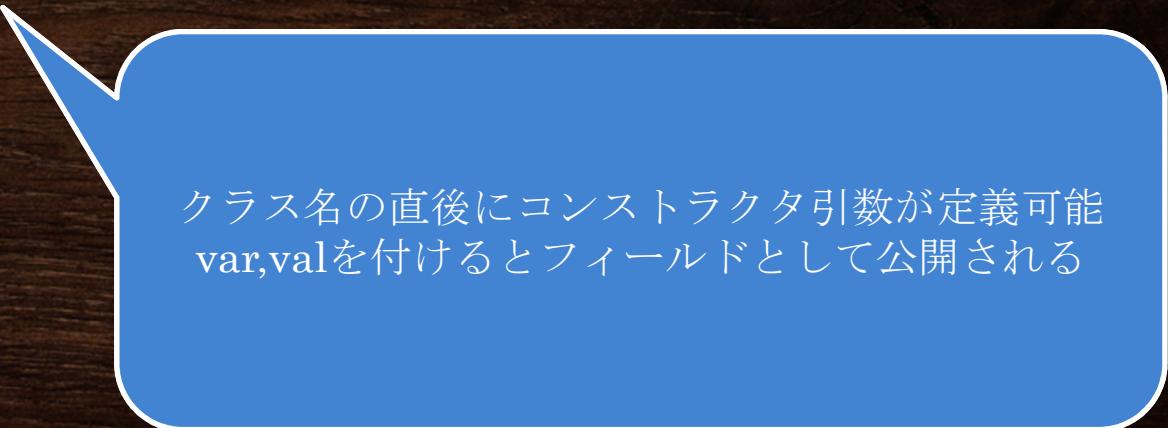
- コンストラクタの引数と同名のフィールドを定義し、それを公開する場合は、以下のように短く書くこともできます。

```
class Point(val x: Int, var y: Int)
```

↓以下のJavaと同義

```
class Point{  
    public final int x  
    public int y  
    public Point(int x, int y){  
        this.x = x;  
        this.y = y;  
    }  
}
```

point.x のようにアクセスができるようになる



クラス名の直後にコンストラクタ引数が定義可能
var, valを付けるとフィールドとして公開される

Scala

メソッドの定義

メソッド修飾子 def メソッド名 戻り型 = (パラメタ1:型1,パラメタ2:型2, ...){

//メソッド定義

}

```
def +(p:Point) Point ={
  new Point(x+p.x,y+p.y)
}
```

Scala

引数リストを分割

def メソッド名(引数リスト...)(引数リスト...)のように定義できる

def add(x:Int)(y:Int) : Int = x+y

adder.add(x)(y)で呼び出す

def add(x:Int,y:Int) : Int = x +y

adder.add(x,y)で呼び出す

Scala

抽象クラスはabstractを使う

```
abstract class myAbstractClass
```

継承はextendsを使う

```
class SubClass(...) extends SuperClass
```

ただし、メソッドのオーバーライドには必ず overrideキーワードを使う

⇒使わないとコンパイルエラー

```
override def print(): Unit = { println("OK") }
```

※Unitはvoidに相当する空の戻り値を表す型

Scala

- ・ Scalaは全ての値がオブジェクト
- ・ 全てのメソッドは何かのオブジェクトに属する
- ・ したがって、Javaでいうクラスに属するstaticはない
⇒object構文を使ってオブジェクト特有のメソッドを作る

```
class Point(val x:Int, val y:Int)  
object Point(def apply(x: Int,y: Int): Point = new Point(x,y)}
```

staticやファクトリメソッドやSingletonになるらしいが、まだ良く分からぬ
なお、同じファイル内でクラスと同じ名前を持つオブジェクトをコンパニオンオブ
ジェクトと呼び、アクセス権などに特権を持つ
よくわからん

Scala

トレイトを使ってMix-Inを実現する(RubyでいうところのModule?)

trait トレイト名 { フィールド/メソッドの定義}

- ・トレイトはインスタンス化できない ⇒ 単体で使うことが想定されない
- ・コンストラクタの引数を取れない ⇒ インスタンス化しないので問題なし

class ClassA extends SuperClass with TraitA with TraitB

のように使う

多重継承を避け、複数の別々の名前空間を持つモジュールを1つのクラスに取り入れたいときに使う

class ClassB extends TraitC はOK

Scala

雑型継承はダメ

$A \Rightarrow B \Rightarrow D$

$A \Rightarrow C \Rightarrow D$

線形継承は定義順に処理される

$A \Rightarrow B \Rightarrow C$

$A \Rightarrow B' \Rightarrow C$

Scala

- 型パラメタ

```
class クラス名[型パラメタ…] (コンストラクタ引数:コンストラクタ型) …
```

```
class cell[T] car value: T){
```

```
  def put (newvalue: T) : unit = { value = newvalue}
```

```
  def get(): T = value
```

```
}
```

```
val cell = new Cell[Int](1)
```

```
cell.put(2)
```

```
cell.get()
```

型パラメタにはJava同様特定のクラスのサブクラスやスーパークラスを定義のみを許容するように可能

Scala

- Scalaにおける関数はトレイト
- メソッド≠関数なので注意

```
val add = new Function2[Int,Int,Int]{def apply(x:Int,y:Int): Int = x+y}  
add.apply(10,20)  
add(10,20) //OK
```

Function2は引数が2個なので2
無名関数とシグニチャントラックスで
val add = (x:Int,y:Int) => x+y
add(10,20)でも使える

Scala

関数のカリー化

```
val addCurried = (x: Int) => ((y: Int) => x + y)  
addCurried(100)(200)
```

メソッドと関数の違い

メソッドはdefで定義されるクラス内のもの。(オブジェクト指向のメソッド)

⇒メソッドは、変数に代入できない

関数は、第1級関数として、持ち運びや代入ができる、カリー化される処理のまとめ

Scala

高級関数

```
def double(n: Int, f: Int => Int): Int = { | f(f(n)) | }
```

「int」と「intを引数にしてintを出力する関数」を引数にして、intを出力
nに関数を2回適応する

double(1,m=> m*2) : 1*2*2で4

double(2,m=> m*3) : 2*3*3で18

Scala

コレクション操作

- 要素の変更が可能なmutableなコレクションに慣れ親しんでいるが、immutableなコレクションを使うようにする

array -> mutable

list -> immutable

map -> mutable or immutable

set -> mutable or immutable

Scala

array -> mutable list -> immutable

```
val arr = array[Int](1,2,3,4,5)  
array(0) = 7 //OK
```

```
val lst = list(1,2,3,4,5)  
lst(0) = 7 // NG
```

⇒新しいリストを作つて使う

```
var lst1 = list(1,2,3,4,5).map(x => x*2)  
var lst2 = list(1,2,3,4,5).filter(x => x%2 == 1)  
var lst3 = lst(1,2,3,4,5).count(x => x %2 == 0) // 2
```

Scala

map -> mutable or immutable

set -> mutable or immutable

なにも指定しないとimmutableなmapが使われる

```
val m1 = Map("A" -> 1, "B" -> 2)
```

m1.updated("B",4) //更新したmapを返すため、 m1は置き換わらない

mutableにしたい場合は、 mutableを指定する

```
val m2 = mutable.Map("A" -> 1, "B" -> 2)
```

m2("B") = 4 // m2が置き換わる

Scala

- ・他にもいろいろあるけど、ここまで

Scala

参考文献

https://dwango.github.io/scala_text/collection.html