Introduction à Scala & Akka

Groupe Madz

Plan

1. Définition générale et tour d’horizon

2. Nouveaux concepts en Scala

3. Akka

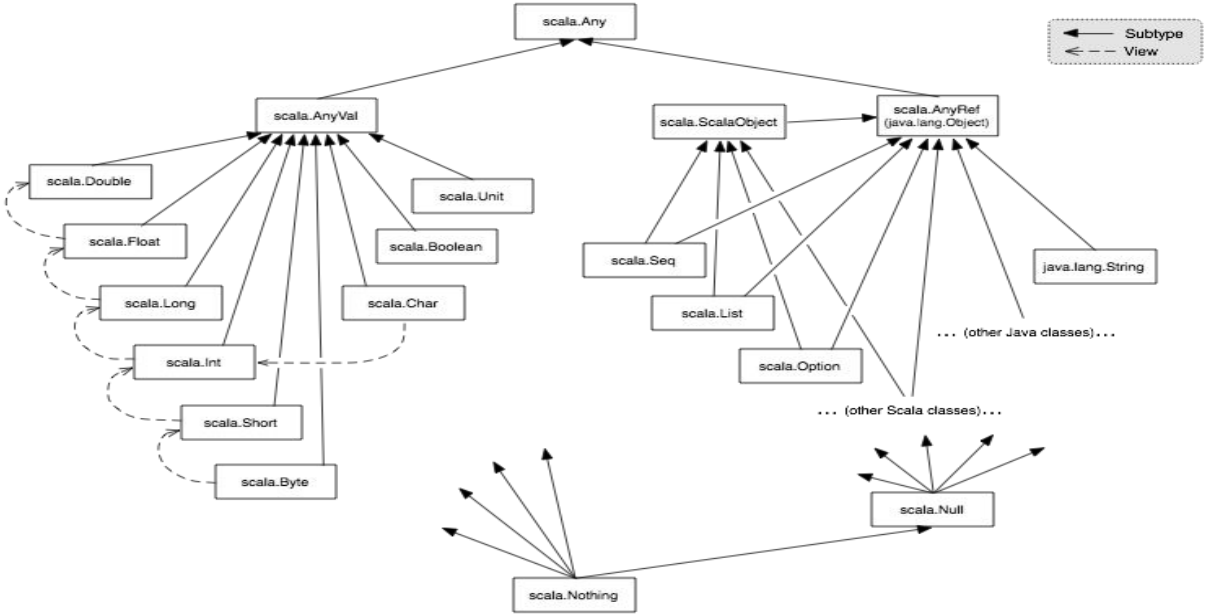
Scala en bref

- Scala pour “Scalable language”
- Conçu par **Martin Odersky** de l’EPFL.
- Scala est un langage haut-niveau et multi-paradigme

- C’est un langage orienté-objet très influencé par Java
- Model objet pur (!= Java): Uniquement object et appels de méthodes
- Tous les programmes Scala peuvent être exécutés sur la JVM
- Typage statique: pas de type à spécifier dans la majorité des cas

Sous-typage : la hiérarchie des classes

- ➔ Il n'y a pas de concept de données primitives car **tout est un objet dans Scala**.
- ➔ **T'** est un sous-type de **T** (noté **T'<:T**) ,
- ➔ si toute fonction opérant sur des valeurs de type **T** peut être utilisé sur des valeurs de type **T'**



Objets, variables, types primitifs, fonctions

➔ Différence entre val et var:

```
var x = 5
val y = 5

x = x+1
y = y+1 // Erreur : "Reassignement to val"
```

➔ Définition de fonction :

```
def carre(valeur : Int) : Int = {
    return valeur * valeur
}

def carre(valeur : Int) = valeur * valeur
```

➔ Syntaxe du for est spécifique à Scala :

```
for(i <- 0 to 9) { ...}
for (i<- 10 to (1,-2)) {...}
```

Collections

➔ **Les tableaux: Array[A]**

- val t = new Array[String](3)
- t(0)=”zéro”;t(1)=”un”;t(2)=”deux”
- val t =new Array(“zéro”,”un”,”deux”)

➔ **Les listes : List[A]**

- val l=List(1,2,3)
- val l1=0::l //Ajouter un élémnet en téte d’une liste
- val l2=l1:+4 //Ajouter un element en queue d’une liste
- val l3=l1++l2 //Concatener des listes
- val i=l(2) //Consulter la valeur d’un element a une position donnée
- l(2)=6 //Echoue!

➔ **Tableaux Associatifs : Map[String,Int]()**

- val m = Map("zéro" -> 0, "un" ->1)
- val zero=m(“zéro”) // zero =0
- m(“deux”)=2 // Echoue!
- val n = scala.collection.mutable.Map[String,Int]()
- n(“zéro”)=0

Pattern Matching

➔ il est possible d'écrire des pattern matching, Tout comme OCaml, pour cela on fait appel au mot clé **case**.

```
abstract class Tree

case class Node(l:Tree, r:Tree) extends Tree
case class Leaf(v:Int) extends Tree

val t = Node(Node(Leaf(1),Leaf(2)),Leaf(3))

def sum(t:Tree):Int = {
    t match {
    case Leaf(v) => v
    case Node(l,r) => sum(l)+sum(r)
    }
}

sum(t)
```

Class, Object, Trait,

- ➔ Classe, Héritage et Classes Abstraites comme en Java.
- ➔ Object est l’équivalent du design pattern “Singleton”
- ➔ pas d'Héritage Multiple MAIS...
- ➔ Trait

Concept introduit par Scala: les traits

- Un trait contient des définitions de méthode et de champs
- Une classe peut avoir un nombre quelconque de traits

Trait = Interface?

Oui & Non

- ➔ OUI
  - ➔ Si on est bloqué sur l'esprit java, aucun concepts ajouté on va rester sur les limitations des interfaces en Java.
  - ➔ en limitant leur rôle à la définition des contrats des classes
- ➔ NON
  - ➔ les Traits **peuvent implémenter des méthodes et peuvent avoir des champs** .
  - ➔ Les Traits sont donc comme des interfaces avec une implémentation partielle
  - ➔ Les Traits en Scala sont empilables
  - ➔ la possibilité d'étendre des interfaces
  - ➔ La notion de Trait est inspiré des Mix-In du langage Ruby.

Trait = Class Abstract?

- ➔ NON...
- ➔ Une classe ne peut étendre qu'une seule super-classe, et donc une seule classe abstraite
- ➔ les classes abstraites établissent une relation classique "est-un" orientée objet tandis que les traits sont une manière de composer.
- ➔ Les classes abstraites peuvent avoir des paramètres de constructeur ainsi que des paramètres de type.
- ➔ Les traits ne peuvent avoir que des paramètres de type.
- ➔ Si vous voulez composer plusieurs classes, la méthode Scala consiste à utiliser **la composition de classe mixin**
- ➔ Les restrictions de traits par rapport aux classes sont introduites pour éviter **les problèmes typiques d'héritage multiple**

La composition mixin avec plusieurs traits

Exemple :

```
class A { def n = ... }
trait C { def m = ... }
trait D { def k = ... }
class B extends A with C with D {}
```

Est (à peu près) équivalent à :

```
class A { def n = ... }
class B extends A {
def m = ...
def k = ...
}
```

C’est tout…

Composition Mixin

```
1  abstract class A {
2    val message: String
3  }
4  class B extends A {
5    val message = "Je suis un B"
6  }
7  trait C extends A {
8    def loudMessage = message.toUpperCase()
9  }
0  class D extends B with C
1
2  val d = new D
3  println(d.message) // Je suis un B
4  println(d.loudMessage) // JE SUIS UN B
5
```

Composition Mixin

```
73  abstract class Bird {
74      def flyMessage: String
75      def fly() = println(flyMessage)
76      def swim() = println("I'm swimming")
77  }
78
79  class Pigeon extends Bird {
80      val flyMessage = "I'm a good flyer"
81  }
82
83  class Hawk extends Bird {
84      val flyMessage = "I'm an excellent flyer"
85  }
86
87  trait Swimming {
88      def swim() = println("I'm swimming")
89  }
90
91  class Duck extends Bird with Flying {
92      val flyMessage = "I'm an excellent flyer"
93  }
```

Conclusion sur les traits

- peuvent être utilisés comme des interfaces & classes abstraites

Mais pas seulement:

- mixin: trait pour changer une classe
- permettent héritage multiple
- avec l’héritage, nous ne pourrions pas atteindre ce niveau de flexibilité

Akka pour gérer la concurrence

C’est quoi Akka

librairie open source

utilisable avec Scala, Java

Pourquoi utiliser Akka

Faciliter la programmation des algorithmes distribués

problème lié à la concurrence: mémoire partagé, etc

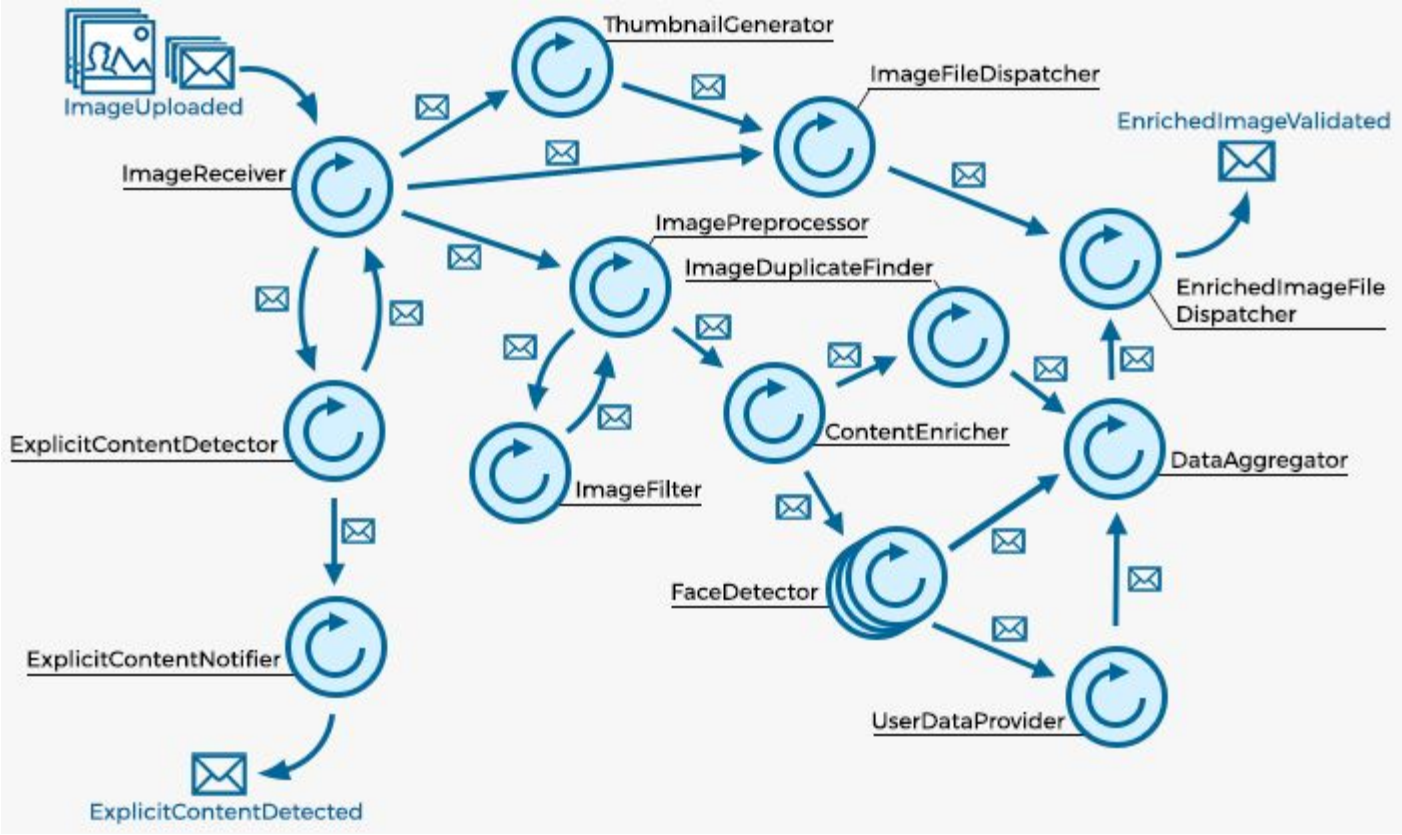
pour le déploiement sur le réseau, problème de gestion de réseau: connexion, etc

Comment Akka travaille

paradigme de programmation avec niveau abstraction de haut niveau

⇒ problème de algorithme distribué transparent pour le programmeur

Modèle acteur


ImageUploaded
ThumbnailGenerator
ImageFileDispatcher
EnrichedImageValidated
ImageReceiver
ImagePreprocessor
ImageDuplicateFinder
EnrichedImageFile
Dispatcher
ExplicitContentDetector
ContentEnricher
DataAggregator
ImageFilter
FaceDetector
ExplicitContentNotifier
UserDataProvider
ExplicitContentDetected

Quizz Simple Scala

donnez le plus petit type des expressions suivantes :

if(1==0) 1 else "toto"

if (1==0) 0

if(1==1) println(1)

Quizz Simple Scala

Les programmes suivant est-il valide ?

1- val t= new Array[Any](10)

t(1)="toto" ; t(1)=189

2- val t= Array("zero","un","deux")

t(1)=189

Quizz Simple Scala

Les programmes suivant est-il valide ?

1- val li= List("zero","un","deux")

li(1)="one"

2- val li= List(1,"toto",2)

val l2= li ++ List(3,4)

Quizz Simple Scala

Ces definitions de fonction sont-elles valides ?

```
def plus(x:Int,y:Int):Unit={x+y; println("x+y")}

def plus(x,y):Int={x+y}

def plus(x:Int,y:Int)=x+y
```

Quizz Simple Scala

Ces programmes sont-ils corrects ?

1- class T(x:Int){val y=x}

val t= new T(10) ;println(t.y)

2- class T(x:Int){val y=x}

val t= new T(10) ; t.y=20

Quizz Akka

Pourquoi le modèle d’acteur est plus efficace en execution par rapport au programmation concurrent avec exclusion mutuel?

Comment sont résolu les problèmes de concurrence dans Akka?

Quels sont les différences entre le modèle POO et le modèle d’acteur?