Traîtement distribué des données

Master 1 ISD

Données massives

Scala et Spark

Kim.Nguyen@lri.fr

Plan

Scala

- Scala : *Scalable Language*. Les développeurs ajoutent des fonctionnalités suivant les demandes des utilisateurs
- Orienté objet : classes, méthodes, héritage
- Fonctionnel : (fonction de première classe, pattern-matching, collections immuables, ...)
- Inter-opérable avec Java (compile vers du byte-code Java, s'exécute sur la JVM, bibliothèque standard Java librement utilisable depuis Scala)
- Développé à l'EPFL par Martin Öderski
- Langage de développement de Spark et principal langage de manipulation

Hello, World !

```
//Fichier HelloWorld.scala
 object HelloWorld {
     def main(args: Array[String])  {
          println("Bonjour tout le monde !")
     }
}
```

Compilation et exécution :

```
$ scalac HelloWord.scala

$ scala HelloWorld

Bonjour tout le monde !
```

Hello, world dans le toplevel

```
$ scala
Welcome to Scala 2.11.11 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_144).
Type in expressions for evaluation. Or try :help.
scala> object HelloWorld {
  | def main (args : Array[String]) {
  |     println("Bonjour tout le monde !")
  | }
  | }

defined object HelloWorld

scala> HelloWorld.main(new Array[String](0))
Bonjour tout le monde !

scala> HelloWorld.main(new Array(0))
Bonjour tout le monde !

scala> HelloWorld.main(Array("foo", "bar"))
Bonjour tout le monde !

scala> HelloWorld.main(Array.empty)
Bonjour tout le monde !
```

Description du langage

Que peut-on dire sur ce code ?

- Syntaxe similaire à Java (parenthèse pour les arguments, accolades pour les blocs, chaînes avec des ")
- Pas de point-virgule
- Types génériques ? (`Array[String]`)
- Inférences de types ? (pas de type de retour pour main, pa s de `String` pour `new Array(0)`)

Au programme

- Bases du langage
- Fonctions
- Collections
- Classes et objets

C'est juste assez pour faire le TP (et éveiller votre curiosité)

Variables en Scala

Deux modes de définitions de variables :

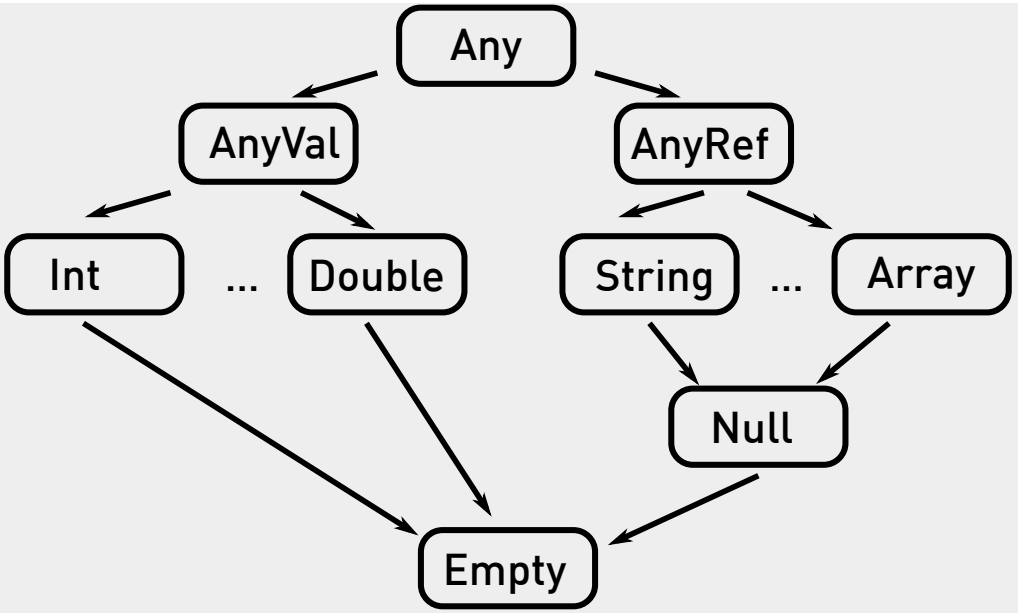
- Définition de valeur (mot clé `val`) ⇒ immuable (similaire à `static` de Java ou `let` d'OCaml)
- Définition de variable (mot clé `var`) ⇒ modifiable

```
var x : Int = 0
val y : Int = 0
x = x + 1        //affiche x: Int = 1
y = y + 1        //erreur : reassignment to val

//On peut omettre les types
val s = "Bonjour"   // affiche s : String = "Bonjour"
val v = 42          // affiche v : Int = 42

//On peut omettre les valeurs, mais il faut donner le type

val u : Int
val z : Double
```

Types

`Boolean` : booléen `true` et `false`
`Byte` : entier signés 8 bits, pas de littéraux
`Short` : entier signés 16 bits, pas de littéraux
`Char` : caractères UTF-16 (comme en Java), 'A', 'B', ...
`Int` : entiers signés 32 bits : 1, 2, -42, ...
`Long` : entiers signés 64 bits : 1l, 2l, -42l, ...
`Float` : Flottants IEEE-754 simple précision : 3.14e10f
`Double` : Flottants IEEE-754 double précision : 3.14e10
`String` : Chaînes de caractères
`Unit` : type des expressions ne produisant pas de résultats (≡ `void`) : `()`
`Any` : super type le plus haut (comme Object mais en plus propre)
`AnyVal` : super type de tous les types de bases
`AnyRef` : super type de tous les types d'objects (≡ Object)
`Null` : Type du pointeur null : `null`
`Empty` : type vide (aucune valeur ne peut avoir le type vide)

Hierarchie des types



- Tous les types sont présents dans la hierarchie
- Pas de distinction arbitraire entre types scalaires et objets
- (les types génériques c'est une autre histoire)

Structures de contrôle : blocs

```
{
    e1
    e2
     …
    en
}
```

- Évalue e_1, ..., e_n tour à tour
- Renvoie comme valeur la valeur de e_n
- Si n vaut 0 (bloc vide) le type de retour est `Unit`
- Si les e_i sont purs (sauf le dernier), le compilateur affiche un *warning*

Structures de contrôle : if

```
if (e)
    b1
else
    b2
```

- e doit avoir le type Boolean
- Le type de l'expression est le plus petit super type commun aux types des deux branches
- Le else peut être absent et il a alors le type Unit

Structures de contrôle : `while` et `for`

```
while (e) {
  …
}
```

- Type de la boucle est `Unit`
- `e` doit être de type `Boolean`

```
for (x <- e if c1; … ; if cn)
     b
```

- `e` doit être une Collection
- la boucle est effectuée en assignant `x` tour à tour à chaque éléments de `e`
- le corps de la boucle n'est exécuté que si tous les c_i s'evaluent à `true`

```
for (x1 <- e; …; xn <- en )
     b
      //est équivalent à
for (x1 <- e1)
        …
     for (xn <- en)
         b
```

Fonctions : syntaxe

```
def functionName(x1: T1, …,  xn:Tn) : S = {
…
}

//appel
functionName(v1, …, vn)
```

- On peut définir des fonctions où l'on veut (pas forcément dans une classe)
- On peut ne pas indiquer le type de retour (et le compilateur essaye de l'inférer)
- Si la fonction n'a pas de paramètre on peut omettre les parenthèse vides.

Fonctions : imbriquées

```
def fibonacci(n : Int) = {

    def fibo_aux(i : Int, accu1 : Int, accu2 : Int) : Int = {
        if (i > n) {
           accu1
      }       else {
              fibo_aux (i+1, accu1+accu2, accu1) }
    }
   fibo_aux(0, 1, 0)

}
```

- La fonction interne a accès au variables de la fonction englobante (cloture)
- On doit indiquer le type de retour pour les fonction récursives
- Les fonction imbriquées ne sont pas visibles depuis l'extérieur

Fonctions anonymes, ordre supérieur et généricité

```
val next = (x : Int) => x + 1

def apply (f : Int => Int, v : Int) = {   f (v)  }

def ident[A]( x : A) = { x }

apply(next, 3)
apply(ident, "Hello")
apply(ident, 42)
```

Encore plein d'autres choses (fonctions variadiques, arguments optionnels, application partielle, appel paraisseux, ...).

Collections

De nombreuses classes de collections en Scala :

`Array :`	les tableaux mutables de taille fixe homogènes
`List :`	les listes immuables homogènes
`Set :`	les ensembles mutables ou immuables homogènes
`Map :`	les dictionnaires mutables ou immuables homogènes
`Option :`	le type optionnel immuable

Tableaux

```
val tab = new Array[String](4) //initialisé à null dans toutes les cases
tab(0) = "a"
tab(1) = "b"
tab(2) = "c"
println(tab(2))

val tab = Array("b", "c", "d", "e", "f")
tab.length
```

Pas besoin de mettre le type des éléments s'il peut être inféré

Listes

Listes fonctionnelles imuables à la OCaml avec pattern matching !

```
val l = 1 :: 2 :: 3 :: Nil
val l1 = List(1,2,3)    // equivalent

val l2 = 0 :: l1        // l1 n'est pas modifiée
val l3 = l1 :+ 4        // ajout en fin, l1 n'est pas modifiée

val l4 = l2 ::: l3      // concaténation

def length[A](l : List[A]) : Int = {
       l match {
        case Nil => 0
        case _ :: ll => 1 + length(ll)
}
length(l4)
```

Set

```
val s1 = Set(1,2,3,4,5)   //immuable, package ouvert par défaut
val s2 = scala.collection.mutable.Set(1,2,3,4,5) // mutable

val s3 = s1 + 6           // ajout
val s4 = s1 - 2           // suppression

s4.contains(3)            // test

val s5 = s2 - 1           // suppression pas d'effet
s2.remove(1)              // effet de bord, s2 contient 2,3,4,5
s2.add(8)                 // effet de bord s2 contient 2,3,4,5,8
```

Les méthodes `.add/.remove` renvoie un Boolean valant `true` si l'ensemble a été modifié

Map

```
val m1 = Map("A" -> 1, "B" -> 10, "C" -> 30)
val m2 = scala.collection.mutable.Map("A" -> 1, "B" -> 10, "C" -> 30)

m1("B")   // renvoie 10
m1("F")   // Exception levée

m1.get("B") // renvoie Some(10)
m1.get("F") // renvoie None

m1 + ("D" -> 50)  // renvoie une nouvelle map
m1 - "A"          // renvoie une nouvelle map

m2("A") = 10      //mise à jour
```

remarque : e_1 `->` e_2 **est juste une notation pour** `(`e_1`,` e_2`)`

Type option

- Type Option[A] : types des valeurs de type A qui sont potentiellement absentes
- Java utilise `null` qui habite tous les types Objet
- Scala autorise `null` pour communiquer avec Java
- Dans du code Scala pur, on utilisera toujours Option

```
val m = Map("A" -> 10, "B" -> 20)

m.get("A").isDefined        // true
m.get("A")                  // Some(10)
m.get("C").isDefined        // false

m.get("C").orElse(50)       // renvoie la valeur contenue dans
                            // le Some ou 50 si l'argument est None

m.get("C") match {
  case None => 50
  case Some(x) => x
}
```

Itérateurs de collection

`.map[A,B](f : A => B)` : applique `f` tour à tour à chaque élément de la collection et renvoie la collection transformée

`.forEach[A](f : A => Unit)` : applique `f` tour à tour à chaque élément de la collection et renvoie Unit

`.filter[A](f : A => Boolean)` : applique `f` tour à tour à chaque élément de la collection et renvoie la collection des éléments pour lesquels f renvoie `true`

`.foldLeft[A,B](x : B) (f : B => A => B)` : cf. List.fold_left du cours précédent.

`.flatten()` : Retire un niveau d'imbrication de collection

`.flatMap(f: A => List[B])` : applique `f` tour à tour à chaque élément de la collection et renvoie la concatenation de toutes les listes résultantes

`.to(n)` : Énumère tous les éléments entre l'objet courant et `n`.

Exemple: `1.to(10)`

Classes et objets

En Scala toutes les valeurs sont des objets. Le compilateur et la JVM se chargent automatiquement de convertir en types de bases si besoin

```
1.toString              // "1"
1.to(10)                // Range(1,2,3,4,5,6,7,8,9,10)
'A'.range('Z')
val f = (x : Int) => x + 1
val g = f.andThen(f)
g (1)                   // 3
"1234".toInt            // 1234
```

Classes : constructeurs et propriétés

```
class Shape(cx : Int, cy : Int) {

    val x = cx
    val y = cy

    override def toString() = {
            "(" + x + ", " + y + ")"
    }

    def this() = {
          this(0,0)
    }
 }
```

- `cx` et `cy` sont les arguments du constructeur
- On peut ajouter d'autres constructeurs en surchargeant une méthode `this`
- Pour redéfinir une méthode on doit explicitement ajouter `override`
- Les attributs `x` et `y` sont publics et non modifiables (car `val`)

Classes : Héritage

```
class Circle(cx : Int, cy : Int, cr : Int) extends Shape(cx, cy) {

    private var theRadius = Math.max(cr,0)
      def radius = theRadius
      def radius_=(newRadius : Int) = {

      if (newRadius >= 0)
           theRadius = newRadius

     }
    }

    val c = new Circle(0,0, 10)
    c.radius                        // appelle c.radius 10
    c.radius = 3                    // appelle c.radius_=(3)
```

Interface Java

On a accès aux types de la bibliothèque standard Java :

```
    class Foo  (value : Int) extends java.lang.Comparable[Foo] {
     val v = value

    def compareTo(other : Foo) = {
            if (v < other.v) -1
            else if (v > other.v) 1
            else 0
    }
}
```

Objet singletons

```
object Bar {
   val x = 0
  val y = 3
  def doSomething() { … }

}



Bar.doSomething()
Bar.x
Bar.y
```

Objet qui sont les seuls habitants de leur classe

Usuellement utilisé pour y stocker des constantes et méhodes « statiques »

C'est dans un objet singleton qu'on met le "main" du programme

… et tout le reste

- Définition de pattern-matching pour les classes
- surcharge des opérateurs
- Traits (≡ interfaces)
- …

Plan

Appache Spark

Framework de calcul distribué

- ne définie pas de stockage de donnée (peut réutiliser des bases de données, des fichiers textes, HDFS, ...)
- vient combler des lacunes de Map/Reduce
- supporte nativement plusieurs langages (Scala, Python et R)
- Est implémenté en Scala
- Fournit en plus en standard un moteur SQL et des bibliothèques de machine learning

On s'intéresse ici à l'API « core » qui se place au même niveau que Map/Reduce. Les autres composants (SQL, Streaming, Machine Learning) sont construits au dessus.

Problèmes de Map/Reduce (1)

Problèmes d'interface avec le programmeur :

- API extrèmement bas-niveau (context.write)
- Pas de sûreté de typage
- Code ultra verbeux
- Réutilisation du code difficile

Problèmes de Map/Reduce (2)

Problèmes de performances :

- Orienté disque : mauvaise utilisation de la mémoire
- Pas de partage possible de sous-tâches au niveau de l'API

Supposons une transformation qui s'exécute comme une opération M suivie de R_1 et une autre qui s'éxécute comme M suivi de R_2

Si on veut les 2 résultats, on va calculer 2 fois M

Si on ne veut pas calculer 2 fois M :

1. On calcule M et on sauve le résultat `m` dans HDFS
2. On charge `m` avec un Map identité et on envoie à R_1
3. On charge `m` avec un Map identité et on envoie à R_2
4. Si on veut combiner les résultats de R_1 et R_2 il faut refaire un map et un reduce...

Resilient Data Set

Un Resilient Data Set (ensemble de données persistent) est une abstraction de haut niveau qui représente un calcul (et non pas son résultat) sur des données

Les RDDs sont à la base des transformations Spark

Le chargement des données crées un nouveau RDD (les données ne sont pas chargée, on crée juste une structure qui, quand elle sera évaluée chargera les données

On peut composer ds RDDs au moyen de transformations

On peut exécuter une action sur un RDD. Cela déclanche le calcul de toute la transformation pour obtenir un résultat final

Transformations de RDD

Les fonctions de transformation sont les itérateurs Scala, plus quelques nouveaux:

- `.map(f)`
- `.filter(f)`
- `.flatMap(f)c`
- `.union(otherDataSet)`
- `.intersection(otherDataSet)`
- `.join(otherDataSet)` sur un RDD de paires (K,V) et un autre RDD de paires (K, W) renvoie les triplets (K,(V,W))
- `.distinct()`
- `.groupByKey()` (s'applique sur un RDD de paires (Clé,Valeur)
- `.reduceByKey(f)` (s'applique sur un RDD de paires (Clé,Valeur). Exécute un fold de f sur le tableau de toutes les valeurs de la même clé.
- ...

Action sur les RDDs

Les actions exécutent le RDD auquels elles sont appliquées pour renvoyer un résultat:

- `.reduce(f)`
- `.foreach(f)`
- `.collect()` retransforme le RDD en collection Scala
- `.count()`
- `.take(n)`
- `.first()`

Spark en pratique

Spark est fourni avec des interpréteurs standards pour Scala, Python (2.7) et R. Ce sont les interpréteurs normaux, dans lesquels sont préchargés les bibliothèques pour Spark.

Pour exécuter un programme Scala, on peut aussi l'exporter comme un jar et utiliser la commande `spark-submit` pour l'exécuter.

Démo