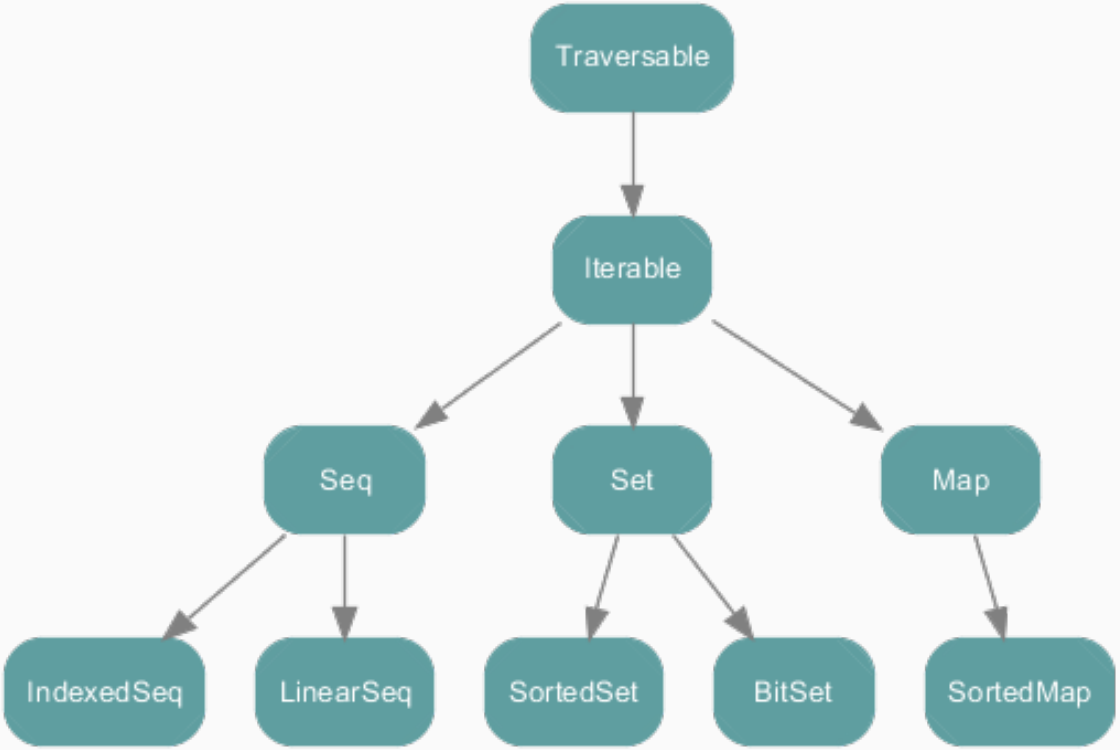
Collections Scala

Jean-Luc Falcone

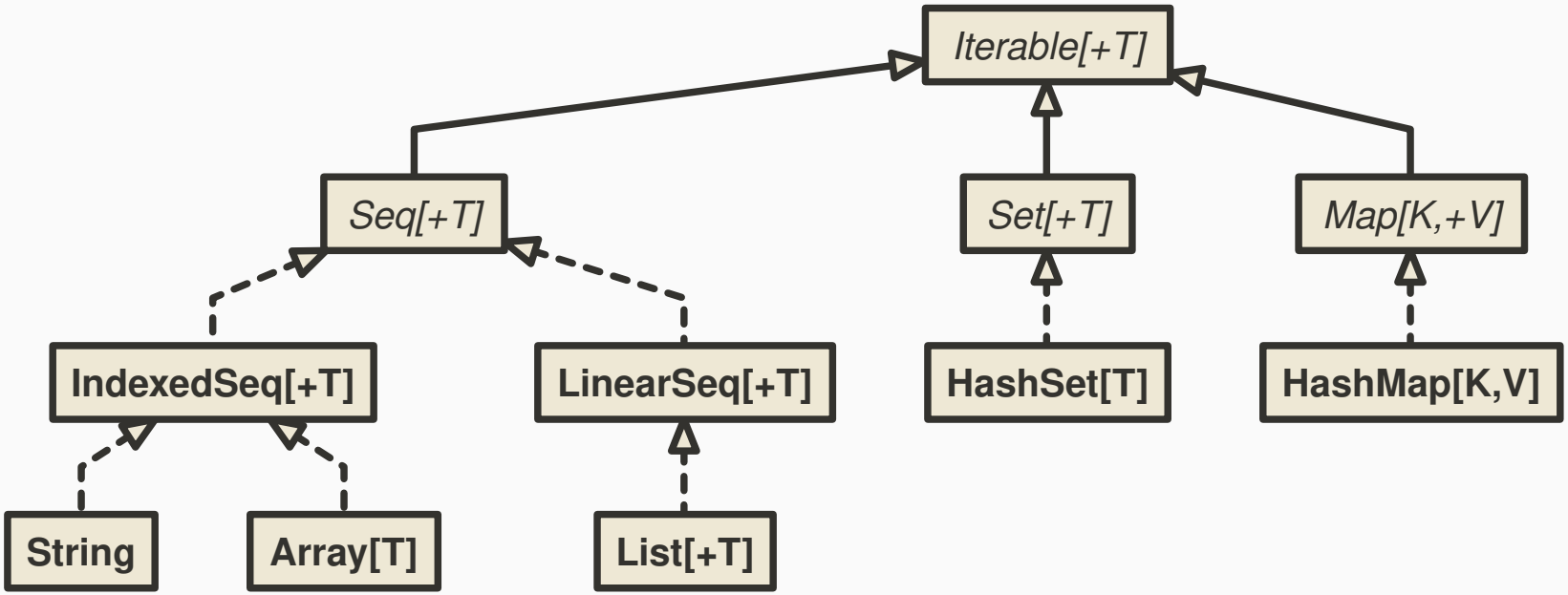
23 Avril 2023

Hierarchie

Collections (<= 2.12)



Collections (>= 2.13)



Types principaux

- `Seq`: Séquences
- `Set`: Ensembles
- `Map`: Tableaux associatifs, dictionnaires

Mutabilité

- `scala.collections.immutable`
  - `List`
  - `Vector`
  - `Set`
  - `Map`
- `scala.collections.mutable`
  - `ArrayBuffer`
  - `Set`
  - `Map`

Séquences: `Seq`

Séquence d'éléments où l'ordre est conservé. Deux sortes:

- `LinearSeq`: optimisé pour parcours
  - `List` (immutable)
  - `LazyList` (immutable)
- `IndexedSeq`: optimisé pour accès aléatoire
  - `mutable.ArrayBuffer`
  - `Vector` (im.)
  - `Array` (mutable)
  - `String` (im.)

Exemple 1: `List`

```scala
val l1 = List( 1, 2, 3 )
val l2 = 1 :: 2 :: 3 :: Nil

val e1: List[Int] = Nil
val e2 = List.empty[Int]
val e3 = List[Int]()
```

Opérations de base

```scala
val l = List( 1, 2, 3 )

l.isEmpty //false
l.size    //3

l.head    //1
l.tail    //List(2,3)

val l2 = 0 :: l //List(0,1,2,3)
val l3 = l2 ++ List(4,5) //List(0,1,2,3,4,5)
```

Pattern matching

```
list match {
  case Nil => ""
  case a :: Nil => s"$a"
  case a :: b :: Nil => s"$a $b"
  case a :: bs => s"$a $bs"
}
```

Pattern matching (exemple)

```
def max( is: List[Int] ): Int = {

  def maxRec( rem: List[Int], max: Int ): Int =
    rem match {
      case Nil => max
      case i :: rest if i>max => maxRec( rest,i )
      case _ :: rest => maxRec( rest, max )
    }

  maxRec( is.tail, is.head )
}
```

Exemple 2: Vector

```
val v = Vector(2,4,6,8)
val v2 = v :+ 10 // Vector(2,4,6,8,10)
val v3 = 0 +: v2 // Vector(0,2,4,6,8,10)
val v4 = v ++ v  // Vector(2,4,6,8,2,4,6,8)

v.size // 4
v(2)   // 6
v.updated(3,11) //Vector(2,4,6,11)
v.head // 2
v.tail // Vector(4,6,8)
```

Opérateurs associatifs à droite

- Certains opérateurs qui se terminent par un point virgule
  - Par exemple: `::` ou bien `+:`

- Le destinataire (`this`) de la méthodeà droite
- Ces opérateurs sont associés également à droite

```
1 :: Nil == Nil.::(1)
```

```
1 +: 2 +: Vector(3) == 1 +: ( 2 +: Vector(3) )
                    == Vector(3).+:(2).+:(1)
```

Ensembles: `Set`

Chaque élément est unique. Plusieurs sortes

- Classiques:
  - `Set` (im.)
  - `mutable.HashSet`
- `SortedSet`: trie les éléments selon leur ordre (`Ordering`):
  - `immutable.SortedSet`
  - `mutable.SortedSet`
- `BitSet`: optimisé pour stocker des entiers
  - Existe en mutable ou immutable

Exemple: Set

```
val s = Set( "A", "B" ) //type: Set[String]
val s2 = s + "I"                  //Set("A","B","I")
val s3 = s ++ Set( "X", "Y" ) //Set("X","A","B","Y","I")

s3.size // 5
s3( "Y" ) //true
s3( "F" ) //false

s3 - "I" //Set("X","A","B","Y")
s3 -- s //Set("X","Y","I")
s3 & Set("A","B","C") //Set("A","B")
```

Tableaux associatifs: `Map`

Associe une valeur à chaque clé. Les clés forment un ensemble.

- Par défaut `Map` est immutable, mais il existe des variantes mutables (`mutable.HashMap`).
- Il existe plusieurs variantes. Par exemple `LongMap` est optimisé pour des clés qui sont des `Long`.

Exemple: Map

```
val m = Map( "A"->4, "B"->8 )
                        //type: Map[String,Int]
val m2 = m + ("D"->2)
val m3 = m2 ++ Map( "E"->0, "A"->1 )
val m4 = m3 - "B"
                  //Map( "A"->1, "D"->2, "E"->0 )

m4.size // 3
m4.contains( "A" ) //true
m4( "A" ) //4
m4( "F" )
 //!!! NoSuchElementException: key not found: F

m4.getOrElse( "F", 0 ) // 0
```

Conversions

```
Map( "A"->4, "B"->8 ).toList
                // List( ("A",4), ("B",8) )

Set( 2, 5, 10 ).toArray // Array(2,5,10)

List( 1, 1, 2, 3, 3, 3 ).toSet // Set(1,2,3)

List( (4,"D"), (6,"F") ).toMap
                // Map(4 -> "D", 6 -> "F")
```

Intervales

Les intervales sont des séquences d'entiers:

```
val a = 1 to 3
val b = 0 until 3

a.toList // List(1, 2, 3)

b.toSet //Set(0, 1, 2)

a.size // 4

a.head // 1
a.tail // Range(2, 3)
```

String

Les chaînes de caractères sont des séquences de caractères:

```
val s = "hello"

s.size // 5

s.toList // List('h', 'e', 'l', 'l', 'o')

(s :+ ' ' ) ++ "world" //"hello world"

s(1) // 'e'
```

Arrays

Les tableaux Java sont aussi des collections:

```
def average( ary: Array[Int] ): Double = {
  val n = ary.size
  var i = 0
  var sum = 0
  while( i < n ) {
    sum += ary(i)
    i += 1
  }
  sum.toDouble / n
}
```

Lambda !

Effet de bord pour chaque élément `foreach`

On peut facilement appliquer un effet de bord à chaque élément avec la méthode `foreach`:

```
List("hello", "world" ).foreach( println )

var s = 0
Set(1,4,8,3).foreach{ i =>
  s += i
}


Map( "A"->2, "B"->4 ).foreach { (k,v) =>
  println( s"Key $k, Value $v" )
}
```

Equivalent `for`

On peut utiliser une expression `for` qui sera traduite en `foreach`:

```
for( w <- List("hello", "world" ) ) {
  println( w )
}

var s = 0
for( i <- Set(1,4,8,3) ) {s += i}

for {
  (k,v) <- Map( "A"->2, "B"->4 )
} println( s"Key $k, Value $v" )
```

Boucle `for`

En utilisant l'expression `for` sur un intervale on peut simuler une boucle `for` comme en Java:

```
val ary: Array[Double] = ...
var sum = 0.0

for( i <- 0 to ary.size ) { //Attention BUG ! Lequel ?
  sum += ary(i)
}
```

Attention aux performances
Préférer une boucle while, une méthode récursive ou utiliser `foldLeft`.

Signature: `foreach`

```
trait CC[A] {
  def foreach( f: A=>Unit ): Unit
}
```

Modifier chaque élément `map`

On peut modifier chaque élément d'une collection avec la méthode `map`:

```
List("hello", "world" ).map( _.toUpperCase )

Set(1,4,8,3).map{ i =>
  1.0 / i
}

Map( "A"->2, "B"->4 ).map {
  (k,v) => k -> (v/6.0)
}
```

Equivalent `for... yield...`

On peut utiliser une expression `for...yield...` qui sera traduite en `map`:

```
for {
  l <- List("hello", "world" )
} yield l.toUpperCase

for( i <- Set(1,4,8,3) ) yield (1.0/i)

for{
  (k,v) <- Map( "A"->2, "B"->4 )
} yield ( k -> v/6.0 )
```

Signature: `map`

```
trait CC[A] {
  def foreach( f: A=>Unit ): Unit
  def map[B]( f: A=>B ): CC[B]
}
```

Filtrer chaque élément `filter`

```
List("hello", "world" ).filter( _.startsWith("w") )

Set(1,4,8,3).filter{ i =>
  i % 2 == 1
}

Map( "A"->2, "B"->4 ).filter{
  (k,v) => v > 3
}
```

Equivalent `for... if ...yield...`

On peut utiliser une expression `for...if yield...` qui sera traduite en `filter`:

```
for {
  l <- List("hello", "world" )
  if l.startsWith("w")
} yield l

for {
  i <- Set(1,4,8,3) if i % 2 == 1
} yield i
```

Signature: `filter`

```
trait CC[A] {
  def foreach( f: A=>Unit ): Unit
  def map[B]( f: A=>B ): CC[B]
  def filter( f: A=>Boolean ): CC[A]
}
```

Remplacer un élément par plusieurs `flatMap`

La méthode `flatMap` permet de remplacer chaque élément par une nouvelle collection.

```
List("hello", "world" ).flatMap{ w =>
  List( w, translate(w, "FR"), translate(w, "DE") )
}

//List( "hello", "bonjour", "hallo",
//                          "world", "monde", "Welt" )

Set(1,4,8,3).flatMap{ i =>
  if( i % 2 == 1 ) Set( i, -i )
  else Set[Int]()
}
//Set(1, -1, 3, -3)
```

Exemple

```
def released( a: Artist ): List[Album]
def content( a: Album ): List[Track]

val jb = Artist( "Justin Bieber" )

released( jb ).flatMap( content )

released( jb ).flatMap { album =>
  content( album ).map( _.title )
}
```

Expressions chainées (1)

```
for {
  album <- released( jb )
  song <- content( album )
} yield song.title
```

Expressions chainées (2)

```
val isAdult = (u:User).age >= 18
users
  .filter( isAdult )
  .flatMap{ u =>
    val fs = u.friends.filter( isAdult )
    fs.map( _.emailAddress )
  }

for {
  u <- users if isAdult(u)
  f <- u.friends if isAdult(f)
} yield f.emailAddress
```

Signature: `flatMap`

```
trait CC[A] {
  def foreach( f: A=>Unit ): Unit
  def map[B]( f: A=>B ): CC[B]
  def filter( f: A=>Boolean ): CC[A]
  def flatMap[B]( f: A=>CC[B] ): CC[B]
}
```

Les collections sont aussi des fonctions !

- `IndexedSeq[A]` est une fonction `(Int)=>A`
- `Set[A]` est une fonction `(A)=>Boolean`
- `Map[K,V]` est une fonction `(K)=>V`

```
val a = Array( "yes", "yes", "no", "no", "yes" )
val s = Set( 1, 3, 4 )
val m = Map( "yes" -> "OUI", "no" -> "NON" )

(2 to 8).filter( s )
        .map( a andThen m )
        .foreach(println)
//NON
//OUI
```