Master 2 EIDD

Langages et Environnements Evolués

Tutoriel

Par :

BA Fatou

Les monades en Scala

Lien du projet Github : https://github.com/bafatou/scala-monades

Introduction :

Bienvenue dans ce tutoriel ! Nous verrons ce qu'est une Monade et comment créer une nouvelle monade en scala. Si vous vous intéressez à la programmation fonctionnelle, vous avez sûrement déjà entendu parler des monades et peut-être les avez-vous utilisées sans le savoir.

Définition d'une monade

En théorie des langages fonctionnels typés, une **monade** est une structure permettant de manipuler des langages fonctionnels purs avec des traits impératifs. Un type monadique est un type de donnée répondant à certaines lois et généralement caractérisé dans Scala par la présence des deux **méthodes unit(aussi appelé return) et flatMap(aussi appelé bind)**, que nous aborderons ici même.
Une définition de compréhension : Vous pouvez considérer les monades comme des **emballages** . Vous prenez un objet et l'envelopper avec une monade. C'est comme emballer un cadeau.

Scala ne vient pas avec un type de monade intégré comme Haskell, nous allons donc modéliser la monade nous-mêmes. Si vous jetez un coup d'œil à quelques bibliothèques de programmation fonctionnelles comme Scalaz, vous y trouverez des monades.

Installation des outils

Pour suivre ce tutoriel il faut avoir installer les outils nécessaires : le JDK (Java Development Kit) et SBT (Simple Build Tool) qui est un outil de build utilisé par le langage Scala et Java. Nous travaillerons avec l'IDE Eclipse

Pour commencer téléchargez le JDK (Java Development Kit) sur le site de Oracle :https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html
Cliquez sur « Accept License Agreement » ensuite téléchargez le JDK en fonction de votre système d'exploitation, dans mon cas je travail sur Windows. Une fois le téléchargement terminé exécutez le fichier pour lancer l'installation.
Nous allons à présent installer l'IDE Eclipse, cliquez sur le lien suivant pour le téléchargement :https://www.eclipse.org/downloads/ nous allons télécharger « Eclipse IDE for java Developers » une fois le téléchargement terminé exécutez le fichier pour lancer l'installation.
Bien ! maintenant nous allons télécharger SBT sur le lien suivant : https://www.scala-lang.org/download/ cliquez sur « Download SBT » une fois le fichier .exc téléchargé, exécuter sur votre terminal (command line) : sbt

Ainsi sbt va essayer de télecharger toutes les dépendances depuis internet.

Ouvrez l'application Eclipse cliquez sur ''help'' ensuite sélectionnez ''Eclipse Marketplace'', une nouvelle fenêtre s'ouvre à vous. Dans la rubrique ''Search'' tapez « scala » et cliquez sur « go »,

cliquez ensuite sur « Install » à droite de « Scala IDE 4.7.x ». une fois le téléchargement terminé suivez les instructions pour finaliser l'installation des plugin. Bien !

Toujours dans Eclipse essayons de créer à présent un nouveau projet Scala suivez les étapes suivantes : « file – New – Scala Project » nommez votre projet comme vous le voulez. Dans notre nouveau projet scala, sur ''src'',faites un clic droit ensuite : « New – Scala Object » que nous nommerons « Demonstration».

Création de monade en scala :

Nous allons modéliser une monade avec un trait générique qui fournit les méthodes unit () et flatMap() *(trait est l'équivalent des Interfaces en Java)*. Nous l'appelons simplement M au lieu de Monad.

```
trait M[A] {
  def flatMap[B](f: A => M[B]): M[B]
  def unit[A](x: A): M[A]
}
```

Les monades prennent un paramètre de type. Nous n'avons pas simplement écrit M; nous avons écrit M[A]. Si nous voulons envelopper un objet avec une monade nous devons paramétrer la monade avec le type de l'objet sous-jacent, par exemple M[Int], M[String], M[MyClass], etc.

Des exemples de monades en scala :
• List est une monade où unit(x) = List(x)
• Option est une monade où unit(x) = Some(x)
• Set est une monade où unit(x) = Set(x)
• Future(x) est une monade où unit(x) = Future(x)

Les méthodes unit de ces monades ont des noms qui diffèrent, cependant elles implémentent toutes flatMap en conservant ce nom.

Lois des monades

Pour pouvoir être considéré comme une monade, un type doit satisfaire 3 lois :

Associativité

La fonction flatMap doit pouvoir être enchainée quelque soit le placement des parenthèses :

```
m flatMap f flatMap g == m flatMap (x => f(x) flatMap g)
```

Il est plus simple d'illustrer la notion d'associativité par l'exemple suivant : x, y et z des Integers :

```
x + (y + z) == (x + y) + z
```

Composition neutre par return à gauche

Cette loi spécifie que si l'on appelle flatMap sur unit(x) en appliquant f, on doit obtenir f(x) :

```
unit(x) flatMap f == f(x)
```

Composition neutre par return à droite

La dernière loi dicte que si l'on applique une unit à flatMap d'une monade m le résultat doit être m :

```
m flatMap unit = m
```

Avant d'entrer dans le détail, maintenant voyons quel est le but des deux fonctions (flatMap, Unit).

• **unit :**
La fonction unit prend un élément de type A en paramètre et retourne une instance de la monade M[A] (c'est une sorte de constructeur de monades si on veut)

• **flatMap :**
La fonction flatMap prend un type arbitraire B et une fonction qui mappe un type A vers une monade de B et retourne une monade M appliquée à B.

• **map :**
La méthode map des monades peut être définie en combinant unit et flatMap :

```
m map f == m flatMap (x => unit(f(x)))
```

Voici la signature de la fonction **flatMap :**

```
def flatMap[B](f: (A) => U[B]): U[B]
```

Disons que U est une liste. Cela fonctionne pour divers autres types, mais nous allons utiliser List pour cet exemple. flatMap prend une fonction avec la signature A → List[B] et il utilise cette fonction pour transformer l'objet sous-jacent de type A en une List[B]. Cette opération est appelée*map*. Après avoir transformé notre A sous-jacent en une list[B], cela nous laisse avec une list[List [B]]. Nous n'avons pas utilisé un map() ordinaire, nous avons utilisé flatMap(). Cela signifie que le travail n'est pas encore terminé, flatMap va maintenant "aplatir" notre list[list[B]] en list[B].

Testons un exemple avec le type List :

```
object Demonstration {
  def main(args: Array[String]) {
```

```
     //Si A est un Int
     //definition de notre fonction f
     val f = (i: Int) => List(i - 1, i, i + 1)
     val list = List(5, 6, 7)
     //flatMap d'une liste d'entiers avec f
     println(list.flatMap(f))
     //map d'une liste d'entiers avec f
     println(list.map(f))
   }
}
```

Cela affiche la liste suivante :

```
List(4, 5, 6, 5, 6, 7, 6, 7, 8)
List(List(4, 5, 6), List(5, 6, 7), List(6, 7, 8))
```

Notez que map() prend chaque numéro i et développe en une mini-liste *(prédécesseur, numéro d'origine, successeur)*, ce qui donne la liste suivante de mini-listes : List((4, 5, 6), (5, 6, 7), (6, 7, 8)). FlatMap va plus loin il l'aplatit en une longue liste, ce qui donne une List(4, 5, 6, 5, 6, 7, 6, 7, 8).

Notez que flatMap ne nécessite pas une fonction A → M[A], mais une fonction plus flexible, A → M[B]. Donc, si M est une liste et A est un Int, nous pouvons alimenter le flatMap avec des fonctions telles que Int → List[Chaîne], Int → List[MaClasse], etc. Il n'est pas nécessaire que ce soit Int →Lis[Int]. Nous aurions pu définir f comme :

```
val f = (i: Int) => List("pred=" + (i - 1), "succ=" + (i + 1))
```

Nous pourrions flatMap une liste d’entiers comme ceci:

```
object Demonstration {
    def main(args: Array[String]) {
      //Si A est un Int
      //definition de notre fonction f
      val f = (i: Int) => List ("pred =" + (i - 1), "succ =" + (i + 1))
      val list = List(5, 6, 7)
      //flatMap d'une liste d'entiers avec f
      println(list.flatMap(f))
      //map d'une liste d'entiers avec f
      println(list.map(f))
   }
}
```

Nous obtenons le résultat suivant :

```
List(pred =4, succ =6, pred =5, succ =7, pred =6, succ =8)
List(List(pred =4, succ =6), List(pred =5, succ =7), List(pred =6, succ =8))
```

Le type Option :

A présent explorons l'un des types monadiques les plus souvent utilisés à savoir le type option. Option est une construction qui nous permet d'éviter les pointeurs nuls dans Scala (en Haskell, il s'agit de Maybe). Une Option[A] est un type abstrait générique permettant de caractériser la présence ou l'absence de valeur via deux sous types qui sont Some[A] ou None. Un exemple classique d'utilisation du type Option est le suivant :

```
object Demonstration {
     def main(args: Array[String]) {
        def divide(x:Double, y:Double) = if (y == 0) None else Some(x/y)
        val res1 = divide(4, 2)
        println(res1)
        val res2 = divide(4, 0)
        println(res2)
   }
}
```

Cela affiche le résultat suivant :

```
Some(2.0)
None
```

Le type Option permet de traiter de façon particulière certains cas limite de fonctions pour lesquels aucun résultat n'existe. Un autre exemple d'utilisation de l'Option est la dénotation de l'absence de valeur (absence de donnée dans un formulaire web, représentation d'un champs nullable en base etc.). Dans ce cas, l'absence de valeur est directement traduite par le type de la donnée.
Le type de retour de divide(4, 2) est **Option[Double]**. Ce type a été inféré par le compilateur en fonction des types de x et de y. Notez bien que Option est un **type abstrait** et qu'il ne peut être construit directement mais doit nécessairement passer par un de ses deux sous types **Some et None**.

Nous pouvons transformer les Options en utilisant map

```
object Demonstration {
   def main(args: Array[String]) {
```

```
      val three = Option(3)
      //On aurait pu ecrire
      //val three = Some(3)
      println(three)
      //three: Option[Int] = Some(3)
      val twelve = three map (_ * 4)
      println(twelve)
     // twelve: Option[Int] = Some(12)
    }
 }
```

Notez que l'utilisation de _ est un sucre syntaxique désignant le paramètre de la fonction passée à map.
Nous obtenons le résultat suivant :

```
Some(3)
Some(12)
```

Mais lorsque nous commençons à combiner les résultats de plusieurs calculs susceptibles d'échouer, nous rencontrons des problèmes avec les types.

```
object Demonstration {
    def main(args: Array[String]) {
       val three = Option(3)
       // three: Option[Int] = Some(3)
       val twelve = three map (_ * 4)
       // twelve: Option[Int] = Some(12)
       val four = Option(4)
       // four: Option[Int] = Some(4)
       val twelveB = three map (i => four map (i * _))
       // twelveB: Option[Option[Int]] = Some(Some(12))
       println(twelveB)
     }
 }
```

Cela affiche le résultat suivant :

```
Some(Some(12))
```

Ici, nous nous sommes retrouvés avec une option encapsulée dans une autre option, ce qui n'est pas ce que nous voulons. Mais nous connaissons maintenant la solution, qui consiste à remplacer le dernier **map par flatMap** ou, mieux encore, à utiliser une for-compréhension qui est fondamentalement la syntaxe pour le mapping, le flatMapping et le filtering.

```
object Demostration {
    def main(args: Array[String]) {
        val three = Option(3)
        // three: Option[Int] = Some(3)
        val twelve = three map (_ * 4)
        // twelve: Option[Int] = Some(12)
        val four = Option(4)
        // four: Option[Int] = Some(4)
        val twelveB = three map (i => four map (i * _))
        // twelveB: Option[Option[Int]] = Some(Some(12))
        val twelveC = three flatMap (i => four map (i * _))
        // twelveC: Option[Int] = Some(12)
        println(twelveC)

        //Utilisation de for-comprehension
        val twelveD = for {
        i <- three
        j <- four
      } yield (i * j)
     // twelveD: Option[Int] = Some(12)
       println(twelveD)

  }
}
```

Nous obtenons le résultat souhaité :

```
Some(12)
Some(12)
```

Utilisation des fonctions isDefined et isEmpty du type Option :

Ces deux fonctions retournent un type Boolean (true ou false). isDefined retourne True si l'Option est un **Some(...)** et False sinon. isEmpty retourne True si l'option est la valeur **None** et False sinon.

```
object Demonstration {
   def main(args: Array[String]) {
      val Test = Option("Test function")
      println(Test.isDefined)
      println(Test.isEmpty)
   }
}
```

Nous obtenons donc :

```
true
false
```

Conclusion :

Il existe plusieurs intérêts à l'usage des monades tel que : analyses statiques et preuves de programmes plus simples, usage de l'appel par nécessité, optimisations. Monad (et ces deux fonctions unit et flatMap) est un type assez puissant. J'espère avoir réussi à faire la lumière sur le mystère des monades