类型驱动的Scala函数式编程

Thoughtworks 杨云
@诺铁

Scala

- 面向对象混合函数式编程
- 忽略面向对方,专注于函数式编程
- 用函数组合来解决问题
- 上下文里的类型

    ——Functor、Applicative、Monad

函数就是A => B

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

```scala
def add1(x:Int)(y:Int) = x + y
```

```scala
def add2: Int => Int => Int = { x => y => x + y }
```

```scala
> add2: Int => (Int => Int)
```

函数就是A => B

```
Optional.java,0,120,P
OptionalTest.java,0,151,T
ExcelImporter.java,345,137,P
ExcelImporterTest.java,0,160,T
StreamWriter.java,0,97,P
StreamWriterTest.java,0,107,T
ExcelReader.java,1007,0,P
```

函数就是A => B

```scala
case class Diff(fileName: String, originalLength: Int,
                currentLength: Int, codeType: CodeType.CodeType)

object Diff {
  def apply(source: Line): Diff = {
    val fields = source.split(",")
    new Diff(fields(0), fields(1).toInt, fields(2).toInt,
                if(fields(3) == "P") Product else Test)
  }
}
```

```scala
type Line = String
type FilePath = String
```

函数就是A => B

```scala
def reportNewFileNo: (FilePath => Int) = ???
```

```scala
def dataLines: (FilePath => List[Line]) = Source.fromFile(_).getLines().toList
```

```scala
def diffs: (List[Line] => List[Diff]) = _.map(Diff(_))
```

```scala
def filterNewAdded:(List[Diff] => List[Diff]) = _.filter(_.originalLength == 0)
```

```scala
def count: (List[Diff] => Int) = _.length
```

```scala
def reportNewFileNo0:(FilePath => Int) = { path =>
  Source.fromFile(path).getLines().toList.
    map(Diff(_)).filter(_.originalLength == 0).length
}
```

```scala
def reportNewFileNo1:(FilePath => Int) = { path =>
      count(filterNewAdded(diffs(dataLines(path)))) }
```

```scala
def reportNewFileNo2: (FilePath => Int) = {
  dataLines andThen diffs andThen filterNewAdded andThen count
}
```

函数就是A => B

```scala
def countOfProductCode: (List[Diff] => Int) = ???

def countOfTestCode: (List[Diff] => Int) = ???
```

```scala
def countOfProductCode1: (List[Diff] => Int) = _.count(_.codeType == Product)
def countOfTestCode1: (List[Diff] => Int) = _.count(_.codeType == Test)
```

```scala
def countByType:(CodeType => List[Diff] => Int) = { codeType => diffs =>
  diffs.count(_.codeType == codeType)
}
```

```scala
def countOfProductCode2:(List[Diff] => Int) = countByType(Product)

def countOfTestCode2:(List[Diff] => Int) = countByType(Test)
```

函数式编程语言需要解决的问题

- 怎么定义类型？
- 怎么定义函数
- 怎么组合起来？

普通类型和普通函数

```scala
def add1: (Int => Int) = _ + 1

def intToString: (Int => String) = _.toString

def reverse: (String => String)= _.reverse

def toInt: (String => Int) = _.toInt

def reverseInt: (Int => Int) = intToString andThen reverse andThen toInt
```

参数化类型——上下文里的类型

```scala
val intList: List[Int] = List(1, 2, 3)

val StrList: List[String] = List("a", "b", "c")

val dbConn: Option[String] = Some("mysql")
```

```scala
def add1: (Int => Int) = _ + 1

def intToString: (Int => String) = _.toString

def reverse: (String => String)= _.reverse

def toInt: (String => Int) = _.toInt

def reverseInt: (Int => Int) = intToString andThen reverse andThen toInt
```

怎么对List(1,2,3)应用函数add1呢？

参数化类型——上下文里的类型

```scala
val intList: List[Int] = List(1, 2, 3)

val StrList: List[String] = List("a", "b", "c")

val dbConn: Option[String] = Some("mysql")
```

```scala
def map:(List[Int] => (Int => Int) => List[Int]) = { xs => f =>
    xs match {
      case Nil => Nil
      case (head :: tail) => f(head) :: map(tail)(f)
    }
}
```

```scala
map(intList)(add1)
```

但是，这个map只能使用Int => Int函数，Int => String怎么办呢？String => String呢？

```scala
def intToString: (Int => String) = _.toString
```

Functor

```scala
val intList: List[Int] = List(1,2,3)

val StrList: List[String] = List("a","b","c")
```

利用泛型

```scala
def map[A,B]: (List[A] => (A =>B) => List[B])  = { xs => f =>
  xs match {
    case Nil => Nil
    case (head::tail) => f(head) :: map(tail)(f)
  }
}
```

```scala
map(map(intList)(add1))(intToString)
```

```scala
def addThenToStr: (List[Int] => List[String]) = map(add1) andThen map(intToString)
```

```scala
def flip[A,B,C]:((A => B => C) => (B => A => C)) = f => a => b => f(b)(a)

def addThenToStr: (List[Int] => List[String]) = {
  flip(map[Int,Int])(add1) andThen flip(map[Int,String])(intToString)
}
```

Functor

```scala
def map[A,B]: (List[A] => (A =>B) => List[B])  = { xs => f =>
  xs match {
    case Nil => Nil
    case (head::tail) => f(head) :: map(tail)(f)
  }
}
```

```scala
def mapOption[A, B]: (Option[A] => (A => B) => Option[B]) = { o => f =>
  o match {
    case None => None
    case Some(x) => Some(f(x))
  }
}
```

```scala
def mapFunction[A,B,C]: ((A => B) => (B => C) => (A => C)) = ???
  f1 => f2 => x => f2(f1(x))
}
```

Applicative Functor

```scala
def flatApply[A,B]: (List[A] => List[A=>B] => List[B]) = ???

def flatApplyOption[A,B]: (Option[A] => Option[A=>B] => Option[B]) = ???
```

```scala
def flatApply[A,B]: (List[A] => List[A=>B] => List[B]) = { xs => fs =>
  xs.map(x => fs.map(_(x))).flatten
}
```

```scala
def flatApply[A,B]: (List[A] => List[A=>B] => List[B]) = { xs => fs =>
  for(x <- xs; f <- fs) yield f(x)
}

def flatApplyOption[A,B]: (Option[A] => Option[A=>B] => Option[B]) = { xs => fs =>
  for(x <- xs; f <- fs) yield f(x)
}
```

Monad

```scala
def flatMap[A,B]: (List[A] => (A => List[B]) => List[B]) = ???

def flatMap[A,B]: (Option[A] => (A => Option[B]) => Option[B]) = ???
```

```scala
def flatMap[A,B]: (List[A] => (A => List[B]) => List[B]) = { xs => f =>
  xs.map(f).flatten
}

def flatMapOption[A,B]: (Option[A] => (A => Option[B]) => Option[B]) = { o => f =>
  o.map(f).flatten
}
```

在上下文里面运算

```scala
type Connection = String
type User = String

def conn: (String => Option[Connection]) = _ => Some("conn")
def user: (Connection => Option[List[User]]) = { _ =>
  Some(List("诺铁","老猪","老高"))
}
```

```scala
conn("mysql").flatMap(user).map(_.foreach(println))
  case None => None
```

```scala
for(
  c <- conn("mysql");
  u <- user(c)
){ u.foreach(println) }
```

```scala
.foreach(println)
```

总结

- 类型A和函数A => B 怎么组合？
- Context[A]和A => B 怎么组合？
- Context[A]和Context[A => B] 怎么组合？
- Context[A]和A => Context[B] 怎么组合？

Monadic

```
trait Future[+T] extends Awaitable[T]
```

```
def flatMap[S](f: (T) ⇒ Future[S])(implicit executor: ExecutionContext): Future[S]
```

Creates a new future by applying a function to the successful result of this future, and returns the result of the function as the new future.

```
def map[S](f: (T) ⇒ S)(implicit executor: ExecutionContext): Future[S]
```

Creates a new future by applying a function to the successful result of this future.