Функциональные паттерны

Функциональные паттерны

Когда принципы теории категорий начали применять в разработке, стало понятно, что просто композиции функций, отвечающих за логику приложения, не достаточно. Почти всегда, на ряду с композицией, нужно “что-то еще”. То контекст пронести, сквозь приложение, то логирование везде добавить, то результат побочный вернуть. Кроме этого, захотелось и с исключениями и с вводом-выводом в функциональном стиле работать. Одним их первых результатов работы, по решению вышепоставленных задач стала россыпь разнообразных монад. “Что-то еще” вшито в каждую из монад, так, чтобы программист мог сосредоточится на решении своей задачи.
Далее мы познакомимся с наиболее востребованными представителями этого функционального семейства. Можно считать их функциональными “паттернами”, ответом своим ООП-шным сородичам.

Важным свойством, которым должны обладать функциональные приложения является **referential transparency (RT).** Говорят, что им обладают приложения, которые можно заменить на результат их работы в каком бы контексте эти приложения не выполнялись. Для того, чтобы достичь **RT**, приложение должно быть детерминированным и не иметь состояния и побочных эффектов. Благодаря **RT** становится возможным передавать функции и целые приложения как значения, производить ленивые вычисления, эффективно оптимизировать общий код и заменять части приложения на результат их работы(табуировать функции)

Функциональные паттерны

Reader монада

С этим зверем мы уже познакомились в **lectures.di.reader.ReaderMonadProgram.scala.** Основная задача Reader - нести композицию функций туда, где ее не было. Вспомним, что функции вида **A => B** и **B => C,** прекрасно композируются, стандартными методами compose и andThan из Function1. А вот с композицией вида **A => F[B]** и **B => F[C]** уже сложнее. Здесь нам и приходит на помощь Reader. Достаточно, реализовать flatMap, map bind и пару других методов, для Reader[F, A, B]

```
type Id[A] = A
type ReaderT[F[_], A, B] = Kleisli[F, A, B]

object Reader {
 def apply[A, B](f: A => B): Reader[A, B] = ReaderT[Id, A, B](f)
}

final case class Kleisli[F[_], A, B](run: A => F[B]) {
 def map[C](f: B => C)(implicit F: Functor[F]): Kleisli[F, A, C] = ???
 def mapF[N[_], C](f: F[B] => N[C]): Kleisli[N, A, C] = ???
 def flatMap[C](f: B => Kleisli[F, A, C])(implicit F: FlatMap[F]): Kleisli[F, A, C] = ???
  ...
```

Функциональные паттерны

Writer монада

Эта монада родилась из необходимости накапливать какой-то контекст по ходу вычисления основной части программы. Часто ее применяют для накопления, например, данных для логирования.
Она оперирует с типом, вида **F(L, V)**, где **L**, это контекст вычисления, а **V** текущий результат.
Ключевым методом этой монады, является

```
def flatMap[U](f: V => WriterT[F, L, U])(implicit flatMapF: FlatMap[F], semigroupL: Semigroup[L]): WriterT[F, L, U]
```

Этот метод принимает **V => WriterT[F, L, U]**, как и остальные монады. Отличительной чертой его является то, что он берет на себя ответственность за композицию контекстов. Для того, чтобы это стало возможным, как видно из сигнатуры функции, контекст должен быть полугруппой. Проще говоря должен обладать бинарной ассоциативной операцией.

Упрощенный вариант этой монады из библиотеки cats, приведен ниже

Функциональные паттерны

```
final case class WriterT[F[_], L, V](run: F[(L, V)]) {

def written(implicit functorF: Functor[F]): F[L] =
  functorF.map(run)(_._1)

 def value(implicit functorF: Functor[F]): F[V] =
  functorF.map(run)(_._2)

 def ap[Z](f: WriterT[F, L, V => Z])(implicit F: Apply[F], L: Semigroup[L]): WriterT[F, L, Z] =
  WriterT(
    F.map2(f.run, run){
      case ((l1, fvz), (l2, v)) => (L.combine(l1, l2), fvz(v))
    })

 def flatMap[U](f: V => WriterT[F, L, U])(implicit flatMapF: FlatMap[F], semigroupL: Semigroup[L]): WriterT[F, L, U] =
  WriterT {
    flatMapF.flatMap(run) { lv =>
      flatMapF.map(f(lv._2).run) { lv2 =>
        (semigroupL.combine(lv._1, lv2._1), lv2._2)
      }
    }
  }

 def mapBoth[M, U](f: (L, V) => (M, U))(implicit functorF: Functor[F]): WriterT[F, M, U] =
  WriterT { functorF.map(run)(f.tupled) }
}
```

Функциональные паттерны

Writer монада

Рассмотрим пару примеров использования **writer**

lectures.cat.WriterTOps.scala

- **logActions - логирование с применением WriterT**
- **filterW - чуть более изощренный пример, фильтрация с WriterT**

На первый взгляд кажется, что с помощью помощью WriterT, может быть удобно реализовывать методы типа fold или aggregate. Но это не так, из-за того, что бинарная ассоциативная операция “вшита в тип” **L** и ее очень трудно переопределить.

задание lectures.cat.SeqWriterTOpsTest

Функциональные паттерны

State монада

Последняя из “большой тройки” монад. Она обеспечивает композицию функций вида **(S) => (S, A).** Где, **S -** состояние, а **A** текущий результат. **State,** в отличии, от **writer,** композирует функции**,** которые на вход принимают состояние. Т.е. если вычисления в приложении полностью задаются своим текущим состоянием, то это сигнал к тому, что в этом месте можно применить **state**. Иногда state применяют в качестве builder паттерна или в качестве “генераторов” каких-либо последовательностей. Например генератор случайных чисел легко можно реализовать с помощью **State Monad**.

Вариант реализации **cats.data.StateT**

Примеры применения **в lectures.cat.State.scala**

Функциональные паттерны

IO

Важными свойством функций, написанных в функциональном стиле, является то, что их можно передавать как значение и композировать с другими частями приложения. При этом, очевидно, откуда бы ни была запущена функция, она должна возвращать один и тот же результат. Это свойство называется **referential transparency(RT).** Чаще всего это свойство формулирую еще более строго. Говорят, что функция **referentially transparent**, если в любом месте ее вызов, можно заменить на результат ее вызова.
Классический пример нарушения **RT**

```
def fun = println("not ref transparent")
def use(p1: Any, p2: Any) = Unit

// this call would print "not ref transparent" twice
use(fun, fun)

// and here we have only one call
val res = fun
use(res, res)
```

Функциональные паттерны

IO

Чтобы сохранять **RT,** функция не должна иметь побочных эффектов в том числе она не должна обладать внутренним состоянием или каким-либо образом зависеть от контекста, в котором выполняется. В scala на уровне синтаксиса языка, мы не можем гарантировать этих свойств. Что мы можем, так это превратить функции вида **() => A** в монады и вместо вызовов этих функций, передавать эти монады как значения.

```
import cats.effect.IO

def fun = IO(println("not ref transparent"))
def use(p1: IO[Unit], p2: IO[Unit]) = Unit

// this call would print "not ref transparent" twice
use(fun, fun)

// and here we have only one call
val res = fun
use(res, res)
```

Функциональные паттерны

IO

Как видно из листинга, **IO** теперь referentially transparent. Но, функции содержащиеся в них - нет и мы ничего не можем с этим сделать. Лучшее чего можно добится - это отложить запуск этих функций до “конца времен”, т.е. до того времени, когда без результата этих функций мы уже не можем обойтись.

В **cats.effect.IO** есть несколькоы способов запустить выолнение

- **def unsafeRunSync(): A** - запускает “не безопасные” функции, внутри **IO.** Если есть функции, работающие асинхронно, главный поток блокируется
- **def unsafeRunAsync(cb: Either[Throwable, A] => Unit): Unit** - тоже самое, но поток не блокируется
- **def runAsync(cb: Either[Throwable, A] => IO[Unit]): IO[Unit] -** этот метод оборачивает **unsafeRunAsync** в **IO** и возвращается сразу.

Функциональные паттерны

IO

Реализаций IO монад в скала великое множество. Их можно найти в библиотеках **Cats.effect**, **scalaz** (лучше 8-й версии), **monix** (Task это тот же IO, по сути) и т.д. Ниже я приведу пример реализацию от JOHN A DE GOES, т.к. он наглядно выражает суть IO.

```
sealed class IO[A](val unsafePerformIO: () => A) {
 final def map[B](ab: A => B): IO[B] =
   new IO(() => ab(unsafePerformIO()))
 final def flatMap[B](afb: A => IO[B]): IO[B] =
   new IO(() => afb(unsafePerformIO()).unsafePerformIO())
 final def attempt: IO[Either[Throwable, A]] = new IO(() => {
   try Right(unsafePerformIO())
   catch {
     case t : Throwable => Left(t)
   }
 })
}
object IO {
 final def apply[A](a: => A): IO[A] = new IO(() => a)

 final def fail[A](t: Throwable): IO[A] = new IO(() => throw t)
}
```

Функциональные паттерны

IO + Async

Что, если нам нужно описать приложение, содержащего компоненты, выполняющиеся асинхронно? Для этого простого **IO** не достаточно, т.к. мы не знаем как с помощью **IO** композировать асинхронные вычисления.
Нам на выручку придет абстракция, которая позволяет организовать композицию через назначения колбэков асинхронным вычислениям. Callback - это функция, имеющая тип **Either[Throwable, A] => IO[Unit]**. Пример такой абстракции можно найти у того же JOHN A DE GOES. На самом деле, практически все реализации IO Async (cats, scalaz, monix) опираются на композицию через колбэки.

Задача по этой теме будет чуть позже, после того, как мы познакомимся с **Free**

Функциональные паттерны

IO + Async

```
final case class Async[A](register: (Either[Throwable, A] => IO[Unit]) => IO[Unit]) { self =>
 final def map[B](ab: A => B): Async[B] = Async[B] { callback =>
   self.register {
     case Left(e) => callback(Left(e))
     case Right(a) => callback(Right(ab(a)))
   }
 }
 final def flatMap[B](afb: A => Async[B]): Async[B] = Async[B] { callback =>
   self.register {
     case Left(e) => callback(Left(e))
     case Right(a) => afb(a).register(callback)
   }
 }
}
object Async {
 final def apply[A](a: => A): Async[A] = Async[A] { callback =>
   callback(Right(a))
 }

 final def fail[A](e: Throwable): Async[A] = Async[A] { callback =>
   callback(Left(e))
 }
}
```

Функциональные паттерны

Trampoline (трамплин)

Это простой но очень любопытный способ организовать хвостовую рекурсию функциональном стиле. Суть его в том, что функция, которую мы вызываем рекурсивно, возвращает одно из 2-х значений

- признак того, что вычисление завершено вместе со значением результата
- признак того, что нужно выполнить еще один наряду с функцией, которую нужно вызвать на следующем шаге

Признаки обычно реализуют в виде кейс классов следующего вида

```
trait Result[T]
case class Data[T](data: T) extends Result[T]
case class Call[T](call: () => Result[T]) extends Result[T]
```

Рекурентная функция запускается из другой функции, которая является интерпретатором ее ответов. Если вернулся, признак останова с результатом, интерпретатор обрабатывает результат и завершает программу. Если вернулся признак необходимости еще одного шага, интерпретатор вызывает рекурентную функцию.

Функциональные паттерны

Trampoline (трамплин)

С помощью трамплина любую рекурсивную функцию можно сделать хвостовой. Еще одной полезной особенностью является то, что рекурсию можно выполнять пошагово, останавливая ход вычисления или передавая частично выполненные вычисления в другую часть программы. Т.к. на каждый шаг рекурсии создается как минимум один дополнительный объект, трамплины немного уступают другим реализациям в производительности.

Если немного обобщить, можно сказать трамплин состоит из:

- доменной модели (Call и Data)
- функции преобразования этой модели
- интерпретатора результатов преобразования.

Такой способ организации приложений лег в основу концепции Free monad, о которых мы поговорим далее.

задание: lectures.cat.TrampolineFibsTest

Функциональные паттерны

Free

Free монады немного сбивающее с толку название т.к. за ним стоит что-то большее, чем реализация очередной монады. Это скорее подход к написанию приложений.
Познакомимся с ним на примере из библиотеки Cats: **lectures.cat.CatsFreeExample.scala.** В ней мы создадим приложение для работы с хранилищем пар ключ\значение.

Любое приложение, реализованное с помощью Free монад, можно разделить на 3 основные составные части:
Первая часть - алгебраический тип данных, описывающей предметную область. В нашем случае предметной областью являются операции над произвольным хранилищем. Эти операции описываются с помощью нескольких кейс классов. Вместе с ними часто описывают функции превращения кейс классов доменной модели в инстансы **Free[S[_], A].** Они нужны для композиции кейс классов при написании программы.

```
sealed trait KVStoreA[A]
case class Put[T](key: String, value: T) extends KVStoreA[Unit]
case class Get[T](key: String) extends KVStoreA[Option[T]]
case class Delete(key: String) extends KVStoreA[Unit]
```

Функциональные паттерны

Free

Вторая часть - программа, написанная в терминах предметной области.

```
val program: Free[KVStoreA, Option[Int]] = for {
 _ <- put("wild-cats", 2)
 _ <- update[Int]("wild-cats", _ + 12)
 _ <- put("tame-cats", 5)
 n <- get[Int]("wild-cats")
 _ <- delete("tame-cats")
} yield n
```

Программа представляет собой композицию операций над нашим ADT и записана в форме **for-comprehension**. Это значит, что наш доменные типы превратились в монады и каким-то образом обзавелись методами **flatMap** и **map,** необходимыми для композиции**.** Эти методы были предоставлены классом **Free[F[_], A].** Одна из причин, почему данные монады называют **Free**(свободными или бесплатными), в том, что имея интерпретатор и **ADT** мы, как бы, получаем монады бесплатно. Это, конечно же, не совсем так. Проанализируем реализацию **Free** из библиотеки **cats**.

Функциональные паттерны

```
/** Результат **/
private[free] final case class Pure[S[_], A](a: A) extends Free[S, A]
/** Результьаь внутри конейнера **/
private[free] final case class Suspend[S[_], A](a: S[A]) extends Free[S, A]
/** Отоженный вызов функции f  **/
private[free] final case class FlatMapped[S[_], B, C](c: Free[S, C], f: C => Free[S, B]) extends Free[S, B]

sealed abstract class Free[S[_], A] extends Product with Serializable {
 import Free.{ Pure, Suspend, FlatMapped }

final def map[B](f: A => B): Free[S, B] =
  flatMap(a => Pure(f(a)))
 /**
  * Bind the given continuation to the result of this computation.
  * All left-associated binds are reassociated to the right.
  */
 final def flatMap[B](f: A => Free[S, B]): Free[S, B] =
  FlatMapped(this, f)

  ...
}
```

Функциональные паттерны

Как видно из листинга, **Free -** это тоже ADT, состоящий из 3-х базовых типов, **Pure**, **Suspend** и **FlatMapped.** Методы **map** и **flatMap,** вместо вызовов, переданных в них функции, создают новые инстансы **Free**. Эти инстансы хранят информацию о действиях, которые надо будет совершить позже. Благодаря тому, что **Free** создают новые объекты, они не расходуют стек, вместо этого используя хип. Композиция **Free** монад по определению ленива, т.к. ни одна функция не вызывается сразу.
Стоит обратить особое внимание на то как реализован **flatMap**. При вызове этого метода создается инстанс **FlatMapped**, функция в котором должна будет быть выполнена последней. Это значит, что в момент, когда мы будем выполнять программу, нам надо будет развернуть вложенность так, чтобы самой внешней была функция, которую надо вызвать первой. Например, для вложенных друг в друга **FlatMapped,** это можно сделать вот так:

```
FlatMapped(FlatMapped(someFree, first), second) =>
        FlatMapped(someFree, (inp) => first(inp).flatMap(second))
```

Эту операцию нужно повторять рекурсивно, пока **someFree != Pure** или **someFree != Suspend**

Функциональные паттерны

Все что мы сделали до сих пор - это описали структуру программы. Осталось дело за малым - интерпретировать и запустить ее. Для этого нам понадобится интерпретатор, который может принимать несколько разных форм. В самом общем случае - это функция или набор функций, которые позволяют превратить **Free[S[], A]** в **M[R],** где **R -** тип результата, а **M[_] -** контейнер для него. Рассмотрим какие функции для интерпретации программ есть в библиотеке cats:

```
/**
 *  В данном случае интерпретатор - это полиморфная функция, которая
 *  знает, как превратить объекты ADT в объекты контейнера результата
 *  S[_] ~> M[_]
**/
final def foldMap[M[_]](f: FunctionK[S, M])(implicit M: Monad[M]): M[A] = ???

/**
 * Для функции S[Free[S, A]] => M[Free[S, A]] можно применить
 * При этом есть доп. ограничение на то, чтобы S - обязательно был функтором
 */
final def runM[M[_]](f: S[Free[S, A]] => M[Free[S, A]])(implicit S: Functor[S], M: Monad[M]): M[A] = ???

final def go(f: S[Free[S, A]] => Free[S, A])(implicit S: Functor[S]): A = ???
```

Функциональные паттерны

Как видно, когда дело доходит до интерпретаторов, **Free** монады, перестают быть бесплатными, т. к. или на **S[_]** или на **M[_]**, накладываются дополнительные ограничения.
Реализуем, наконец, интерпретатор для нашего примера:

```
new (KVStoreA ~> Id) {
  val kvs = mutable.Map.empty[String, Any]

  def apply[A](fa: KVStoreA[A]): Id[A] =
    fa match {
      case Put(key, value) =>
        println(s"put($key, $value)")
        kvs(key) = value
        ()
      case Get(key) =>
        println(s"get($key)")
        kvs.get(key).map(_.asInstanceOf[A])
      case Delete(key) =>
        println(s"delete($key)")
        kvs.remove(key)
        ()
    }
```

Функциональные паттерны

Осталось запустить нашу программу. Воспользуемся для этого методом **foldMap**

```
val result: Option[Int] = program.foldMap(impureCompiler)
```

Пример целиком находится в **lectures.cat.CatsFreeExample.scala**

Что бы убедится, что **Free**, это универсальный подход к разработке приложений, рассмотрим еще один пример, находящийся в **lectures.cat.FreeAsync.scala.**

Достоинства **Free :**

- Универсальность
- Переиспользуемость
- Простота. В случае небольших приложений

Недостатки

- Сложность, особенно для новичков
- Сложность больших приложений
- Значительные накладные расходы

Функциональные паттерны

Домашнее задание:

- **lectures.cat.IO4Free.scala**
- **lectures.cat.FreeAsync.scala**