Monad

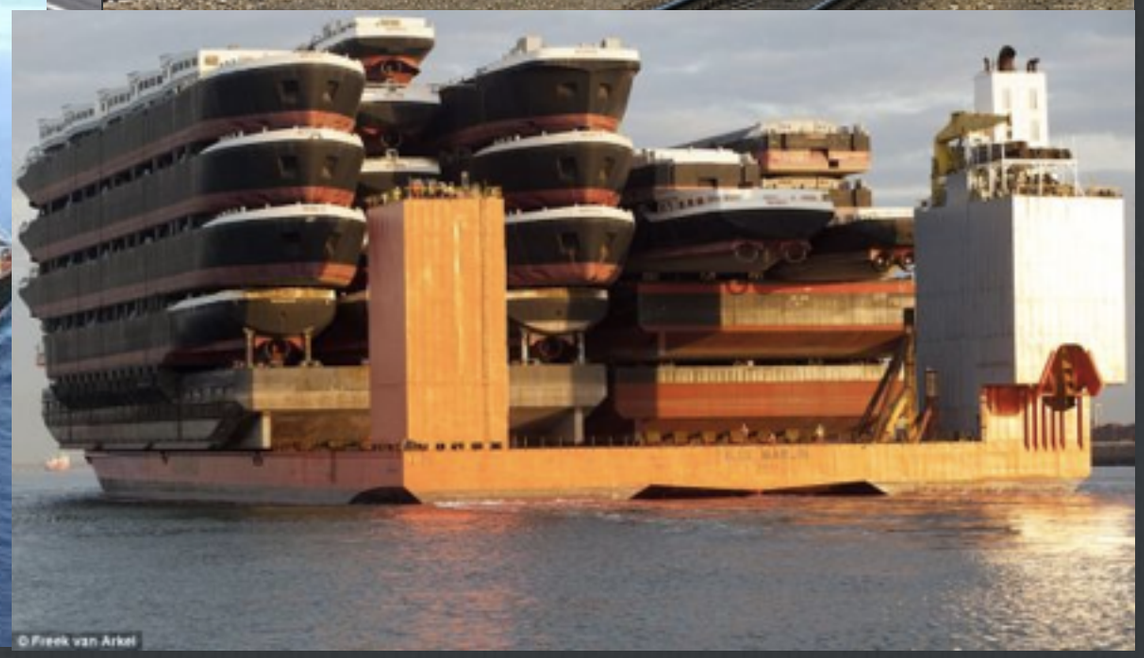
吴雪峰@ThoughtWorks

2014年6月22日

Monad is

- [ ] magic!
- [ ] category theory
- [ ] design pattern
- [ ] some api

石油运输



石油运输


俄 罗 斯
泰舍特
克拉斯诺雅尔斯克
伊尔库茨克
贝加尔湖
赤塔
斯科沃罗季诺
（距中国边境 70 公里）
戈尔诺－阿尔泰斯克
（西线）
（东线）
布里亚特
哈萨克斯坦
蒙 古
大庆
别列沃兹纳亚
北京★
朝鲜
日本
韩国
中 国
上海
东西伯利亚－太平洋（ESPO）石油管道工程
天然气管线
中国西气东输管线
（虚线为预计路线）

Monad is API

in Scala

Monads

- [ ] Option
- [ ] Try
- [ ] Future
- [ ] all collections

Option

```
trait Option {
  def isEmpty: Boolean
  def get: A

  def map[B](f: A => B): Option[B] =
        if (isEmpty) None else Some(f(this.get))

  def flatMap[B](f: A => Option[B]): Option[B] =
        if (isEmpty) None else f(this.get)
}

case class Some[+A](x: A) extends Option[A] {
  def isEmpty = false
  def get = x
}

case object None extends Option[Nothing] {
  def isEmpty = true
  def get = throw new NoSuchElementException("None.get")
}
```

```scala
def getPrice(): Option[Int]

def getQuantities(): Option[Int]
```

```scala
def amount(): Option[Int] =
  getPrice().flatMap(price => getQuantities().map(price * _))
```

```scala
def amount(): Option[Int] = {
  for {
    price <- getPrice()
    quantities <- getQuantities()
  } yield price * quantities
}
```

Try

```scala
abstract class Try[+T] {
  def map[U](f: T => U): Try[U]
  def flatMap[U](f: T => Try[U]): Try[U]
}
```

```scala
case class Success[+T](value: T) extends Try[T] {
  def map[U](f: T => U): Try[U] = Try[U](f(value))
  def flatMap[U](f: T => Try[U]): Try[U] =
    try f(value)
    catch {
      case NonFatal(e) => Failure(e)
    }
}

case class Failure[+T](exception: Throwable) extends Try[T] {
  def map[U](f: T => U): Try[U] = this.asInstanceOf[Try[U]]
  def flatMap[U](f: T => Try[U]): Try[U] = this.asInstanceOf[Try[U]]
}
```

Try

- [ ] get DB Url file I/O
- [ ] get Connection network I/O
- [ ] create Statement
- [ ] execute Query

```
try {
  val driver = getDriver()
  val url = getDBUrl()
  val username = getUserName()
  val password = getPassword()
  Class.forName(driver)
  val conn = DriverManager.getConnection(url, user, password)
  val statement = conn.createStatement()
  val resultSet = statement.executeQuery("select * from student")
  resultSet
} catch {
    case e: Exception =>
}
```

说好的

- [ ] 可重用
- [ ] 可测试
- [ ] 可扩展

```scala
def getDriver(): Try[String]
def getDBUrl(): Try[String]
def getUserName(): Try[String]
def getPassword(): Try[String]


for {
  driver <- getDriver()
  url <- getDBUrl()
  username <- getUserName()
  password <- getPassword()
  _ <- Try(Class.forName(driver))
  conn <- Try(DriverManager.getConnection(url, user, password))
  statement <- conn.createStatement()
  resultSet <- statement.executeQuery("select * from student")
} yield resultSet

resultSet: Try[ResultSet]
```

Future

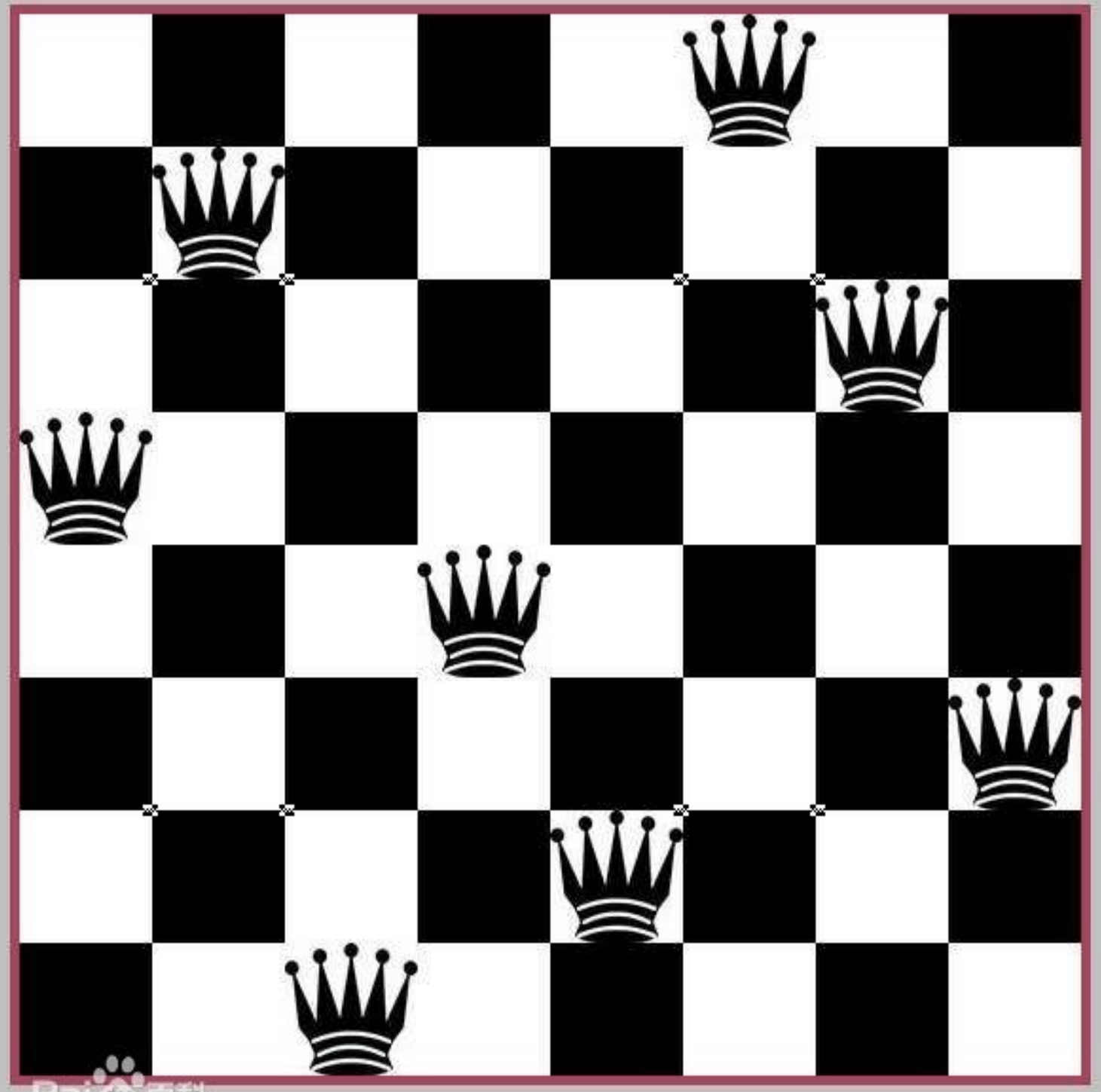
```
scala.concurrent.Future

def getFollowers(url: String): String
def extractWeiboFollowers(json: String): Int
def extractWechatFriends(xml: String): Int

for {
  weiboJson <- Future(getFollowers("http://api.weibo.com/u/123/followers"))
  wechatXml <- Future(getFollowers("http://api.wechat.com/user/123/friends"))
  weboFollower = extractWeiboFollowers(weiboJson)
  wechatFriends = extractWechatFriends(wechatXml)
} yield weboFollower + wechatFriends * 5

influence: Future[Int]

Await.result(influence, 10 seconds)
```



已经搜索到第67组解，准备搜索第68组解...

List

```
def queens(n: Int): List[List[(Int, Int)]] = {
    def placeQueens(k: Int): List[List[(Int, Int)]] =
      if (k == 0)
        List(List())
      else for {
          queens <- placeQueens(k - 1)
          column <- 1 to n
          queen = (k, column) if isSafe(queen, queens)
        } yield queen :: queens
    placeQueens(n)
 }

def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =
    queens forall (q => !inCheck(queen, q))
  def inCheck(q1: (Int, Int), q2: (Int, Int)) =
    q1._1 == q2._1 ||  // same row
    q1._2 == q2._2 ||  // same column
    (q1._1 - q2._1).abs == (q1._2 - q2._2).abs // on diagonal
```

Monad is Design Pattern

Use Cases

- ☐ nondeterminism
- ☐ exception handling
- ☐ concurrency
- ☐ parsing
- ☐ continuations
- ☐ input/output
- ☐ variable assignment

scalaz Validation

```
sealed trait Validation[E,A] {
     def map[B](f: A => B): Validation[E,B]
     def flatMap[B](f: A => Validation[E,B]): Validation[E,B]
     def liftFail[F](f: E => F): Validation[F,A] //unrelated to monads
}
case class Success[E,A](a: A) extends Validation[E,A] {
     def map[B](f: A => B): Validation[E,B] = new Success(f(a))
     def flatMap[B](f: A => Validation[E,B]): Validation[E,B] = f(a)
     def liftFail[E](f: E => F): Validation[F, A] = new Success(a)
}

case class Failure[E,A](e: E) extends Validation[E,A] {
     def map[B](f: A => B): Validation[E,B] = new Failure(e)
     def flatMap[B](f: A => Validation[E,B]): Validation[E,B] = new Failure(e)
     def liftFail[E](f: E => F): Validation[F, A] =  new Failure(e)
}
```

```scala
case class Person(name: String, birthDate: Date, address: List[String])
val bad = "Name Only"
val partial = "Joe Colleague;1974-??-??;Rotterdam"
val good = "Bart Schuller;2012-02-29;Some Street 123, Some Town"

def tryParse(s: String) {
   println("Trying to parse: "+s)
   parsePerson(s) match {
     case Success(p) => println("  Succesfully parsed a person: " + p)
     case Failure(f) => {
       println("  Parsing failed, with the following errors:")
       f foreach { error => println("    "+error) }
     }
   }
 }
```

```scala
def parseDate(in: String): ValidationNEL[String, Date] = {
   val sdf = new SimpleDateFormat("yyyy-MM-dd")
   sdf.parse(in, new ParsePosition(0)) match {
     case null => ("Can't parse ["+in+"] as a date").failNel[Date]
     case date => date.successNel[String]
   }
}
```

```scala
def parsePerson(in: String): ValidationNEL[String, Person] = {
   val components = in.split(';').lift
   val name = components(0).fold(_.successNel[String],
                                 "No name found".failNel[String])
   val date = components(1).fold(parseDate(_),
                                 "No date found".failNel[Date])
   val address = components(2).fold(parseAddress(_),
                                 "No address found".failNel[List[String]])
   (name ⊛ date ⊛ address) { Person(_, _, _) }
}
```

greet

```
def greet {
  println("What is your name?")
  val name = readLine
  println(s"Hello, $name!")
}
```

Problems with I/O

- ☐ I/O (file, network)
- ☐ Monolithic, non-modular, limited reuse
- ☐ Novel compositions are difficult
- ☐ Difficult to test
- ☐ Difficult to scale

greet

```
case class IO[A](run: () => A) {
 def map[B](f: A => B): IO[B] = IO(() => f(run()))
 def flatMap[B](f: A => IO[B]): IO[B] = f(run())
}
```

```
def io[A](a: => A): IO[A] = IO(() => a)
def putLine(s: String): IO[Unit] = io(println(s))
def getLine: IO[String] = io(readLine)

 def greet: IO[Unit] = for {
  _ <- putLine("What is your name?")
  name <- getLine
  result <- putLine(s"Hello, $name")
 } yield ()

 greet.run()

 def getLine: IO[String] = io("Xuefeng")
```

Future[Try[Option]]

–Johnny Appleseed

What have we gained?

- ☐ Separation of I/O code from your logic
- ☐ An IO data type that we can inspect and is highly extensible.
- ☐ We can test programs without performing I/O actions (e.g. Console).
- ☐ Type safety
- ☐ First-class compositional I/O actions
- ☐ Algebraic reasoning

Monad is category theory