ZIO

Rising star in asynchronous and concurrent programming

Jiří

April 17, 2021

Hsinchu

Functional effects

Imperative programming

```
import scala.util.control.Breaks._
object Main {
  def main(args: Array[String]): Unit = {
    val result = collection.mutable.ArrayBuffer.empty[String]

    breakable {
      while (true) {
        val line = scala.io.StdIn.readLine()
        if (line == "q") break()
        else result.append(line)
      }
    }
    println(result)
  }
}
```

Imperative programming

```
import scala.util.control.Breaks._
object Main {
  def main(args: Array[String]): Unit = {
    val result = collection.mutable.ArrayBuffer.empty[String]

    breakable {
      while (true) {
        val line = scala.io.StdIn.readLine()
        if (line == "q") break()
        else result.append(line)
      }
    }
    println(result)
  }
}
```

Imperative programming

```
def initialize(conf: Config): Unit = {
  initializeDB(conf.db)
  // 10 more lines
  initializeMaps(conf.geo)
  // 15 more lines
  preloadCache()
  // 12 more lines
  initializeNotifications()
}
```

Declarative programming

```
def initializeDB(conf: Config): Database = ???
def initializeMaps(c: Cache, conf: Config): Maps = ???
def preloadCache(): Cache = ???
def initializeNotifications(c: Cache): Notifications = ???

def initialize(conf: Config): Env = {
  val cache = preloadCache()
  val maps = initializeMaps(cache, conf)
  val db = initializeDB(conf)
  val notif = initializeNotifications(cache)

  Env(cache, maps, db, notif)
}
```

Referential transparency

$$a = 3 + 5$$

$$b = 4 + 1$$

$$c = a + b$$

$$c = (3 + 5) + 5 = 13$$

$$c = 8 + 5 = 13$$

$$d = a + a$$

$$d = (3 + 5) + (3 + 5) = 16$$

$$d = 8 + 8 = 16$$

$$d = 2 \cdot (3 + 5) = 16$$

Referential transparency

```
val y1 = Random.nextInt() + Random.nextInt()

val x = Random.nextInt()
val y2 = x + x
```

Referential transparency

An expression is *referentially transparent* if we can substitute all its subexpressions with their respective values while preserving the overall result.

Referential transparency

An expression is *referentially transparent* if we can substitute all its subexpressions with their respective values while preserving the overall result.

f is *pure function* if the expression $f(x)$ is referentially transparent for all referentially transparent x.

Referential transparency

An expression is *referentially transparent* if we can substitute all its subexpressions with their respective values while preserving the overall result.

f is *pure function* if the expression $f(x)$ is referentially transparent for all referentially transparent x.

Expression has *side effects* if it is not referentially transparent.

Referential transparency

An expression is *referentially transparent* if we can substitute all its subexpressions with their respective values while preserving the overall result.

f is *pure function* if the expression $f(x)$ is referentially transparent for all referentially transparent x.

Expression has *side effects* if it is not referentially transparent.

Functional programming is programming with pure functions.

Functional programming — consequences

- no mutability
- no assignments
- result of pure function depends only on its arguments
- no side effects

Functional programming — advantages

- sharing values in concurrent applications
- reasoning about code (equational reasoning)
- hold less things in your memory
- testability

Functional programming — equational reasoning

```
def sum(l: List[Int]): Int =
  l match {
    case Nil => 0
    case first :: rest =>
          first + sum(rest)
  }

sum(List(1, 2, 3))
```

Functional programming — equational reasoning

```
def sum(l: List[Int]): Int =
  l match {
    case Nil => 0
    case first :: rest =>
          first + sum(rest)
  }

sum(List(1, 2, 3))
```

```
   sum(List(1, 2, 3))
⇒ sum(1 :: 2 :: 3 :: Nil)
⇒ 1 + sum(2 :: 3 :: Nil)
⇒ 1 + 2 + sum(3 :: Nil)
⇒ 1 + 2 + 3 + sum(Nil)
⇒ 1 + 2 + 3 + 0
⇒ 6
```

Functional effects

Functional effects are a method of turning side-effecting expressions into referentially transparent ones.

Naïve effect type

```scala
class IO[A](val run: () => A)

def say(s: String): IO[Unit] = new IO(() => println(s))

val program: IO[Unit] = say("Hello")
```


program.run()

Functional effects

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

val rnd: IO[Int] = new IO(() => Random.nextInt())
val rndMod5: IO[Int] = rnd.map(n => math.abs(n % 5))
```

Functional effects

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

val rnd: IO[Int] = new IO(() => Random.nextInt())
val rndMod5: IO[Int] = rnd.map(n => math.abs(n % 5))


 val program: IO[Unit] =
   rndMod5
     .flatMap(n =>
       say(s"Number $n")
     )
```

Functional effects

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

val rnd: IO[Int] = new IO(() => Random.nextInt())
val rndMod5: IO[Int] = rnd.map(n => math.abs(n % 5))
```

```
val program: IO[Unit] =
  rndMod5
    .flatMap(n =>
      say(s"Number $n")
    )
```

```
val program: IO[Unit] =
  for {
    n <- rndMod5
    _ <- say(s"Number $n")
  } yield ()
```

Functional effects

```
class IO[A](val run: () => A) {
  def zip[B](other: IO[B]): IO[(A, B)] =
    for {
      a <- this
      b <- other
    } yield (a, b)
}

val rnd: IO[Int] = new IO(() => Random.nextInt())
val rnd2: IO[(Int, Int)] = rnd.zip(rnd)
```


ZIO
Type-safe, composable asynchronous and concurrent programming for Scala


ZIO[■, E, A]

```scala
import zio._

object Main extends App {

  val rnd: ZIO[■, Throwable, Int] = ZIO.effect(Random.nextInt())
  def say(s: String): ZIO[■, Throwable, Unit] = ZIO.effect(println(s))

  val program: ZIO[■, Throwable, Unit] =
    for {
      n <- rnd
      _ <- say(s"Number $n")
    } yield ()

  def run(args: List[String]) = program.exitCode
}
```

```scala
val rnd: ZIO[■, Nothing, Int] =
  ZIO.effectTotal(Random.nextInt())

def say(s: String): ZIO[■, Nothing, Unit] =
  ZIO.effectTotal(println(s))

val program: ZIO[■, Nothing, Unit] =
  for {
    n <- rnd
    _ <- say(s"Number $n")
  } yield ()
}
```

```scala
import zio._

object Main extends App {

  val rnd: ZIO[■, Nothing, Int] = ???
  def say(s: String): ZIO[■, Nothing, Unit] = ???

  val program: ZIO[■, Nothing, Unit] =
    for {
      n <- rnd
      _ <- say(s"Number $n")
    } yield ()

  def run(args: List[String]): ZIO[■, Nothing, ExitCode] =
    program.exitCode
}
```

```scala
import zio._
import zio.console._
import zio.random._

object Main extends App {

  val program: ZIO[■, Nothing, Unit] =
    for {
      n <- nextInt
      _ <- putStrLn(s"Number $n")
    } yield ()

  def run(args: List[String]): ZIO[■, Nothing, ExitCode] =
    program.exitCode
}
```

```scala
// from zio.console._
val getStrLn: ZIO[■, IOException, String] = ???

val program: ZIO[■, String, Unit] =
  for {
    _ <- putStr("Answer: ")
    line <- getStrLn.mapError(_ => "IO Error")
    response <-
      if (line == "42") ZIO.succeed("Correct")
      else ZIO.fail("Wrong")
    _ <- putStrLn(response)
  } yield ()
```

```scala
val dialog: ZIO[■, String, String] =
  putStr("Answer: ") *> getStrLn.mapError(_ => "IO Error")

val program: ZIO[■, String, Unit] =
  for {
    line <- dialog
    response <-
      if (line == "42") ZIO.succeed("Correct")
      else ZIO.fail("Wrong")
    _ <- putStrLn(response)
  } yield ()

val program2 = program.retry(Schedule.recurs(3))
```

```scala
import zio.random._

val rnd2: ZIO[■, Nothing, (Int, Int)] = nextInt.zip(nextInt)

val rnd2: ZIO[■, Nothing, (Int, Int)] = nextInt <*> nextInt
```

```
val program1: ZIO[■, E, A] = ???
val program2: ZIO[■, E, B] = ???

// program1.zip(program2)
val program: ZIO[■, E, (A, B)] = program1 <*> program2

// program1.zipPar(program2)
val program: ZIO[■, E, (A, B)] = program1 <&> program2
```

```
val program1: ZIO[■, E, A] = ???
val program2: ZIO[■, E, A] = ???

val program: ZIO[■, E, A] = program1.race(program2)
```

```scala
val program1: ZIO[■, E, A] = ???
val program2: ZIO[■, E, B] = ???


val program: ZIO[■, E, B] =
  for {
    f1 <- program1.fork    // Fiber[E, A]
    f2 <- program2.fork    // Fiber[E, B]
    // ...
    res1 <- f1.join
    // ...
    res2 <- f2.join
  } yield res2
```

Concurrency primitives

- `Semaphore` — asynchronous non-blocking semaphore
- `Promise[A]` — eventually provided value
- `Ref[A]` — mutable reference
- `Queue[A]` — asynchronous non-blocking queue
- `Hub[A]` — asynchronous message hub

Mutable reference

```
val program: ZIO[■, Nothing, Unit] =
  for {
    ref <- Ref.make(0L)
    _   <- ZIO.foreachPar(1 to 1e6)(n => ref.update(_ + n))
    res <- ref.get
    _   <- putStrLn(s"Result: $res")
  } yield ()

// Result: 500000500000
```

Resource safety

```
def connect: ZIO[■, Error, Connection] = ???
def close(c: Connection): ZIO[■, Nothing, Unit] = ???

val program: ZIO[■, Error, Unit] =
  connect.bracket(conn => close(conn)) { conn =>
    // use conn
  }
```

Resource safety

```
def connect: ZIO[■, Error, Connection] = ???
def close(c: Connection): ZIO[■, Nothing, Unit] = ???

val connection: Managed[Error, Connection] =
  Managed.make(connect)(close(_))

val program: ZIO[■, Error, Unit] =
  connection.use(conn => /* use connection */ )
```

Resource safety

```
def server: ZIO[■, Error, Server] = ???
def closeServer(s: Server): ZIO[■, Nothing, Unit] = ???
def connect: ZIO[■, Error, Connection] = ???
def closeConn(c: Connection): ZIO[■, Nothing, Unit] = ???

case class Resources(s: Server, c: Connection)

val resources: Managed[Error, Resources] =
  for {
    s <- Managed.make(server)(closeServer(_))
    c <- Managed.make(connect)(closeConn(_))
  } yield Resources(s, c)

val program: ZIO[■, Error, Unit] =
  resources.use(res => /* use resources */ )
```

The third parameter


ZIO[■, E, A]

```
ZIO[R, E, A]
```

Declaring program dependencies

```
val prgA: ZIO[Database, E, A] = ???
val prgB: ZIO[Cache, E, B] = ???
```

Declaring program dependencies

```
val prgA: ZIO[Database, E, A] = ???
val prgB: ZIO[Cache, E, B] = ???

val prg1: ZIO[Database with Cache, E, (A, B)] =
  for {
    a <- prgA
    b <- prgB
  } yield (a, b)
```

Declaring program dependencies

```
val prgA: ZIO[Database, E, A] = ???
val prgB: ZIO[Cache, E, B] = ???

val prg1 =
  for {
    a <- prgA
    b <- prgB
  } yield (a, b)
```

Declaring program dependencies

```
val prgA: ZIO[Database, E, A] = ???
val prgB: ZIO[Cache, E, B] = ???

val prg1 =
  for {
    a <- prgA
    b <- prgB
  } yield (a, b)

val prg2 = prgA.zip(prgB)
```

Declaring program dependencies

```
val prg: ZIO[Any, E, A] = ???
```

Declaring program dependencies

```
val prg: ZIO[Any, E, A] = ???

type IO[E, A] = ZIO[Any, E, A]
```

Declaring program dependencies

```
val prg: ZIO[Any, E, A] = ???

type IO[E, A] = ZIO[Any, E, A]

val prg: IO[E, A] = ???
```

Providing dependencies

```
object MyApp extends zio.App {

  val prg: ZIO[Database with Cache, Nothing, Unit] = ???

  def run(args: List[String]) =
    prg
      .provideLayer(???)
      .exitCode

}
```

ZLayer creates dependency instances

```scala
val dbLayer: ZLayer[Any, Nothing, Database] = ???

val cacheLayer: ZLayer[Any, Nothing, Cache] = ???
```

ZLayer creates dependency instances

```
val dbLayer: ZLayer[Any, Nothing, Database] = ???

val cacheLayer: ZLayer[Any, Nothing, Cache] = ???

val fullLayer: ZLayer[Any, Nothing, Database with Cache] =
  dbLayer ++ cacheLayer
```

Providing dependencies

```
object MyApp extends zio.App {

  val prg: ZIO[Database with Cache, Nothing, Unit] = ???

  val dbLayer: ZLayer[Any, Nothing, Database] = ???
  val cacheLayer: ZLayer[Any, Nothing, Cache] = ???

  def run(args: List[String]) =
    prg
      .provideLayer(dbLayer ++ cacheLayer)
      .exitCode
}
```

Providing dependencies

```
object MyApp extends zio.App {

  val prg: ZIO[Database with Cache, Nothing, Unit] = ???

  val dbLayer: ZLayer[Config, Nothing, Database] = ???
  val cacheLayer: ZLayer[Any, Nothing, Cache] = ???
  val configLayer: ZLayer[Any, Nothing, Config] = ???

  def run(args: List[String]) =
    prg
      .provideLayer((configLayer >>> dbLayer) ++ cacheLayer)
      .exitCode
}
```

Defining dependency types

```
object database {




}
```

Defining dependency types

```
object database {

  trait Service {
    def load(key: String): IO[DbError, Int]
  }




}
```

Defining dependency types

```
type Database = Has[database.Service]

object database {

  trait Service {
    def load(key: String): IO[DbError, Int]
  }




}
```

Defining dependency types

```
type Database = Has[database.Service]

object database {

  trait Service {
    def load(key: String): IO[DbError, Int]
  }

  val postgres: ZLayer[Any, Nothing, Database] = ???

  val inMemory: ZLayer[Any, Nothing, Database] = ???
}
```

Defining dependency types

```
object database {

  trait Service {
    def load(key: String): IO[DbError, Int]
  }

  def load(key: String): ZIO[Database, DbError, Int] =
    ZIO.accessM(_.get.load(key))

}
```

Defining dependency types

```
object database {
  def load(key: String): ZIO[Database, DbError, Int] =
    ZIO.accessM(_.get.load(key))
}

import database._

val prg: ZIO[Database, DbError, Unit] =
  for {
    i <- load("key")
    ...
  } yield ()
```

An example revisited

```scala
import zio._
import zio.console._
import zio.random._

object Main extends App {

  val program: ZIO[■, Nothing, Unit] =
    for {
      n <- nextInt
      _ <- putStrLn(s"Number $n")
    } yield ()

  def run(args: List[String]): ZIO[■, Nothing, ExitCode] =
    program.exitCode
}
```

An example revisited

```
import zio._
import zio.console._
import zio.random._

object Main extends App {

  val program: ZIO[Console with Random, Nothing, Unit] =
    for {
      n <- nextInt
      _ <- putStrLn(s"Number $n")
    } yield ()

  def run(args: List[String]): ZIO[ZEnv, Nothing, ExitCode] =
    program.exitCode
}
```

Ecosystem

ZIO Magic

```
val dbLayer: ZLayer[Config, Nothing, Database] = ???
val cacheLayer: ZLayer[Any, Nothing, Cache] = ???
val configLayer: ZLayer[Any, Nothing, Config] = ???

val program: ZIO[Database with Cache, Nothing, Unit] = ???

def run(args: List[String]) =
  program
    .provideLayer((configLayer >>> dbLayer) ++ cacheLayer)
    .exitCode
```

ZIO Magic

```scala
import zio.magic._

// def run(args: List[String]) =
//  program
//    .provideLayer((configLayer >>> dbLayer) ++ cacheLayer)
//    .exitCode

def run(args: List[String]) =
  program
    .inject(configLayer, dbLayer, cacheLayer)
    .exitCode
```

ZIO Magic

```scala
def run(args: List[String]) =
  program
    .inject(dbLayer, cacheLayer)
    .exitCode
```

```
[error] ZLayer Auto Assemble
[error]
[error] missing Main.Config
[error]     for dbLayer
[error]
[error]     program.inject(dbLayer, cacheLayer).exitCode
[error]                    ^
[error] one error found
```

ZIO Stream (example by Itamar Ravid)

```
val files = ZIO.effect(Files.walk(Paths.get("~/dev/scala/zio")))

val countLines =
  ZStream.fromJavaStream(files)
    .mapMPar(6) { path =>
      ZStream.fromInputStream(Files.newInputStream(path))
             .aggregate(ZSink.utf8DecodeChunk)
             .aggregate(ZSink.splitLines)
             .fold(0)(_ + _.size)
    }
    .fold(0)(_ + _)
    .flatMap(lines => putStrLn(lines.toString()))

val program = countLines.runDrain
```

Other

- zio-http
- zio-json
- zio-logging
- zio-metrics
- zio-kafka
- zio-actors
- zio-aws
- caliban (GraphQL)
- zio-sql
- zio-gprc
- zio-prelude
- zio-config

Interoperability

- Cats Effect
- Scala Future
- Java Future
- JavaScript
- Monix
- Twitter
- Reactive Streams


台灣製造．品質可靠
Purely Functional
Order 沙發

Thank you for attention