MONIX TASK

LAZY, ASYNC & AWESOME

Alexandru Nedelcu

Software Developer @ eloquentix.com
@alexelcu / alexn.org

WHAT IS MONIX?

- Scala / Scala.js library
- For composing asynchronous programs
- Exposes Observable & Task
- Typelevel Incubator
- 2.0-RC2
- See: monix.io

EVALUATION

EVALUATION IN SCALA

	Eager	Lazy
	A	() => A

EVALUATION IN SCALA

	Eager	Lazy
Synchronous	A	() => A
Asynchronous	(A => Unit) => Unit	(A => Unit) => Unit

EVALUATION IN SCALA

	Eager	Lazy
Synchronous	A	() => A
		Function0[A]
Asynchronous	(A => Unit) => Unit	(A => Unit) => Unit
	Future[A]	Task[A]

> "A FUTURE REPRESENTS A VALUE, DETACHED FROM TIME"

Viktor Klang

MONIX TASK

TASK

```scala
import monix.execution.Scheduler
import Scheduler.Implicits.global
import monix.eval.Task

val task =
  Task { 1 + 1 }

// Later ...
task.runAsync {
  case Success(value) =>
    println(v)

  case Failure(ex) =>
    println(ex.getMessage)
}
```

FUTURE

```scala
import scala.concurrent.ExecutionContext
import ExecutionContext.Implicits.global
import scala.concurrent.Future

val future =
  Future { 1 + 1 }

// Later ...
future.onComplete {
  case Success(value) =>
    println(v)

  case Failure(ex) =>
    println(ex.getMessage)
}
```

TASK'S BEHAVIOR

- allows fine-grained control over the evaluation model
- doesn't trigger any effects until runAsync
- doesn't necessarily execute on another logical thread
- allows for cancelling of a running computation

EVALUATION

```
// Strict evaluation
Task.now { println("effect"); "immediate" }

// Lazy / memoized evaluation
Task.evalOnce { println("effect"); "memoized" }

// Equivalent to a function
Task.evalAlways { println("effect"); "always" }

// Builds a factory of tasks ;-)
Task.defer(Task.now { println("effect") })

// Guarantees asynchronous execution
Task.fork(Task.evalAlways("Hello!"))
```

MEMOIZATION (1/2)

```
val task1 = Task.evalOnce("effect")

val task2 = Task.evalAlways("effect")

val task3 = Task.evalAlways("effect").memoize
```

MEMOIZATION (2/2)

`task.memoize` vs `task.runAsync`

TAIL RECURSIVE LOOPS (1/4)

```
@tailrec
def fib(cycles: Int, a: BigInt, b: BigInt): BigInt =
  if (cycles > 0)
    fib(cycles-1, b, a + b)
  else
    b
```

TAIL RECURSIVE LOOPS (2/4)

```scala
def fib(cycles: Int, a: BigInt, b: BigInt): Task[BigInt] =
  if (cycles > 0)
    Task.defer(fib(cycles-1, b, a+b))
  else
    Task.now(b)
```

TAIL RECURSIVE LOOPS (3/4)

```scala
def fib(cycles: Int, a: BigInt, b: BigInt): Task[BigInt] =
  Task.evalAlways(cycles > 0).flatMap {
    case true =>
      fib(cycles-1, b, a+b)
    case false =>
      Task.now(b)
  }
```

FlatMap, like all of Task’s operators, is stack-safe ;-)

TAIL RECURSIVE LOOPS (4/4)

```
// Mutual Tail Recursion, ftw!!!
def odd(n: Int): Task[Boolean] =
  Task.evalAlways(n == 0).flatMap {
    case true => Task.now(false)
    case false => even(n - 1)
  }

def even(n: Int): Task[Boolean] =
  Task.evalAlways(n == 0).flatMap {
    case true => Task.now(true)
    case false => odd(n - 1)
  }

even(1000000)
```

SCHEDULER

```scala
package monix.execution

trait Cancelable {
  def cancel(): Unit
}

trait Scheduler extends ExecutionContext {
  def scheduleOnce(initialDelay: Long, unit: TimeUnit,
    r: Runnable): Cancelable

  def currentTimeMillis(): Long
  def executionModel: ExecutionModel

  def scheduleWithFixedDelay(...): Cancelable
  def scheduleAtFixedRate(...): Cancelable
}
```

EXECUTION MODEL

EXECUTION MODEL

- in batches, by default
- always asynchronous
- preferably synchronous

EXECUTION MODEL: BATCHED

```
import monix.execution._
import monix.execution.schedulers._
import ExecutionModel.BatchedExecution

implicit val scheduler =
  Scheduler.computation(
    parallelism=4,
    executionModel=BatchedExecution(batchSize=1000)
  )
```

EXECUTION MODEL: ALWAYS ASYNC

```
import monix.execution._
import monix.execution.schedulers._
import ExecutionModel.AlwaysAsyncExecution

implicit val scheduler =
  Scheduler.computation(
    parallelism=4,
    executionModel=AlwaysAsyncExecution
  )
```

EXECUTION MODEL: PREFER SYNCHRONOUS

```
import monix.execution._
import monix.execution.schedulers._
import ExecutionModel.SynchronousExecution

implicit val scheduler =
  Scheduler.computation(
    parallelism=4,
    executionModel=SynchronousExecution
  )
```

REAL ASYNCHRONY

REAL ASYNCHRONY

```
(A => Unit) => Unit
```

REAL ASYNCHRONY

Future[A] => A

REAL ASYNCHRONY

~~Future[A] => A~~

Always a platform specific hack, just say no to hacks!

REAL ASYNCHRONY

```
def fromFuture[A](future: Future[A]): Task[A] =
  Task.create { (scheduler, callback) =>
    implicit val ec = scheduler
    // Waiting ...
    future.onComplete {
      case Success(v) =>
        callback.onSuccess(v)
      case Failure(ex) =>
        callback.onError(ex)
    }
    // Futures can't be canceled
    Cancelable.empty
  }
```

REAL ASYNCHRONY

```
// From Future ...
val task = Task.defer(
  Task.fromFuture(Future { "effect" }))

// And back again ...
val future = task.runAsync

// If we want the result ...
Await.result(future, 10.seconds)
```

REAL ASYNCHRONY

(◕‿◕✿)

```
// From Future ...
val task = Task.defer(
  Task.fromFuture(Future { "effect" }))

// And back again ...
val future = task.runAsync

~~// If we want the result ...~~
~~Await.result(future, 10.seconds)~~
```

I DON'T USUALLY BLOCK THREADS, BUT WHEN I DO ...
I USE THE BLOCKCONTEXT AND SPECIFY TIMEOUTS
memegenerator.net

CANCELABLES

BECAUSE WE SHOULDN’T LEAK

CANCELABLES

```scala
package monix.eval

sealed abstract class Task[+A] {
  def runAsync(implicit s: Scheduler): CancelableFuture[A]

  def runAsync(cb: Callback[A])
    (implicit s: Scheduler): Cancelable

  def runAsync(f: Try[A] => Unit)
    (implicit s: Scheduler): Cancelable

  ???
}
```

CANCELABLES

```scala
// In monix.execution ...
trait CancelableFuture[+A]
  extends Future[A] with Cancelable

val result: CancelableFuture[String] =
  Task.evalOnce { "result" }
    .delayExecution(10.seconds)
    .runAsync

// If we change our mind ...
result.cancel()
```

CANCELABLES

```scala
def delayed[A](timespan: FiniteDuration)(f: => A) =
  Task.create[A] { (scheduler, callback) =>
    // Register a task in the thread-pool
    val cancelable = scheduler.scheduleOnce(
      timespan.length, timespan.unit,
      new Runnable {
        def run(): Unit =
          callback(Try(f))
      })

    cancelable
  }
```

CANCELABLES: SAFE FALLBACKS (1/2)

```
def chooseFirstOf[A,B](fa: Task[A], fb: Task[B]):
  Task[(A, CancelableFuture[B]) \/ (CancelableFuture[A], B)]
```

CANCELABLES: SAFE FALLBACKS (2/2)

```
val source: Task[Int] = ???
val other: Task[Int] = ???

val fallback: Task[Int] =
  other.delayExecution(5.seconds)

Task.chooseFirstOf(source, fallback).map {
  case Left(((a, futureB))) =>
    futureB.cancel()
    a
  case Right((futureA, b)) =>
    futureA.cancel()
    b
}
```

CANCELABLES: BETTER FUTURE.SEQUENCE

```
val result: Task[Seq[Int]] =
  Task.zipList(Seq(task1, task2, task3, task4))
```

On error it does not wait and cancels the unfinished ;-)

CANCELABLES: BETTER FUTURE.FIRSTCOMPLETEDOF

```
val result: Task[Int] =
  Task.chooseFirstOfList(Seq(task1, task2, task3))
```

Cancels the unfinished ;-)

THE MONAD VERSUS THE APPLICATIVE :-)

```
// Ordered operations ...
for {
  location <- locationTask
  phone <- phoneTask
  address <- addressTask
} yield {
  "Gotcha!"
}

// Potentially in parallel
Task.zip3(locationTask, phoneTask, addressTask).map {
  (location, phone, address) =>
    "Gotcha!"
}
```

RESTART, FTW

```
Task.evalAlways(Random.nextInt())
  .restartUntil(_ % 2 == 0)
```

ERROR HANDLING

"If a tree falls in a forest and no one is around to hear it, does it make a sound?"

ERROR HANDLING (1/4)

```
task.onErrorHandleWith {
  case _: TimeoutException => fallbackTask
  case ex => Task.raiseError(ex)
}
```

ERROR HANDLING (2/4)

```
task.onErrorRestart(maxRetries = 20)

task.onErrorRestartIf {
  case _: TimeoutException => true
  case _ => false
}
```

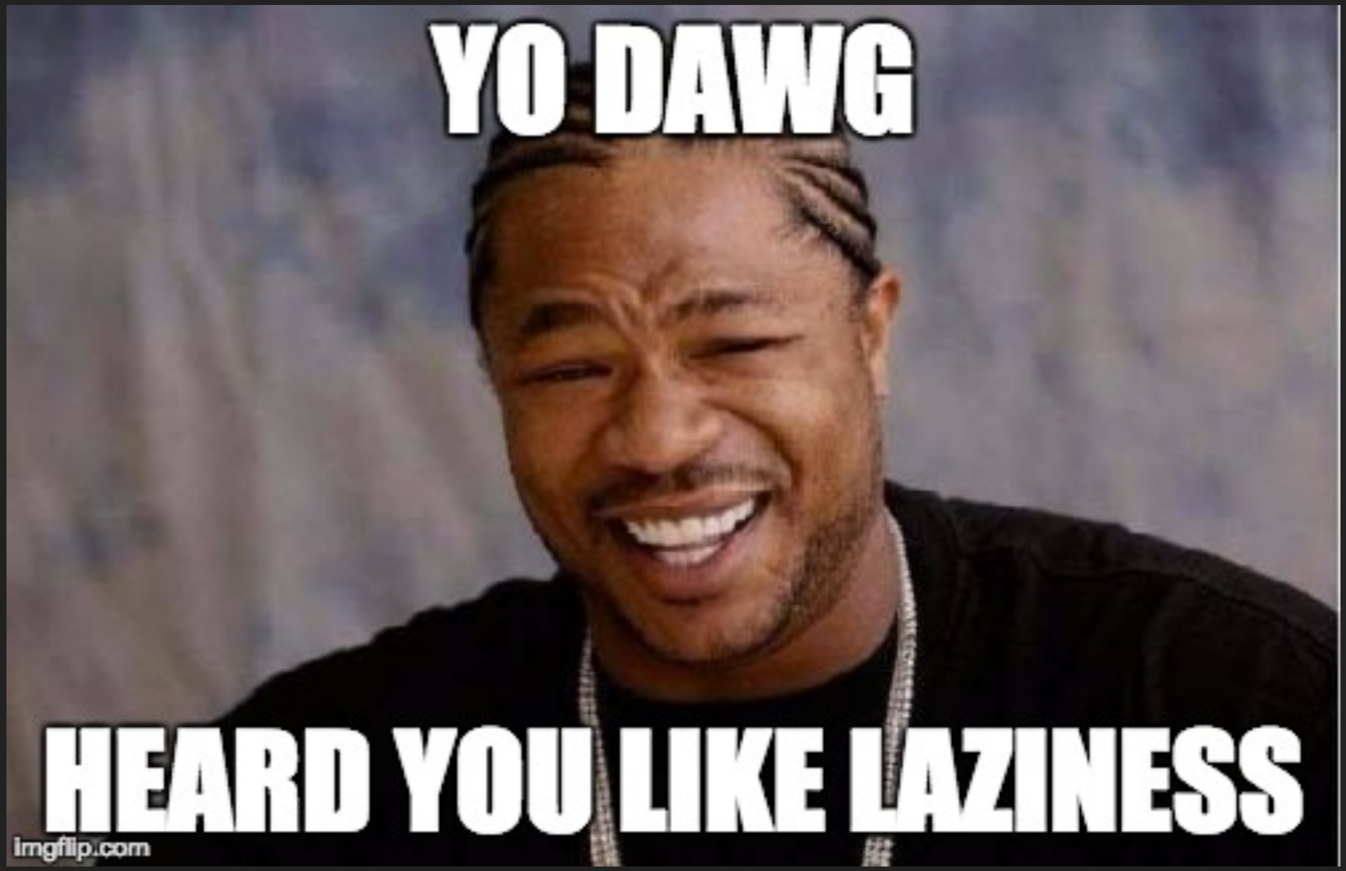
ERROR HANDLING (3/4)

```scala
def retryWithBackoff[A](source: Task[A],
  maxRetries: Int, firstDelay: FiniteDuration): Task[A] = {

  source.onErrorHandleWith {
    case ex: Exception =>
      if (maxRetries > 0)
        retryWithBackoff(source, maxRetries-1, firstDelay*2)
          .delayExecution(firstDelay)
      else
        Task.raiseError(ex)
  }
}
```

ERROR HANDLING (4/4)

```
task.timeout(10.seconds)

task.timeoutTo(10.seconds,
  Task.raiseError(new TimeoutException()))
```

IS THAT IT?



COEVAL

- *having the same age or date of origin; contemporary.*
- *something of the same era*
- *synchronous*

COEVAL

- like Task, but *only* for synchronous evaluation
- Coeval.now
- Coeval.evalOnce
- Coeval.evalAlways
- coeval.memoize

COEVAL

- replacement for **by-name** parameters
- replacement for **lazy val**
- replacement for **Function0**
- stack-safe

SYNCHRONOUS TAIL RECURSIVE LOOPS :-)

```scala
import monix.eval.Coeval

def odd(n: Int): Coeval[Boolean] =
  Coeval.evalAlways(n == 0).flatMap {
    case true => Coeval.now(false)
    case false => even(n - 1)
  }

def even(n: Int): Coeval[Boolean] =
  Coeval.evalAlways(n == 0).flatMap {
    case true => Coeval.now(true)
    case false => odd(n - 1)
  }

val result: Boolean =
  even(1000000).value
```

CONVERSION IS EASY

```
val task: Task[Int] = ???

val coeval: Coeval[Either[CancelableFuture[Int], Int]] =
  task.coeval
```

CONVERSION IS EASY

```scala
val coeval: Coeval[Int] = ???

val task: Task[Int] = coeval.task
```

EVALUATION IN SCALA

	Eager	Lazy
Synchronous	A	() => A
		Coeval[A]
Asynchronous	(A => Unit) => Unit	(A => Unit) => Unit
	Future[A]	Task[A]

STREAMS? (1/4)

```
sealed abstract class ConsStream[+A]

case class Next[A](head: A, tail: ConsStream[A])
  extends ConsStream[A]

case class Error(ex: Throwable)
  extends ConsStream[Nothing]
case object Empty
  extends ConsStream[Nothing]
```

STREAMS? (2/4)

```
sealed abstract class ConsStream[+A]

case class Next[A](head: A, tail: Coeval[ConsStream[A]])
  extends ConsStream[A]

case class Error(ex: Throwable)
  extends ConsStream[Nothing]
case object Empty
  extends ConsStream[Nothing]
```

STREAMS? (3/4)

```
sealed abstract class ConsStream[+A]

case class Next[A](head: A, tail: Task[ConsStream[A]])
  extends ConsStream[A]

case class Error(ex: Throwable)
  extends ConsStream[Nothing]
case object Empty
  extends ConsStream[Nothing]
```

STREAMS? (4/4)

```
import monix.types.Evaluable

sealed abstract class ConsStream[+A, F[_] : Evaluable]

case class Next[A, F[_] : Evaluable]
  (head: A, tail: F[ConsStream[A,F]])
  extends ConsStream[A,F]

case class Error[F[_] : Evaluable](ex: Throwable)
  extends ConsStream[Nothing,F]
case class Empty[F[_] : Evaluable]()
  extends ConsStream[Nothing,F]
```

MONIX.IO

QUESTIONS?