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 (see typelevel.org)
- 2.3.0
- See: monix.io

WHAT'S WRONG WITH THE FUTURE[A]?

DELAYING THE FUTURE[A]

- Side-effects handling
- Throughput
- Error handling

MONIX TASK

A type that:

- Describes lazy or (possibly) async computations
- Is effective at controlling side-effects
- Is effective at dealing with concurrency

MONIX TASK

A type that:

- Describes lazy or (possibly) async computations
- Is effective at controlling side-effects
- Is effective at dealing with concurrency

MONIX TASK

A type that:

- Describes lazy or (possibly) async computations
- Is effective at controlling side-effects
- Is effective at dealing with concurrency

MONIX TASK

Inspired by:

- Scalaz Task (scalaz.concurrent.Task)
- Haskel's IO

Alternatives:

- FS2 Task
- cats.effect.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]**

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)
}
```

MONIX 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.eval { println("effect"); "always" }

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

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

MEMOIZATION (1/4)

```scala
import monix.execution.atomic.Atomic

val task1 = {
  val effect = Atomic(0)
  Task.evalOnce(effect.incrementAndGet())
}

val task2 = {
  val effect = Atomic(0)
  Task.eval(effect.incrementAndGet()).memoize
}
```

MEMOIZATION (2/4)

`task.memoize` vs `task.runAsync`

MEMOIZATION (3/4)

```
val effect = Atomic(0)

val source = Task.eval {
  val current = effect.incrementAndGet()
  if (current ⩾ 3) current
  else throw new RuntimeException("dummy")
}

source.memoizeOnSuccess
```

MEMOIZATION (4/4)

- **memoizeOnSuccess** cannot be done with Future
- Task can do it because Task is a function ;-)

TAIL RECURSIVE LOOPS (1/4)

```scala
@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)

```
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.eval(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)

```scala
// 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)
```

NO BOOBY TRAPS

NO BOOBY TRAPS (1/2)

```scala
def signal[A](f: ⇒ A): Task[A] =
  Task.async { (_, callback) ⇒
    callback.onSuccess(f)
    Cancelable.empty
  }

// Look Ma, no Stack Overflows
def loop(n: Int, acc: Int): Task[Int] =
  Task.now(n).flatMap { x ⇒
    if (x ⩽ 0) Task.now(acc)
    else loop(x-1, acc+x)
  }
```

NO BOOBY TRAPS (2/2)

```scala
// Look Ma, no Stack Overflows
def all[A](list: List[Task[A]]): Task[List[A]] = {
  val initial = Task.now(List.empty[A])

  list.foldLeft(initial) { (acc, e) ⇒
    Task.mapBoth(e, acc)(_ :: _)
  }
}
```

SCHEDULER

SCHEDULER (1/4)

```scala
package monix.execution

trait Cancelable {
  def cancel(): Unit
}
```

```scala
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
}
```

SCHEDULER (2/4)

```
import monix.execution.CancelableFuture
import monix.execution.Scheduler.Implicits.global

val f: CancelableFuture[String] =
  Task("hello").runAsync(global)
```

SCHEDULER (3/4)

```scala
import monix.execution.Scheduler

val effect = Atomic(0)
val io = Scheduler.io("my-io")

Task(effect.incrementAndGet())
  .executeOn(io)
  .asyncBoundary
  .map(_ + 1)
```

SCHEDULER (4/4)

```
import monix.execution.Scheduler

val effect = Atomic(0)
val io = Scheduler.io("my-io")
val computation = Scheduler.computation()

Task(effect.incrementAndGet())
  .executeOn(io)
  .asyncBoundary(computation)
  .map(_ + 1)
```

EXECUTION MODEL

EXECUTION MODEL

- in batches, by default
  (fair, reasonable performance)
- always asynchronous
  (fair, like Scala's Future)
- preferably synchronous
  (unfair, like Scalaz's Task)

EXECUTION MODEL: BATCHED

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

implicit val scheduler =
  Scheduler.computation(
    parallelism = 4,
    executionModel=BatchedExecution(batchSize=1024)
  )
```

EXECUTION MODEL: ALWAYS ASYNC

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

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

EXECUTION MODEL: PREFER SYNCHRONOUS

```
import monix.execution._
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 eval[A](d: FiniteDuration, f: ⇒ A): Task[A] =
  Task.create { (scheduler, callback) ⇒
    // Execution
    val cancelable = scheduler.scheduleOnce(d) {
      try callback.onSuccess(f) catch {
        case NonFatal(e) ⇒
          callback.onError(e)
      }
    }
    // For early termination
    cancelable
  }
```

REAL ASYNCHRONY

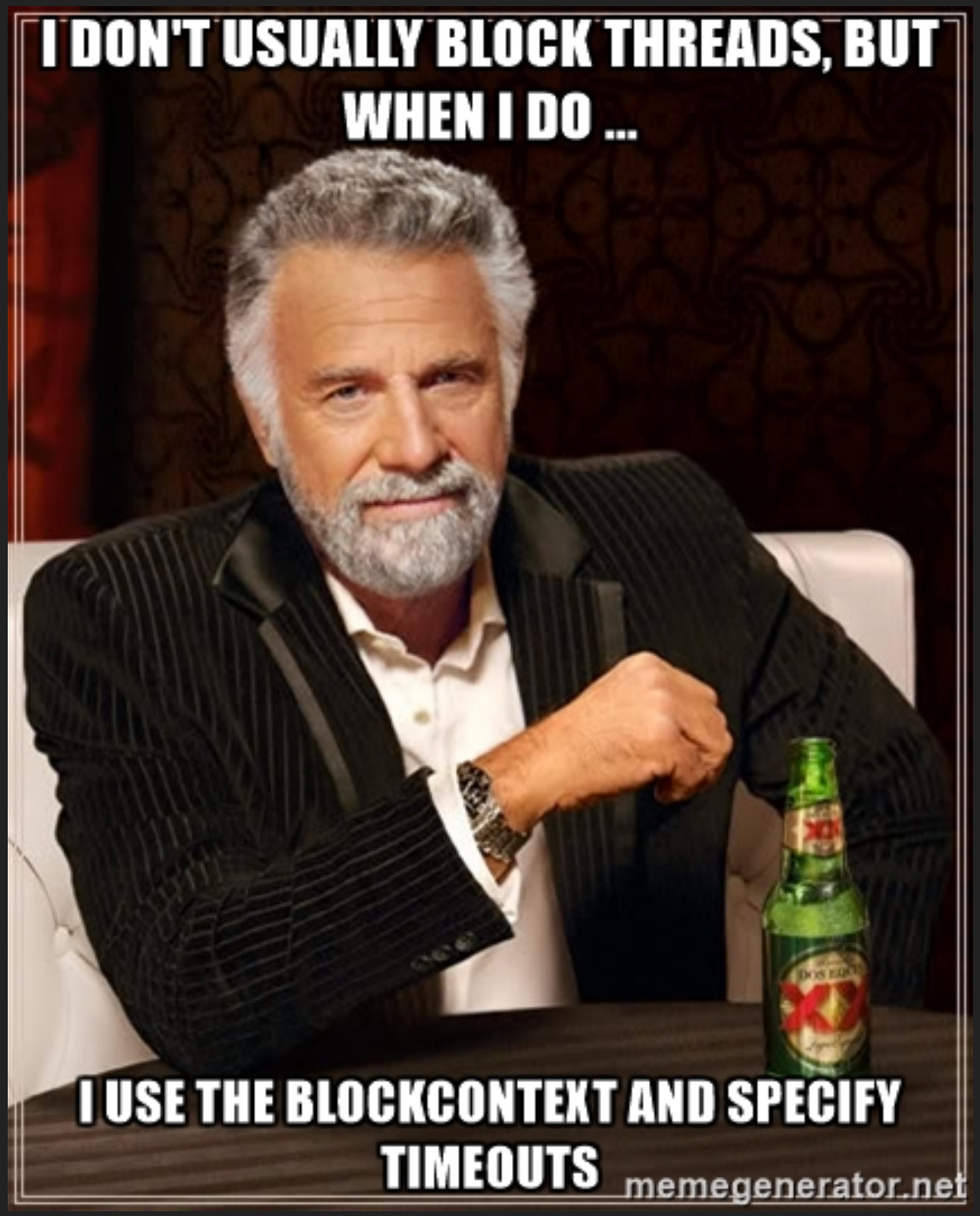
```
def fromFuture[A](future: Future[A]): Task[A] =
  Task.create { (scheduler, callback) =>
    implicit val ec = scheduler
    // Waiting ...
    future.onComplete {
      case Success(v) =>
        callback.onSuccess(v)
      case Failure(ex) =>
        callback.onError(ex)
    }
    // Futures can't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

```scala
// 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

FUTURE INTEROP

FUTURE INTEROP (1/3)

```
val effect = Atomic(0)

def increment()
  (implicit ec: ExecutionContext): Future[Int] =
  Future(effect.incrementAndGet())
```

FUTURE INTEROP (2/3)

```scala
val effect = Atomic(0)

def increment()
  (implicit ec: ExecutionContext): Future[Int] =
  Future(effect.incrementAndGet())

def incrementTask()
  (implicit ec: ExecutionContext): Task[Int] =
  Task.deferFuture(increment())
```

FUTURE INTEROP (3/3)

```scala
val effect = Atomic(0)

def increment()
  (implicit ec: ExecutionContext): Future[Int] =
  Future(effect.incrementAndGet())

// Look Ma, no implicit ExecutionContext
val task = Task.deferFutureAction { implicit ec ⇒
  increment()
}
```

CANCELABLES

BECAUSE WE SHOULDN'T LEAK

CANCELABLES

```
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 runOnComplete(f: Try[A] => Unit)
    (implicit s: Scheduler): Cancelable

  ???
}
```

CANCELABLES

```
// 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

```
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[Either[(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 ;-)

CANCELABLES: LOOPS

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

  loop(cycles, 0, 1)
    .executeWithOptions(_.enableAutoCancelableRunLoops)
}
```

CANCELABLES

- **cancel** is a concurrent action (with the Task's execution)
- **NOT cancelable** by default (e.g. bring your own booze)

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)

```
def retryBackoff[A](source: Task[A])
  (maxRetries: Int, delay: FiniteDuration): Task[A] = {

  source.onErrorHandleWith { ex ⇒
    if (maxRetries > 0)
      retryBackoff(source)(maxRetries - 1, delay * 2)
        .delayExecution(delay)
    else
      Task.raiseError(ex)
  }
}
```

ERROR HANDLING (4/4)

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

PRIMITIVES

ASYNCHRONOUS SEMAPHORE

```scala
import monix.eval.TaskSemaphore

val semaphore =
  TaskSemaphore(maxParallelism = 16)

val request =
  Task("response").delayExecution(1.second)

semaphore.greenLight(request)
```

CIRCUIT BREAKER

```scala
val circuitBreaker = TaskCircuitBreaker(
  maxFailures = 5,
  resetTimeout = 10.seconds
)

// ...
val problematic = Task {
  val nr = util.Random.nextInt()
  if (nr % 2 == 0) nr else
    throw new RuntimeException("dummy")
}

val task = circuitBreaker.protect(problematic)
```

MVAR

Behaves like a **BlockingQueue(size = 1)**

```
import monix.eval.MVar

val state = MVar.empty[Int]

// Blocks until MVar is empty!
val producer: Task[Unit] =
  state.put(1)

// Blocks until a value is available
val consumer: Task[Int] =
  state.take
```

MVAR – USE CASES

- synchronized mutable vars
- asynchronous locks
- simple Producer/Consumer channels

QUESTIONS?

monix.io

@monix

@monix

alexn.org

@alexelcu

@alexandru