Future vs. Monix Task

https://github.com/hermannhueck/future-vs-monix-task

Abstract

scala.concurrent.Future is familiar to most Scala devs.

This presentation first talks about referential transparency and the IO Monad in general. (Monix *Task* is an impl of the IO Monad.)

Then it compares *Future* Monix 3.x *Task* with their Pros and Cons.

Interop with *Future*: As Scala's *Future* is used in many environments and libraries, we look at the conversion from *Task* to *Future* and - vice versa - from *Future* to *Task*.

I will also take a look at *Task* evaluation, cancelation and memoization as well as tail recursive loops and asynchronous boundaries.

The presentation will include a comparative discussion on *ExecutionContext* (required for *Future*) and *Scheduler* (required for *Task*, but only to run it).

Often recurring on the valuable Monix *Task* doumentation at https://monix.io/docs/3x/eval/task.html the presentation can also be seen as an introduction to Monix *Task*.

Agenda

1. Referential Transparency
2. IO Monad (My Impl)
3. Is Future referentially transparent?
4. What is Monix Task?
5. Task Evaluation
6. Task Cancelation
7. Memoization
8. Task Builders
9. Converting Future to Task
10. Task as Monad
11. Tail Recursive Loops
12. Async Boundaries
13. Schedulers
14. Error Handling
15. Races
16. Delaying a Task
17. Parallelism (cats.Parallel)
18. TaskApp
19. CompletableFuture
20. Resource Management
21. Resources

1. Referential Transparency

Referential Transparency

An expression is called referentially transparent if it can be replaced with its corresponding value without changing the program's behavior.

This requires that the expression is ***pure***, that is to say the expression value must be the same for the same inputs and its evaluation must have ***no side effects***.

https://en.wikipedia.org/wiki/Referential_transparency

Referential Transparency Benefits

- (Equational) Reasoning about code
- Refactoring is easier
- Testing is easier
- Separate pure code from impure code
- Potential compiler optimizations (more in Haskell than in Scala)
  (e.g. memoization, parallelisation, compute expressions at compile time)

"What Referential Transparency can do for you"
Talk by Luka Jacobowitz at ScalaIO 2017
https://www.youtube.com/watch?v=X-cEGEJMx_4

2. IO Monad (My Impl)

Impure IO Program with side effects

```scala
// impure program
def program(): Unit = {
  print("Welcome to Scala!  What's your name?   ")
  val name = scala.io.StdIn.readLine
  println(s"Well hello, $name!")
}
```

Impure IO Program with side effects

```scala
// impure program
def program(): Unit = {
  print("Welcome to Scala!  What's your name?   ")
  val name = scala.io.StdIn.readLine
  println(s"Well hello, $name!")
}
```

```scala
program()
```

Impure IO Program <u>with</u> side effects

```scala
// impure program
def program(): Unit = {
  print("Welcome to Scala!  What's your name?   ")
  val name = scala.io.StdIn.readLine
  println(s"Well hello, $name!")
}
```

```scala
program()
```

- Whenever a method or a function returns *Unit* it is ***impure*** (or it is a noop). It's intension is to produce a side effect.
- A ***pure*** function always returns a value of some type (and doesn't produce a side effect inside).

Pure IO Program without side effects

```scala
// pure program
val program: () => Unit =  // () => Unit  is syntactic sugar for:  Function0[Unit]
  () => {
    print("Welcome to Scala!  What's your name?   ")
    val name = scala.io.StdIn.readLine
    println(s"Well hello, $name!")
  }
```

Pure IO Program without side effects

```scala
// pure program
val program: () => Unit =  // () => Unit  is syntactic sugar for:  Function0[Unit]
  () => {
    print("Welcome to Scala!  What's your name?   ")
    val name = scala.io.StdIn.readLine
    println(s"Well hello, $name!")
  }
```

```scala
program()    // producing the side effects "at the end of the world"
```

Pure IO Program <u>without</u> side effects

```
// pure program
val program: () => Unit =  // () => Unit  is syntactic sugar for:  Function0[Unit]
  () => {
    print("Welcome to Scala!  What's your name?   ")
    val name = scala.io.StdIn.readLine
    println(s"Well hello, $name!")
  }
```

```
program()    // producing the side effects "at the end of the world"
```

- Make the program a function returning *Unit*: *Function0[Unit]*
- Free of side effects in it's definition
- Produces side effects only when run (at the end of the world)

Wrap Function0[A] in a case class

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

Wrap Function0[A] in a case class

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

```
// pure program
val program: IO[Unit] = IO {
  () => {
    print("Welcome to Scala!  What's your name?   ")
    val name = scala.io.StdIn.readLine
    println(s"Well hello, $name!")
  }
}
```

Wrap Function0[A] in a case class

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

```
// pure program
val program: IO[Unit] = IO {
  () => {
    print("Welcome to Scala!  What's your name?   ")
    val name = scala.io.StdIn.readLine
    println(s"Well hello, $name!")
  }
}
```

```
program.run()    // producing the side effects "at the end of the world"
```

Wrap Function0[A] in a case class

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

```
// pure program
val program: IO[Unit] = IO {
  () => {
    print("Welcome to Scala!  What's your name?   ")
    val name = scala.io.StdIn.readLine
    println(s"Well hello, $name!")
  }
}
```

```
program.run()    // producing the side effects "at the end of the world"
```

- *IO[A]* wraps a *Function0[A]* in a case class.
- This is useful to implement further extensions on that case class.

IO#map and *IO#flatMap*

```scala
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = IO { () => f(run()) }
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
```

IO#map and *IO#flatMap*

```
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = IO { () => f(run()) }
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
```

```
val program: IO[Unit] = for {
  _       <- IO { () => print(s"Welcome to Scala!  What's your name?   ") }
  name    <- IO { () => scala.io.StdIn.readLine }
  _       <- IO { () => println(s"Well hello, $name!") }
} yield ()
```

IO#map and *IO#flatMap*

```scala
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = IO { () => f(run()) }
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
```

```scala
val program: IO[Unit] = for {
  _       <- IO { () => print(s"Welcome to Scala!  What's your name?   ") }
  name    <- IO { () => scala.io.StdIn.readLine }
  _       <- IO { () => println(s"Well hello, $name!") }
} yield ()
```

```scala
program.run()    // producing the side effects "at the end of the world"
```

IO#map and *IO#flatMap*

```scala
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = IO { () => f(run()) }
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
```

```scala
val program: IO[Unit] = for {
  _       <- IO { () => print(s"Welcome to Scala!  What's your name?   ") }
  name    <- IO { () => scala.io.StdIn.readLine }
  _       <- IO { () => println(s"Well hello, $name!") }
} yield ()
```

```scala
program.run()    // producing the side effects "at the end of the world"
```

- With *map* and *flatMap* *IO[A]* is monadic (but it is not yet a Monad).
- *IO* is ready for for-comprehensions.
- This allows the composition of programs from smaller components.

Companion object *IO* with *pure* and *eval*

```
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
object IO {
  def pure[A](value: A): IO[A] = IO { () => value }         // eager
  def eval[A](thunk: => A): IO[A] = IO { () => thunk }      // lazy
}
```

Companion object *IO* with *pure* and *eval*

```
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
object IO {
  def pure[A](value: A): IO[A] = IO { () => value }         // eager
  def eval[A](thunk: => A): IO[A] = IO { () => thunk }      // lazy
}
```

```
val program: IO[Unit] = for {
  welcome <- IO.pure("Welcome to Scala!")
  _       <- IO.eval { print(s"$welcome  What's your name?   ") }
  name    <- IO.eval { scala.io.StdIn.readLine }         // simpler with IO.eval
  _       <- IO.eval { println(s"Well hello, $name!") }
} yield ()
```

Companion object *IO* with *pure* and *eval*

```
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
object IO {
  def pure[A](value: A): IO[A] = IO { () => value }         // eager
  def eval[A](thunk: => A): IO[A] = IO { () => thunk }      // lazy
}
```

```
val program: IO[Unit] = for {
  welcome <- IO.pure("Welcome to Scala!")
  _       <- IO.eval { print(s"$welcome  What's your name?   ") }
  name    <- IO.eval { scala.io.StdIn.readLine }        // simpler with IO.eval
  _       <- IO.eval { println(s"Well hello, $name!") }
} yield ()
```

```
program.run()    // producing the side effects "at the end of the world"
```

Companion object *IO* with *pure* and *eval*

```
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
object IO {
  def pure[A](value: A): IO[A] = IO { () => value }         // eager
  def eval[A](thunk: => A): IO[A] = IO { () => thunk }      // lazy
}
```

```
val program: IO[Unit] = for {
  welcome <- IO.pure("Welcome to Scala!")
  _       <- IO.eval { print(s"$welcome  What's your name?   ") }
  name    <- IO.eval { scala.io.StdIn.readLine }        // simpler with IO.eval
  _       <- IO.eval { println(s"Well hello, $name!") }
} yield ()
```

```
program.run()    // producing the side effects "at the end of the world"
```

- *IO.pure* is eager and accepts a pure value.
- *IO.eval* is lazy and accepts a computation.
- *map* can be written in terms of *flatMap* and *pure*.

Monad instance for *IO*

```scala
sealed trait IO[+A] {

  def flatMap[B](f: A => IO[B]): IO[B] = IO { f(run()).run() }
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
}

object IO {

  implicit val ioMonad: Monad[IO] = new Monad[IO] {
    override def pure[A](value: A): IO[A] = IO.pure(value)
    override def flatMap[A, B](fa: IO[A])(f: A => IO[B]): IO[B] = fa flatMap f
  }
}
```

Monad instance for *IO*

```
sealed trait IO[+A] {

  def flatMap[B](f: A => IO[B]): IO[B] = IO { f(run()).run() }
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
}

object IO {

  implicit val ioMonad: Monad[IO] = new Monad[IO] {
    override def pure[A](value: A): IO[A] = IO.pure(value)
    override def flatMap[A, B](fa: IO[A])(f: A => IO[B]): IO[B] = fa flatMap f
  }
}
```

- Monad instance defined in companion object (implicit scope)

Computations that abstract over HKT: *F[_]: Monad*

```scala
import scala.language.higherKinds
import cats.syntax.flatMap._
import cats.syntax.functor._

def sumF[F[_]: Monad](from: Int, to: Int): F[Int] =
  Monad[F].pure { sumOfRange(from, to) }

def fibonacciF[F[_]: Monad](num: Int): F[BigInt] =
  Monad[F].pure { fibonacci(num) }

def factorialF[F[_]: Monad](num: Int): F[BigInt] =
  Monad[F].pure { factorial(num) }

def computeF[F[_]: Monad](from: Int, to: Int): F[BigInt] =
  for {
    x <- sumF(from, to)
    y <- fibonacciF(x)
    z <- factorialF(y.intValue)
  } yield z
```

- This code can be used with *IO* or any other Monad.

- **Reify *F[_] : Monad* with *IO***

```scala
val io: IO[BigInt] = computeF[IO](1, 4)
val result: BigInt = io.run()
println(result)                                                   //=> 6227020800
```

- **Reify *F[_] : Monad* with *IO***

```
val io: IO[BigInt] = computeF[IO](1, 4)
val result: BigInt = io.run()
println(result)                                                 //=> 6227020800
```

- **Reify *F[_] : Monad* with *cats.Id***

```
val result: cats.Id[BigInt] = computeF[cats.Id](1, 4)
println(result)                                                 //=> 6227020800
```

- Reify *F[_] : Monad* with *IO*

```
val io: IO[BigInt] = computeF[IO](1, 4)
val result: BigInt = io.run()
println(result)                                                         //=> 6227020800
```

- Reify *F[_] : Monad* with *cats.Id*

```
val result: cats.Id[BigInt] = computeF[cats.Id](1, 4)
println(result)                                                         //=> 6227020800
```

- Reify *F[_] : Monad* with *Option*

```
import cats.instances.option._

val maybeResult: Option[BigInt] = computeF[Option](1, 4)
maybeResult foreach println                                             //=> 6227020800
```

- **Reify *F[_] : Monad* with *IO***

```
val io: IO[BigInt] = computeF[IO](1, 4)
val result: BigInt = io.run()
println(result)                                                   //=> 6227020800
```

- **Reify *F[_] : Monad* with *cats.Id***

```
val result: cats.Id[BigInt] = computeF[cats.Id](1, 4)
println(result)                                                   //=> 6227020800
```

- **Reify *F[_] : Monad* with *Option***

```
import cats.instances.option._

val maybeResult: Option[BigInt] = computeF[Option](1, 4)
maybeResult foreach println                                       //=> 6227020800
```

- **Reify *F[_] : Monad* with *Future***

```
import scala.concurrent.{Future, ExecutionContext}
import ExecutionContext.Implicits.global
import cats.instances.future._

val future: Future[BigInt] = computeF[Future](1, 4)
future foreach println                                            //=> 6227020800
```

Sync and async run* methods

```
case class IO[A](run: () => A) {

  // ----- sync run* methods

  def runToTry: Try[A] = Try { run() }

  def runToEither: Either[Throwable, A] = runToTry.toEither

  // ----- async run* methods (all take an EC)

  def runToFuture(implicit ec: ExecutionContext): Future[A] =
    Future { run() }

  def runOnComplete(callback: Try[A] => Unit)
                                    (implicit ec: ExecutionContext): Unit =
    runToFuture onComplete callback

  def runAsync(callback: Either[Throwable, A] => Unit)
                                    (implicit ec: ExecutionContext): Unit =
    runOnComplete(tryy => callback(tryy.toEither))
}
```

The extended version of my IO Monad impl

can be found here:

https://github.com/hermannhueck/implementing-io-monad

Monix Task is the IO Monad

… a bit more sophisticated than my implementation.

;-)

But it can also be seen as …

a lazy Future.

3. Is *Future* referentially transparent?

Is *Future* referentially transparent?

```scala
val future1: Future[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  val future: Future[Int] = Future { atomicInt.incrementAndGet }
  for {
    x <- future
    y <- future
  } yield (x, y)
}

future1 onComplete println     // Success((1,1))
```

```scala
// same as future1, but inlined
val future2: Future[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  for {
    x <- Future { atomicInt.incrementAndGet }
    y <- Future { atomicInt.incrementAndGet }
  } yield (x, y)
}

future2 onComplete println     // Success((1,2))    <-- not the same result
```

Is *Future* referentially transparent?

```
val future1: Future[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  val future: Future[Int] = Future { atomicInt.incrementAndGet }
  for {
    x <- future
    y <- future
  } yield (x, y)
}

future1 onComplete println     // Success((1,1))
```

```
// same as future1, but inlined
val future2: Future[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  for {
    x <- Future { atomicInt.incrementAndGet }
    y <- Future { atomicInt.incrementAndGet }
  } yield (x, y)
}

future2 onComplete println     // Success((1,2))    <-- not the same result
```

No!

Is my IO Monad referentially transparent?

```
val io1: IO[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  val io: IO[Int] = IO.eval { atomicInt.incrementAndGet }
  for {
    x <- io
    y <- io
  } yield (x, y)
}

io1.runToFuture onComplete println     // Success((1,2))
```

```
// same as io1, but inlined
val io2: IO[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  for {
    x <- IO.eval { atomicInt.incrementAndGet }
    y <- IO.eval { atomicInt.incrementAndGet }
  } yield (x, y)
}

io2.runToFuture onComplete println     // Success((1,2))    <-- same result
```

Is my IO Monad referentially transparent?

```
val io1: IO[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  val io: IO[Int] = IO.eval { atomicInt.incrementAndGet }
  for {
    x <- io
    y <- io
  } yield (x, y)
}

io1.runToFuture onComplete println     // Success((1,2))
```

```
// same as io1, but inlined
val io2: IO[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  for {
    x <- IO.eval { atomicInt.incrementAndGet }
    y <- IO.eval { atomicInt.incrementAndGet }
  } yield (x, y)
}

io2.runToFuture onComplete println     // Success((1,2))    <-- same result
```

Yes!

Is Monix *Task* referentially transparent?

```
val task1: Task[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  val task: Task[Int] = Task { atomicInt.incrementAndGet }
  for {
    x <- task
    y <- task
  } yield (x, y)
}

task1 runAsync println     // Success((1,2))
```

```
// same as task1, but inlined
val task2: Task[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  for {
    x <- Task { atomicInt.incrementAndGet }
    y <- Task { atomicInt.incrementAndGet }
  } yield (x, y)
}

task2 runAsync println     // Success((1,2))    <-- same result
```

Is Monix *Task* referentially transparent?

```scala
val task1: Task[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  val task: Task[Int] = Task { atomicInt.incrementAndGet }
  for {
    x <- task
    y <- task
  } yield (x, y)
}

task1 runAsync println     // Success((1,2))
```

```scala
// same as task1, but inlined
val task2: Task[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  for {
    x <- Task { atomicInt.incrementAndGet }
    y <- Task { atomicInt.incrementAndGet }
  } yield (x, y)
}

task2 runAsync println     // Success((1,2))    <-- same result
```

Yes!

4. What is Monix Task?

Monix 3.x

- library for asynchronous, reactive computations in Scala
- available for the JVM and ScalaJS
- dependent on: *cats* and *cats-effect*
- Web site: https://monix.io

Monix 3.x abstractions:

- *monix.execution.**Scheduler***: Scheduler that extends *scala.concurrrent.ExecutionContext*
- *monix.eval.**Task***: abstraction for async effects
- *monix.eval.**Coeval***: abstraction for sync effects
- *monix.reactive.**Observable***: push-based streaming library (compatible with the *reactivestreams.org* spec)
- *monix.tail.**Iterant***: pull-based streaming library (compatible with the *reactivestreams.org* spec)

Monix Task

- inspired by: Haskell's IO Monad, Scalaz Task
- implementation of the IO Monad for Scala
- lazy and referentially transparent alternative for *scala.concurrrent.Future*
- *Future* is a value-wannabe. It starts executing when created.
- *Task* is a function (a wrapped *Function0[A]*). It is just a description of a (possibly asynchronous) computation. It doesn't run unless explicitly told to do so.
- good interop with *Future*: easy to turn *Future* to *Task* or *Task* to *Future*

Comparison of Features

	Future	**Task**
Evaluation	eager / strict	lazy / non-strict
Control of side effects	no	yes
Memoization	always	possible, default: no
Referential Transparency	no	yes
Control of side effects	no	yes
Supports cancelation	no	yes
Supports blocking	yes	no (possible via Future)
Supports sync execution	no	yes
Runs on an other thread	always	not necessarily
Supports green threads	no	yes
Requires implicit ...	ExecutionContext (for nearly every method)	Scheduler (only when run)
Stack Safety	no	yes (due to trampolining)
Supports ScalaJS	yes	yes

5. Task Evaluation

Evaluation

	Eager	**Lazy**
Synchronous	A	() => A
		Eval[A], Coeval[A]
Asynchronous	(A => Unit) => Unit	() => (A => Unit) => Unit
	Future[A]	Task[A], IO[A]

build.sbt

```
libraryDependencies += "io.monix" %% "monix" % "3.0.0-RC2"
```

Running a Task requires an implicit *Scheduler*

Running a Task requires an implicit *Scheduler*

Either ...

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

or create an implicit instance ...

```
import monix.execution.Scheduler.global
implicit val scheduler: Scheduler = Scheduler.global
```

Running a Task requires an implicit *Scheduler*

Either ...

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

or create an implicit instance ...

```
import monix.execution.Scheduler.global
implicit val scheduler: Scheduler = Scheduler.global
```

- *Scheduler* is a subclass of *ExecutionContext*, hence is serves as well as an *ExecutionContext* for *Future*.

Running a Task requires an implicit *Scheduler*

Either ...

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

or create an implicit instance ...

```
import monix.execution.Scheduler.global
implicit val scheduler: Scheduler = Scheduler.global
```

- *Scheduler* is a subclass of *ExecutionContext*, hence is serves as well as an *ExecutionContext* for *Future*.
- The global *Scheduler* is backed by the global *ExecutionContext* which is backed by Java's *ForkJoinPool*.

Running a Task requires an implicit *Scheduler*

Either ...

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

or create an implicit instance ...

```
import monix.execution.Scheduler.global
implicit val scheduler: Scheduler = Scheduler.global
```

- *Scheduler* is a subclass of *ExecutionContext*, hence is serves as well as an *ExecutionContext* for *Future*.
- The global *Scheduler* is backed by the global *ExecutionContext* which is backed by Java's *ForkJoinPool*.
- *Future* requires an *ExecutionContext* for almost every method (*apply*, *map*, *flatMap*, *onComplete*, etc.) whereas ...

Running a Task requires an implicit *Scheduler*

Either ...

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

or create an implicit instance ...

```
import monix.execution.Scheduler.global
implicit val scheduler: Scheduler = Scheduler.global
```

- *Scheduler* is a subclass of *ExecutionContext*, hence is serves as well as an *ExecutionContext* for *Future*.
- The global *Scheduler* is backed by the global *ExecutionContext* which is backed by Java's *ForkJoinPool*.
- *Future* requires an *ExecutionContext* for almost every method (*apply*, *map*, *flatMap*, *onComplete*, etc.) whereas ...
- *Task* (due to it's laziness) requires a *Scheduler* only when run.

Running *Future* and *Task*

Running *Future* and *Task*

- *Future* is eager. It starts running when you create it.

```
implicit val ec: ExecutionContext = ExecutionContext.global
                                                    // needs EC to create Future
val task: Future[Int] = Future { compute }          // starts running HERE!

val callback: Try[Int] => Unit = {                  // Try based callback
  case Success(result) => println(s"result = $result")
  case Failure(ex) => println(s"ERROR: ${ex.getMessage}")
}

task onComplete callback
```

Running *Future* and *Task*

- *Future* is eager. It starts running when you create it.

```
implicit val ec: ExecutionContext = ExecutionContext.global
                                                        // needs EC to create Future
val task: Future[Int] = Future { compute }              // starts running HERE!

val callback: Try[Int] => Unit = {                      // Try based callback
  case Success(result) => println(s"result = $result")
  case Failure(ex) => println(s"ERROR: ${ex.getMessage}")
}

task onComplete callback
```

- *Task* is a function, hence lazy. By it's creation nothing is run.

```
val task: Task[Int] = Task { compute }

val callback: Either[Throwable, Int] => Unit = {        // Either based callback
  case Right(result) => println(s"result = $result")
  case Left(ex) => println(s"ERROR: ${ex.getMessage}")
}

implicit val scheduler: Scheduler = Scheduler.global
                                                        // needs Scheduler to run Task
task runAsync callback                                  // starts running HERE!
```

Two flavours of *Callback*

Two flavours of *Callback*

- Both are *Either* based.

```scala
val callback: Either[Throwable, Int] => Unit = {
  case Right(result) => println(s"result = $result")
  case Left(ex) => println(s"ERROR: ${ex.getMessage}")
}
```

```scala
val callback: Callback[Throwable, Int] = new Callback[Throwable, Int] {
  def onSuccess(result: Int): Unit = println(s"result = $result")
  def onError(ex: Throwable): Unit = println(s"ERROR: ${ex.getMessage}")
}
```

Two flavours of *Callback*

- Both are *Either* based.

```
val callback: Either[Throwable, Int] => Unit = {
  case Right(result) => println(s"result = $result")
  case Left(ex) => println(s"ERROR: ${ex.getMessage}")
}
```

```
val callback: Callback[Throwable, Int] = new Callback[Throwable, Int] {
  def onSuccess(result: Int): Unit = println(s"result = $result")
  def onError(ex: Throwable): Unit = println(s"ERROR: ${ex.getMessage}")
}
```

- A *Callback* is a subclass of a function of type (Either[E, A] => Unit).

```
package monix.execution

abstract class Callback[-E, -A] extends (Either[E, A] => Unit) {

  def onSuccess(value: A): Unit
  def onError(e: E): Unit
  // ...
}
```

Converting *Task* to *Future*

- *Task#runToFuture* turns a *Task* into a *Future* and runs it.
- A *Scheduler* is required in implicit scope.

Converting *Task* to *Future*

- *Task#runToFuture* turns a *Task* into a *Future* and runs it.
- A *Scheduler* is required in implicit scope.

```scala
val task: Task[Int] = Task {
  compute
}

implicit val scheduler: Scheduler = Scheduler.global

val future: Future[Int] = task.runToFuture

val callback: Try[Int] => Unit = {                    // Try based callback
  case Success(result) => println(s"result = $result")
  case Failure(ex) => println(s"ERROR: ${ex.getMessage}")
}

future onComplete callback
```

Task cannot be blocked (directly)

- The API of *Task* doesn't support any means to wait for a result.
- But (if really needed) we can convert *Task* to *Future* and wait for the Future's result to become available.

Task cannot be blocked (directly)

- The API of *Task* doesn't support any means to wait for a result.
- But (if really needed) we can convert *Task* to *Future* and wait for the Future's result to become available.

```scala
val task: Task[Int] = Task {
  compute
}.delayExecution(1.second)

implicit val scheduler: Scheduler = Scheduler.global

val future: Future[Int] = task.runToFuture

val result = Await.result(future, 3.seconds)
println(s"result = $result")
```

Task#foreach

- corresponds to *Future#foreach
- *Task#foreach* runs the *Task*.
- It processes the result, but ignores possible errors.

Task#foreach

- corresponds to *Future#foreach
- *Task#foreach* runs the *Task*.
- It processes the result, but ignores possible errors.

```
val task: Task[Int] = Task {
  compute                       // returns an Int or throws an exception
}

implicit val scheduler: Scheduler = Scheduler.global

task foreach { result => println(s"result = $result") }

// prints computed value in case of success
// prints nothing in case of error
```

- Use *foreach* only if the *Task* is guaranteed not to throw an *Exception*.

Task#failed

- corresponds to *Future#failed
- Returns a failed projection of the current *Task*.
- If the source fails we get a *Task[Throwable]* containing the error.
  If the source *Task* succeeds we get a *NoSuchElementException*.

Task#failed

- corresponds to *Future#failed
- Returns a failed projection of the current *Task*.
- If the source fails we get a *Task[Throwable]* containing the error.
  If the source *Task* succeeds we get a *NoSuchElementException*.

```scala
val task: Task[Int] = Task {
  throw new IllegalStateException("illegal state")
}

val failed: Task[Throwable] = task.failed

implicit val scheduler: Scheduler = Scheduler.global

task foreach println
// prints nothing
failed foreach println
// prints: java.lang.IllegalStateException: illegal state
```

6. Task Cancelation

Canceling a *Task*

- *Future* cannot be canceled, but *Task* can.

Canceling a *Task*

- *Future* cannot be canceled, but *Task* can.

```
val task: Task[Int] = Task {
  println("side effect")
  compute
}.delayExecution(1.second)

val callback: Either[Throwable, Int] => Unit = {
  case Right(result) => println(s"result = $result")
  case Left(ex) => println(s"ERROR: ${ex.getMessage}")
}

implicit val scheduler: Scheduler = Scheduler.global

val cancelable: Cancelable = task runAsync callback

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

Cancelable & CancelableFuture

A *Cancelable* is a trait with a *cancel* method.

```
package monix.execution

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

Cancelable & *CancelableFuture*

A *Cancelable* is a trait with a *cancel* method.

```
package monix.execution

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

- *CancelableFuture* is a *(Future with Cancelable)*, hence a *Future* augmented with a *cancel* method.

```
package monix.execution

sealed abstract class CancelableFuture[+A] extends Future[A] with Cancelable {
  // ...
}
```

Canceling a *Future*

- *Task#runToFuture* returns a *CancelableFuture*.

Canceling a *Future*

- *Task#runToFuture* returns a *CancelableFuture*.

```
val task: Task[Int] = Task {
  compute
}.delayExecution(1.second)

implicit val scheduler: Scheduler = Scheduler.global

val future: CancelableFuture[Int] = task.runToFuture

future onComplete {
  case Success(result) => println(s"result = $result")
  case Failure(ex) => println(s"ERROR: ${ex.getMessage}")
}

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

7. Memoization

Memoization - Overview

Memoization - Overview

```scala
// non-strict + memoized evaluation - equivalent to a lazy val
val task = Task.eval { println("side effect"); "some value" } // same as apply
val memoized = task.memoize
```

Memoization - Overview

```
// non-strict + memoized evaluation - equivalent to a lazy val
val task = Task.eval { println("side effect"); "some value" } // same as apply
val memoized = task.memoize
```

```
// non-strict + memoized evaluation - equivalent to a lazy val
Task.evalOnce { println("side effect"); "memoized" }
```

Memoization - Overview

```scala
// non-strict + memoized evaluation - equivalent to a lazy val
val task = Task.eval { println("side effect"); "some value" } // same as apply
val memoized = task.memoize
```

```scala
// non-strict + memoized evaluation - equivalent to a lazy val
Task.evalOnce { println("side effect"); "memoized" }
```

```scala
// memoized only if successful
val task = Task.eval { println("side effect"); "some value" }
val memoizedIfSuccessful = task.memoizeOnSuccess
```

Memoization - Overview

```scala
// non-strict + memoized evaluation - equivalent to a lazy val
val task = Task.eval { println("side effect"); "some value" } // same as apply
val memoized = task.memoize
```

```scala
// non-strict + memoized evaluation - equivalent to a lazy val
Task.evalOnce { println("side effect"); "memoized" }
```

```scala
// memoized only if successful
val task = Task.eval { println("side effect"); "some value" }
val memoizedIfSuccessful = task.memoizeOnSuccess
```

Skip details

Task#memoize

- memoizes (caches) the *Task*'s result on the first run (like a *lazy val*)
- is impure, as memoization is a side effect
- guarantees idempotency

Task#memoize

- memoizes (caches) the *Task*'s result on the first run (like a *lazy val*)
- is impure, as memoization is a side effect
- guarantees idempotency

```
val task = Task {
  println("side effect")
  fibonacci(20)
}

val memoized = task.memoize

memoized runAsync printCallback
//=> side effect
//=> 10946

// Result was memoized on the first run!
memoized runAsync printCallback
//=> 10946
```

Task.evalOnce

- evaluates a thunk lazily on the first run and memoizes/caches the result
- is impure, as memoization is a side effect
- guarantees idempotency
- *Task.evalOnce* { *thunk* } <-> *Task* { *thunk* }.*memoize*

Task.evalOnce

- evaluates a thunk lazily on the first run and memoizes/caches the result
- is impure, as memoization is a side effect
- guarantees idempotency
- *Task.evalOnce* { *thunk* } <-> *Task* { *thunk* }*.memoize*

```
val task = Task.evalOnce {
  println("side effect")
  fibonacci(20)
}

implicit val scheduler: Scheduler = Scheduler.global

task runAsync printCallback
//=> side effect
//=> 10946

// Result was memoized on the first run!
task runAsync printCallback
//=> 10946
```

Task#memoizeOnSuccess

- memoizes (caches) the *Task*'s result on the first run only if it succeeds
- is impure, as memoization is a side effect

Task#memoizeOnSuccess

- memoizes (caches) the *Task*'s result on the first run only if it succeeds
- is impure, as memoization is a side effect

```
var effect = 0

val source = Task.eval {
  println("side effect")
  effect += 1
  if (effect < 3) throw new RuntimeException("dummy") else effect
}

val cached = source.memoizeOnSuccess

cached runAsync printCallback   //=> java.lang.RuntimeException: dummy
cached runAsync printCallback   //=> java.lang.RuntimeException: dummy
cached runAsync printCallback   //=> 3
cached runAsync printCallback   //=> 3
```

8. Task Builders

Task Builders - Overview

Task Builders - Overview

```
// non-strict evaluation - equivalent to a function
Task.eval { println("side effect"); "always" }
```

Task Builders - Overview

```scala
// non-strict evaluation - equivalent to a function
Task.eval { println("side effect"); "always" }
```

```scala
// strict evaluation - lifts a pure value into Task context
Task.now { println("side effect"); "immediate" }
```

Task Builders - Overview

```scala
// non-strict evaluation - equivalent to a function
Task.eval { println("side effect"); "always" }
```

```scala
// strict evaluation - lifts a pure value into Task context
Task.now { println("side effect"); "immediate" }
```

```scala
// strict evaluation - lifts a Throwable into Task context
Task.raiseError { println("side effect"); new Exception }
```

Task Builders - Overview

```
// non-strict evaluation - equivalent to a function
Task.eval { println("side effect"); "always" }
```

```
// strict evaluation - lifts a pure value into Task context
Task.now { println("side effect"); "immediate" }
```

```
// strict evaluation - lifts a Throwable into Task context
Task.raiseError { println("side effect"); new Exception }
```

```
// non-strict + memoized evaluation - equivalent to a lazy val
Task.evalOnce { println("side effect"); "memoized" }
```

Task Builders - Overview

```scala
// non-strict evaluation - equivalent to a function
Task.eval { println("side effect"); "always" }
```

```scala
// strict evaluation - lifts a pure value into Task context
Task.now { println("side effect"); "immediate" }
```

```scala
// strict evaluation - lifts a Throwable into Task context
Task.raiseError { println("side effect"); new Exception }
```

```scala
// non-strict + memoized evaluation - equivalent to a lazy val
Task.evalOnce { println("side effect"); "memoized" }
```

```scala
// non-strict evaluation - turns a (possibly eager) Task into a lazy one
Task.defer { println("side effect"); Task.now("no longer immediate") }
```

Task Builders - Overview

```scala
// non-strict evaluation - equivalent to a function
Task.eval { println("side effect"); "always" }
```

```scala
// strict evaluation - lifts a pure value into Task context
Task.now { println("side effect"); "immediate" }
```

```scala
// strict evaluation - lifts a Throwable into Task context
Task.raiseError { println("side effect"); new Exception }
```

```scala
// non-strict + memoized evaluation - equivalent to a lazy val
Task.evalOnce { println("side effect"); "memoized" }
```

```scala
// non-strict evaluation - turns a (possibly eager) Task into a lazy one
Task.defer { println("side effect"); Task.now("no longer immediate") }
```

```scala
// non-strict evaluation on a different "logical" thread
Task.evalAsync { println("side effect"); "always on different thread" }
```

Task Builders - Overview

```scala
// non-strict evaluation - equivalent to a function
Task.eval { println("side effect"); "always" }
```

```scala
// strict evaluation - lifts a pure value into Task context
Task.now { println("side effect"); "immediate" }
```

```scala
// strict evaluation - lifts a Throwable into Task context
Task.raiseError { println("side effect"); new Exception }
```

```scala
// non-strict + memoized evaluation - equivalent to a lazy val
Task.evalOnce { println("side effect"); "memoized" }
```

```scala
// non-strict evaluation - turns a (possibly eager) Task into a lazy one
Task.defer { println("side effect"); Task.now("no longer immediate") }
```

```scala
// non-strict evaluation on a different "logical" thread
Task.evalAsync { println("side effect"); "always on different thread" }
```

Skip details

Task.eval / Task.apply / Task.delay

- corresponds to *Future.apply*
- evaluates a thunk lazily

Task.eval / Task.apply / Task.delay

- corresponds to *Future.apply*
- evaluates a thunk lazily

```
val task = Task.eval {
  println("side effect")
  fibonacci(20)
}

implicit val scheduler: Scheduler = Scheduler.global

task foreach println
//=> side effect
//=> 10946

// The evaluation (and thus all contained side effects)
// gets triggered on each run:
task runAsync printCallback
//=> side effect
//=> 10946
```

Task.now / Task.pure

- corresponds to *Future.successful*
- lifts a pure value into the *Task* context
- evaluates eagerly / immediately, never spawning a separate thread
- !!! Don't use it for computations and side effects, only for pure values !!!

Task.now / *Task.pure*

- corresponds to *Future.successful*
- lifts a pure value into the *Task* context
- evaluates eagerly / immediately, never spawning a separate thread
- !!! Don't use it for computations and side effects, only for pure values !!!

```
val task = Task.now {
  println("(eagerly produced) side effect; DON'T DO THAT !!")
  42
}
//=> (eagerly produced) side effect; DON'T DO THAT !!

implicit val scheduler: Scheduler = Scheduler.global

task runAsync printCallback
//=> 42
```

Task.unit

- corresponds to *Future.unit*
- lifts a pure Unit value into the *Task* context
- evaluates eagerly / immediately, never spawning a separate thread
- Task.unit <-> Task.now(())

Task.unit

- corresponds to *Future.unit*
- lifts a pure Unit value into the *Task* context
- evaluates eagerly / immediately, never spawning a separate thread
- Task.unit <-> Task.now(())

```
val task: Task[Unit] = Task.unit

implicit val scheduler: Scheduler = Scheduler.global

task runAsync println
//=> Right(())
```

Task.raiseError

- corresponds to *Future.failed*
- lifts a *Throwable* into the *Task* context (analogous to *Task.now*)
- evaluates eagerly / immediately, never spawning a separate thread
- !!! Don't use it for computations and side effects, only for pure errors/exceptions !!!

Task.raiseError

- corresponds to *Future.failed*
- lifts a *Throwable* into the *Task* context (analogous to *Task.now*)
- evaluates eagerly / immediately, never spawning a separate thread
- !!! Don't use it for computations and side effects, only for pure errors/exceptions !!!

```
val task = Task.raiseError {
  println("(eagerly produced) side effect; DON'T DO THAT !!")
  new IllegalStateException("illegal state")
}
//=> (eagerly produced) side effect; DON'T DO THAT !!

implicit val scheduler: Scheduler = Scheduler.global

task runAsync printCallback
//=> java.lang.IllegalStateException: illegal state
```

Task.evalOnce

- corresponds to *Future.apply*
- evaluates a thunk lazily on the first run and memoizes/caches the result
- is impure, as memoization is a side effect
- guarantees idempotency
- Task.evalOnce <--> Task.eval.memoize

Task.evalOnce

- corresponds to *Future.apply*
- evaluates a thunk lazily on the first run and memoizes/caches the result
- is impure, as memoization is a side effect
- guarantees idempotency
- Task.evalOnce <--> Task.eval.memoize

```
val task = Task.evalOnce {
  println("side effect")
  fibonacci(20)
}

implicit val scheduler: Scheduler = Scheduler.global

task foreach println
//=> side effect
//=> 10946

// Result was memoized on the first run!
task runAsync printCallback
//=> 10946
```

Task.never

- corresponds to *Future.never*
- creates a *Task* that never completes
- evaluates lazily

Task.never

- corresponds to *Future.never*
- creates a *Task* that never completes
- evaluates lazily

```scala
val never: Task[Int] = Task.never[Int]

val timedOut: Task[Int] =
      never.timeoutTo(3.seconds, Task.raiseError(new TimeoutException))

implicit val scheduler: Scheduler = Scheduler.global

timedOut runAsync println
// After 3 seconds:
// => Left(java.util.concurrent.TimeoutException)
```

Task.defer / Task.suspend

- takes a (possibly eager) *Task* and turns it into a lazy *Task* of the same type.
- evaluates lazily
- *Task.defer(Task.now(value))* <-> Task.eval(value)

Task.defer / Task.suspend

- takes a (possibly eager) *Task* and turns it into a lazy *Task* of the same type.
- evaluates lazily
- *Task.defer(Task.now(value))* <-> Task.eval(value)

```scala
val task = Task.defer {
  println("side effect")
  Task.now(42)
}

implicit val scheduler: Scheduler = Scheduler.global

task runAsync printCallback
//=> side effect
//=> 10946

// The evaluation (and thus all contained side effects)
// gets triggered on each run:
task runAsync printCallback
//=> side effect
//=> 10946
```

Task.evalAsync

- evaluates a thunk lazily like *Task.eval*
- guaranteed to spawn a separate "logical" thread
- Task.evalAsync(a) <-> Task.eval(a).executeAsync
- See also: Async Boundaries

Task.evalAsync

- evaluates a thunk lazily like *Task.eval*
- guaranteed to spawn a separate "logical" thread
- Task.evalAsync(a) <-> Task.eval(a).executeAsync
- See also: Async Boundaries

```scala
val task = Task.evalAsync {
  println("side effect")
  fibonacci(20)
}

implicit val scheduler: Scheduler = Scheduler.global

// The evaluation (and thus all contained side effects)
// gets triggered on each run.
// But it is run asynchronously on a different "logical" thread.
task runAsync printCallback
//=> side effect
//=> 10946
```

9. Converting Future to Task

Future to *Task* - Overview

Future to *Task* - Overview

```scala
// --- SOME LIBRARY ---
private def compute: Int = ???     // compute Int value

// compute Int value async - library code returns a Future
def computeAsync(implicit ec: ExecutionContext): Future[Int] = Future { compute }
```

Future to *Task* - Overview

```scala
// --- SOME LIBRARY ---
private def compute: Int = ???     // compute Int value

// compute Int value async - library code returns a Future
def computeAsync(implicit ec: ExecutionContext): Future[Int] = Future { compute }
```

```scala
// converts a Future to a effectively memoized Task. Strict evaluation !!!
import monix.execution.Scheduler.Implicits.global
val task = Task.fromFuture(computeAsync) // needs an EC
```

Future to *Task* - Overview

```scala
// --- SOME LIBRARY ---
private def compute: Int = ???     // compute Int value

// compute Int value async - library code returns a Future
def computeAsync(implicit ec: ExecutionContext): Future[Int] = Future { compute }
```

```scala
// converts a Future to a effectively memoized Task. Strict evaluation !!!
import monix.execution.Scheduler.Implicits.global
val task = Task.fromFuture(computeAsync) // needs an EC
```

```scala
// converts a Future to a lazy Task. Non-strict evaluation !!!
import monix.execution.Scheduler.Implicits.global
val task = Task.defer(Task.fromFuture(computeAsync)) // needs an EC
```

Future to *Task* - Overview

```scala
// --- SOME LIBRARY ---
private def compute: Int = ???     // compute Int value

// compute Int value async - library code returns a Future
def computeAsync(implicit ec: ExecutionContext): Future[Int] = Future { compute }
```

```scala
// converts a Future to a effectively memoized Task. Strict evaluation !!!
import monix.execution.Scheduler.Implicits.global
val task = Task.fromFuture(computeAsync) // needs an EC
```

```scala
// converts a Future to a lazy Task. Non-strict evaluation !!!
import monix.execution.Scheduler.Implicits.global
val task = Task.defer(Task.fromFuture(computeAsync)) // needs an EC
```

```scala
// converts a Future to a lazy Task. Non-strict evaluation !!!
import monix.execution.Scheduler.Implicits.global
val task = Task.deferFuture(computeAsync) // needs an EC
```

Future to *Task* - Overview

```scala
// --- SOME LIBRARY ---
private def compute: Int = ???     // compute Int value

// compute Int value async - library code returns a Future
def computeAsync(implicit ec: ExecutionContext): Future[Int] = Future { compute }
```

```scala
// converts a Future to a effectively memoized Task. Strict evaluation !!!
import monix.execution.Scheduler.Implicits.global
val task = Task.fromFuture(computeAsync) // needs an EC
```

```scala
// converts a Future to a lazy Task. Non-strict evaluation !!!
import monix.execution.Scheduler.Implicits.global
val task = Task.defer(Task.fromFuture(computeAsync)) // needs an EC
```

```scala
// converts a Future to a lazy Task. Non-strict evaluation !!!
import monix.execution.Scheduler.Implicits.global
val task = Task.deferFuture(computeAsync) // needs an EC
```

```scala
// converts a Future to a lazy Task. Non-strict evaluation !!!
val task = Task.deferFutureAction(implicit scheduler => computeAsync)
// Scheduler is required when the task is run.
```

Future to *Task* - Overview

```
// --- SOME LIBRARY ---
private def compute: Int = ???     // compute Int value

// compute Int value async - library code returns a Future
def computeAsync(implicit ec: ExecutionContext): Future[Int] = Future { compute }
```

```
// converts a Future to a effectively memoized Task. Strict evaluation !!!
import monix.execution.Scheduler.Implicits.global
val task = Task.fromFuture(computeAsync) // needs an EC
```

```
// converts a Future to a lazy Task. Non-strict evaluation !!!
import monix.execution.Scheduler.Implicits.global
val task = Task.defer(Task.fromFuture(computeAsync)) // needs an EC
```

```
// converts a Future to a lazy Task. Non-strict evaluation !!!
import monix.execution.Scheduler.Implicits.global
val task = Task.deferFuture(computeAsync) // needs an EC
```

```
// converts a Future to a lazy Task. Non-strict evaluation !!!
val task = Task.deferFutureAction(implicit scheduler => computeAsync)
// Scheduler is required when the task is run.
```

Skip details

Task.fromFuture

- converts a *Future* to an effectively memoized *Task*. **Strict** evaluation !!!
- The *Future* starts running eagerly, before *Task.fromFuture* is invoked.
- An implicit *ExecutionContext* or *Scheduler* must be provided.

Task.fromFuture

- converts a *Future* to an effectively memoized *Task*. **Strict** evaluation !!!
- The *Future* starts running eagerly, before *Task.fromFuture* is invoked.
- An implicit *ExecutionContext* or *Scheduler* must be provided.

```
object library {
  def futureFactorial(n: Int)(implicit ec: ExecutionContext): Future[BigInt] =
    Future { println("side effect"); factorial(n) }
}

  import library._
  import Scheduler.Implicits.global

  // Future created before Task.fromFuture is invoked
  // Hence the Future is already running before we convert it to a Task
  val task = Task.fromFuture(futureFactorial(10))
  //=> side effect

  task foreach println
  //=> 3628800
```

Task.defer & *Task.fromFuture*

- converts a *Future* into a lazy *Task*. **Non-strict**/lazy evaluation !!!
- *Task.defer* effectively creates a function which - when invoked - will create the *Future*.
- The *Future* doesn't start running (unless created outside *Task.defer*).
- An implicit *ExecutionContext* or *Scheduler* must be provided.

Task.defer & *Task.fromFuture*

- converts a *Future* into a lazy *Task*. **Non-strict**/lazy evaluation !!!
- *Task.defer* effectively creates a function which - when invoked - will create the *Future*.
- The *Future* doesn't start running (unless created outside *Task.defer*).
- An implicit *ExecutionContext* or *Scheduler* must be provided.

```
object library {
  def futureFactorial(n: Int)(implicit ec: ExecutionContext): Future[BigInt] =
    Future { println("side effect"); factorial(n) }
}

  import library._
  import Scheduler.Implicits.global

  val task = Task.defer {
    Task.fromFuture(futureFactorial(10))
  }

  task foreach println
  //=> side effect
  //=> 3628800
```

Task.deferFuture

- converts a *Future* into a lazy *Task*. **Non-strict**/lazy evaluation !!!
- *Task.deferFuture(f)* <-> *Task.defer(Task.fromFuture(f))*
- The *Future* doesn't start running (unless created outside *Task.deferFuture*)..
- An implicit *ExecutionContext* or *Scheduler* must be provided.

Task.deferFuture

- converts a *Future* into a lazy *Task*. **Non-strict**/lazy evaluation !!!
- *Task.deferFuture(f) <-> Task.defer(Task.fromFuture(f))*
- The *Future* doesn't start running (unless created outside *Task.deferFuture*)..
- An implicit *ExecutionContext* or *Scheduler* must be provided.

```
object library {
  def futureFactorial(n: Int)(implicit ec: ExecutionContext): Future[BigInt] =
    Future { println("side effect"); factorial(n) }
}

  import library._
  import Scheduler.Implicits.global

  val task = Task.deferFuture(futureFactorial(10))

  task foreach println
  //=> side effect
  //=> 3628800
```

Task.deferFutureAction

- converts a *Future* into a lazy *Task*. **Non-strict**/lazy evaluation !!!
- The *Future* doesn't start running (unless created outside *Task.deferFutureAction*)..
- **No** implicit *ExecutionContext* or *Scheduler* must be provided.
- An implicit *Scheduler* must be provided when the task is run.

Task.deferFutureAction

- converts a *Future* into a lazy *Task*. **Non-strict**/lazy evaluation !!!
- The *Future* doesn't start running (unless created outside *Task.deferFutureAction*)..
- **No** implicit *ExecutionContext* or *Scheduler* must be provided.
- An implicit *Scheduler* must be provided when the task is run.

```
object library {
  def futureFactorial(n: Int)(implicit ec: ExecutionContext): Future[BigInt] =
    Future { println("side effect"); factorial(n) }
}

  import library._

  val task = Task.deferFutureAction(implicit scheduler => futureFactorial(10))

  import Scheduler.Implicits.global

  task foreach println
  //=> side effect
  //=> 3628800
```

10. Task as Monad

Task#map

```scala
val task: Task[List[(String, Int)]] =
  Task.eval {     // Attention! This Task doesn't close the resource "README.md"
    Source.fromURL("file:./README.md")
  } map {
    _.getLines
  } map {
    _.toList
  } map {
    wordCount(limit = 2)
  }
```

```scala
import Scheduler.Implicits.global

task runAsync new Callback[Throwable, List[(String, Int)]] {
  def onError(ex: Throwable): Unit =
    println(s"ERROR: ${ex.toString}")
  def onSuccess(wcList: List[(String, Int)]): Unit =
    wcList foreach { case (word, count) => println(s"$word: $count")}
}
```

Task#flatMap

```scala
def sumTask(from: Int, to: Int) = Task { sumOfRange(from, to) }
def fibonacciTask(num: Int) = Task { fibonacci(num) }
def factorialTask(num: Int) = Task { factorial(num) }

def computeTask(from: Int, to: Int): Task[BigInt] =
  sumTask(from, to)
    .flatMap(fibonacciTask)
    .map(_.intValue)
    .flatMap(factorialTask)
```

```scala
val task = computeTask(1, 4)

import Scheduler.Implicits.global

task runAsync printCallback
//=> 6227020800
```

For-comprehension over a *Task* context

```
def sumTask(from: Int, to: Int) = Task { sumOfRange(from, to) }
def fibonacciTask(num: Int) = Task { fibonacci(num) }
def factorialTask(num: Int) = Task { factorial(num) }

def computeTask(from: Int, to: Int): Task[BigInt] =
  for {
    x <- sumTask(from, to)
    y <- fibonacciTask(x)
    z <- factorialTask(y.intValue)
  } yield z
```

```
val task = computeTask(1, 4)

import Scheduler.Implicits.global

task runAsync printCallback
//=> 6227020800
```

Task has a *Monad* instance

```scala
def sumF[F[_]: Monad](from: Int, to: Int): F[Int] =
  Monad[F].pure { sumOfRange(from, to) }

def fibonacciF[F[_]: Monad](num: Int): F[BigInt] =
  Monad[F].pure { fibonacci(num) }

def factorialF[F[_]: Monad](num: Int): F[BigInt] =
  Monad[F].pure { factorial(num) }

def computeF[F[_]: Monad](from: Int, to: Int): F[BigInt] =
  for {
    x <- sumF(from, to)
    y <- fibonacciF(x)
    z <- factorialF(y.intValue)
  } yield z
```

```scala
// reify F[] with Task
val task: Task[BigInt] = computeF[Task](1, 4)

import Scheduler.Implicits.global

task runAsync printCallback
//=> 6227020800
```

computeF can be used with any *Monad* instance

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import cats.instances.future._

// reify F[] with Future
val task: Future[BigInt] = computeF[Future](1, 4)

task foreach { result => println(s"result = $result")}
//=> 6227020800
```

```
import cats.Id

// reify F[] with Id
val result: Id[BigInt] = computeF[Id](1, 4)     // Id[BigInt] ^= BigInt

println(s"result = $result")
//=> 6227020800
```

```
import cats.instances.option._

// reify F[] with Option
val maybeResult: Option[BigInt] = computeF[Option](1, 4)

maybeResult foreach { result => println(s"result = ${result}") }
//=> 6227020800
```

11. Tail Recursive Loops

Tail recursion with annotation *@tailrec*

- With the *@tailrec* annaotation the compiler guarantees us stack-safety.

```
@scala.annotation.tailrec
def fibonacci(cycles: Int, x: BigInt = 0, y: BigInt = 1): BigInt =
  if (cycles > 0)
    fibonacci(cycles-1, y, x + y)
  else
    y
```

```
val result: BigInt = fibonacci(6)

println(result)
//=> 13
```

Tail recursion with *Task*

- With *Task* the fibonacci impl is lazy (due to *Task.defer*).
- The annotation *@tailrec* is no longer required.
- The impl is nevertheless stack-safe.

```
def fibonacciTask(cycles: Int, x: BigInt = 0, y: BigInt = 1): Task[BigInt] =
  if (cycles > 0)
    Task.defer(fibonacciTask(cycles-1, y, x+y))
  else
    Task.now(y)
```

```
val task: Task[BigInt] = fibonacciTask(6)

implicit val scheduler: Scheduler = Scheduler.global

task foreach println    // fibonacci computation starts here
//=> 13
```

Tail recursion with *Task#flatMap*

- With *Task* the fibonacci impl is lazy (due to *Task#flatMap*).
- The annotation *@tailrec* is no longer required.
- The impl is nevertheless stack-safe.

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

```scala
val task: Task[BigInt] = fibonacciTask(6)

implicit val scheduler: Scheduler = Scheduler.global

task foreach println    // fibonacci computation starts here
//=> 13
```

Mutual tail recursion

- With *Task* even mutual tail recursive calls are possible.

```scala
def odd(n: Int): Task[Boolean] =
  Task.eval(n == 0).flatMap {
    case true => Task.now(false)
    case false => if (n > 0) even(n - 1) else even(n + 1)
  }

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

```scala
val task: Task[Boolean] = even(-1000000)

implicit val scheduler: Scheduler = Scheduler.global

task foreach println
//=> true
```

12. Async Boundaries

Green Threads and JVM Threads

1. JVM threads map 1:1 to OS threads. They have their own stack and register set each. They are managed by the OS using preemptive multitasking in kernel space. JVM threads are a scarce resource. Switching context between them is expensive.

2. Green threads (aka *Fibers*) are managed by the runtime (Haskell) or a library (Monix, Cats Effect, Akka). This allows for cooperative multitasking in user space. Green threads can yield execution back to the thread pool which then might choose another green thread to run on the same JVM thread. A context switch between green threads is cheap.

Your application may use relatively few JVM threads but many green threads.

Monix *Task* provides asynchrony with JVM threads and green threads.

- Avoid blocking a JVM thread!

- You may block a green thread without blocking the JVM thread. ("asynchronous blocking" or "semantic blocking").

Asynchronous Boundaries - Overview

- An asynchronous boundary is a context switch between two logical threads.
- A logical thread may be a green thread or a JVM thread.

Asynchronous Boundaries - Overview

- An asynchronous boundary is a context switch between two logical threads.
- A logical thread may be a green thread or a JVM thread.

```
// Task#executeAsync ensures an async boundary, forks into another logical thread.
val source = Task.eval { computation }
val forked = source.executeAsync
```

Asynchronous Boundaries - Overview

- An asynchronous boundary is a context switch between two logical threads.
- A logical thread may be a green thread or a JVM thread.

```scala
// Task#executeAsync ensures an async boundary, forks into another logical thread.
val source = Task.eval { computation }
val forked = source.executeAsync
```

```scala
// Task#executeOn executes the current Task on another Scheduler.
// This implies a context switch between JVM threads.
val source = Task.eval { computation }
val forked = source.executeOn(someScheduler)
```

Asynchronous Boundaries - Overview

- An asynchronous boundary is a context switch between two logical threads.
- A logical thread may be a green thread or a JVM thread.

```scala
// Task#executeAsync ensures an async boundary, forks into another logical thread.
val source = Task.eval { computation }
val forked = source.executeAsync
```

```scala
// Task#executeOn executes the current Task on another Scheduler.
// This implies a context switch between JVM threads.
val source = Task.eval { computation }
val forked = source.executeOn(someScheduler)
```

```scala
// Task#asyncBoundary introduces an async Boundary
// (possibly switching back to the default Scheduler).
val source = Task.eval { computation }
val forked = source.executeOn(someScheduler)
val back = forked.asyncBoundary
```

Asynchronous Boundaries - Overview

- An asynchronous boundary is a context switch between two logical threads.
- A logical thread may be a green thread or a JVM thread.

```
// Task#executeAsync ensures an async boundary, forks into another logical thread.
val source = Task.eval { computation }
val forked = source.executeAsync
```

```
// Task#executeOn executes the current Task on another Scheduler.
// This implies a context switch between JVM threads.
val source = Task.eval { computation }
val forked = source.executeOn(someScheduler)
```

```
// Task#asyncBoundary introduces an async Boundary
// (possibly switching back to the default Scheduler).
val source = Task.eval { computation }
val forked = source.executeOn(someScheduler)
val back = forked.asyncBoundary
```

Skip details

Task#executeAsync

- ensures an asynchronous boundary
- forces the fork into another thread on execution

Task#executeAsync

- ensures an asynchronous boundary
- forces the fork into another thread on execution

```scala
val task = Task.eval {
  println(s"Running on $currentThread")
  factorial(10)
}.executeAsync

import Scheduler.Implicits.global

task.runToFuture foreach println
//=> Running on Thread: scala-execution-context-global-241
//=> 3628800
```

Task#executeOn (1/2)

- executes the current Task on another Scheduler
- (possibly) introduces an asynchronous boundary
- (possibly) implies a context switch between JVM threads

Task#executeOn (1/2)

- executes the current Task on another Scheduler
- (possibly) introduces an asynchronous boundary
- (possibly) implies a context switch between JVM threads

```scala
lazy val io = Scheduler.io(name="my-io")  // I/O scheduler

val source = Task { println(s"Running on $currentThread") }

val forked = source.executeOn(io)         // override default Scheduler by fork

import Scheduler.Implicits.global

source.runToFuture
//=> Running on Thread: run-main-9
forked.runToFuture
//=> Running on Thread: my-io-265
```

Task#executeOn (2/2)

```
lazy val io = Scheduler.io(name="my-io")  // I/O scheduler

val source = Task { println(s"Running on $currentThread") }

val forked = source.executeOn(io)         // override default Scheduler by fork

val onFinish = Task { println(s"Ending on $currentThread") }

val task =
  source
    .flatMap(_ => forked)
    .doOnFinish(_ => onFinish)


import Scheduler.Implicits.global

task.runToFuture
//=> Running on thread: run-main-2a
//=> Running on thread: my-io-803
//=> Ending on thread: scala-execution-context-global-800
```

Task#asyncBoundary

- switches execution back to the default scheduler
- introduces an asynchronous boundary (if task is already running on the default scheduler)

Task#asyncBoundary

- switches execution back to the default scheduler
- introduces an asynchronous boundary (if task is already running on the default scheduler)

```
lazy val io = Scheduler.io(name="my-io")  // I/O scheduler

val source = Task { println(s"Running on $currentThread") }

val forked = source.executeOn(io)         // override default Scheduler by fork

val onFinish = Task { println(s"Ending on $currentThread") }

val task =
  source // executes on global
    .flatMap(_ => forked) // executes on io
    .asyncBoundary // switch back to global
    .doOnFinish(_ => onFinish) // executes on global

import Scheduler.Implicits.global

task.runToFuture
//=> Running on Thread: run-main-e
//=> Running on Thread: my-io-358
//=> Ending on Thread: scala-execution-context-global-343
```

Asynchrony on a single JVM Thread

- Green threads allow for asynchrony on a single JVM thread.
- The example *Task* introduces several async boundaries which allow to shift the context of logical threads.
- The Task is run twice on a thread pool with a single thread.

Asynchrony on a single JVM Thread

- Green threads allow for asynchrony on a single JVM thread.
- The example *Task* introduces several async boundaries which allow to shift the context of logical threads.
- The Task is run twice on a thread pool with a single thread.

```
def createTask(id: Int): Task[Int] =
  Task { println(s"Task no $id starting on $currentThread"); id }
    .asyncBoundary
    .map { someComputation }
    .asyncBoundary
    .map { someComputation }
    .asyncBoundary
    .map { someComputation }
    .asyncBoundary
    .map { someComputation }

implicit val scheduler: Scheduler = Scheduler.singleThread("SingleThreadScheduler"

createTask(1) runAsync (_ => println(s"Task no 1 finished"))
createTask(2) runAsync (_ => println(s"Task no 2 finished"))
```

Output

```
Task no 1 starting on Thread: run-main-1b
Computation no 1 running on Thread: SingleThreadScheduler-587
Task no 2 starting on Thread: run-main-1b
Computation no 2 running on Thread: SingleThreadScheduler-587
Computation no 1 running on Thread: SingleThreadScheduler-587
Computation no 2 running on Thread: SingleThreadScheduler-587
Computation no 1 running on Thread: SingleThreadScheduler-587
Computation no 2 running on Thread: SingleThreadScheduler-587
Computation no 1 running on Thread: SingleThreadScheduler-587
Task no 1 finished on Thread: SingleThreadScheduler-587
Computation no 2 running on Thread: SingleThreadScheduler-587
Task no 2 finished on Thread: SingleThreadScheduler-587
```

13. Schedulers

ExecutionContext - Downsides

ExecutionContext is limited because …

- It cannot execute things with a given delay.
- It cannot execute units of work periodically (e.g. once every second).
- The execute() method doesn't return a token you could use to cancel the pending execution of a task.

ExecutionContext is **shared mutable state** … and *Scheduler* is too.

But there is a big difference:

- A *Scheduler* is only needed when running a *Task* (at the end of the world). Whereas an *ExecutionContext* is already required when you create a *Future* and again for nearly every method of *Future* like *map*, *flatMap* and *onComplete*.

Scheduler

- *Scheduler* is a subclass of *ExecutionContext*, hence it also serves as enhanced *ExecutionContext* for *Future*.
- It is also a replacement for Java’s *ScheduledExecutorService* (due to it's scheduling and cancelation capabilities).
- *Future* requires an *ExecutionContext* for almost every method (*apply*, *map*, *flatMap*, *onComplete*, etc.) whereas …
- *Task* (due to it's laziness) requires a *Scheduler* only when run.

Run a *Runnable* on a *Scheduler*

- A *Scheduler* can execute *Runnables* directly.

```
Scheduler.global.execute(new Runnable {
  def run(): Unit = println("Hello, world!")
})
```

- We normally do not work with *Runnables*, we use other abstractions like *Future* or *Task*.
- Running a *Task* requires an implicit *Scheduler* in scope.

Compared to *ExecutionContext* a *Scheduler* …

- can execute a *Runnable* with a delay (EC cannot).
- can execute a *Runnable* periodically (EC cannot).
- can provide a cancel token to cancel a running execution (EC cannot).

Execute *Runnables* - Overview

- *Scheduler* provides different ways to run *Runnables*.

Execute *Runnable*s - Overview

- *Scheduler* provides different ways to run *Runnable*s.

```
// executes a Runnable
scheduler.execute(runnable)
```

Execute *Runnables* - Overview

- *Scheduler* provides different ways to run *Runnables*.

```
// executes a Runnable
scheduler.execute(runnable)
```

```
// schedules a thunk for execution with a delay
val cancelable: Cancelable = scheduler.scheduleOnce(initialDelay) { thunk }
```

Execute *Runnables* - Overview

- *Scheduler* provides different ways to run *Runnables*.

```
// executes a Runnable
scheduler.execute(runnable)
```

```
// schedules a thunk for execution with a delay
val cancelable: Cancelable = scheduler.scheduleOnce(initialDelay) { thunk }
```

```
// schedules a thunk with an initial delay and a periodic delay
// between the end of a task and the start of the next task.
// The effective delay is   periodic delay + duration of the task
val cancelable: Cancelable = scheduler.scheduleWithFixedDelay(
                                      initialDelay, periodicDelay) { thunk }
```

Execute *Runnables* - Overview

- *Scheduler* provides different ways to run *Runnables*.

```
// executes a Runnable
scheduler.execute(runnable)
```

```
// schedules a thunk for execution with a delay
val cancelable: Cancelable = scheduler.scheduleOnce(initialDelay) { thunk }
```

```
// schedules a thunk with an initial delay and a periodic delay
// between the end of a task and the start of the next task.
// The effective delay is   periodic delay + duration of the task
val cancelable: Cancelable = scheduler.scheduleWithFixedDelay(
                                      initialDelay, periodicDelay) { thunk }
```

```
// schedules a thunk with an initial delay and a delay between periodic restarts
// The effective delay is only the periodic delay.
// (The duration of the task is not taken into account.)
val cancelable: Cancelable = scheduler.scheduleAtFixedRate(
                                      initialDelay, periodicDelay) { thunk }
```

Execution Models

- A Scheduler has an *ExecutionModel*.
- The *ExecutionModel* affects **how** a *Task* is evaluated.
- Monix 3.x provides three different *ExecutionModel*s:
    - SynchronousExecution: preferres sync execution as long as possible
    - AlwaysAsyncExecution: ensures async execution for each step of a task
    - BatchedExecution (default): specifies a mixed execution mode under which tasks are executed synchronously in batches up to a maximum size, after which an asynchronous boundary is forced.
      (The batch size is configurable. Default: 1024)

UncaughtExceptionReporter

- A *Scheduler* has an *UncaughtExceptionReporter*.
- *UncaughtExceptionReporter* is a trait with a method *reportFailure*:

```
package monix.execution

trait UncaughtExceptionReporter extends Serializable {
  def reportFailure(ex: Throwable): Unit
}
```

- The default reporter logs exceptions to STDERR (on the JVM, not on ScalaJS).
- Create your own customized *UncaughtExceptionReporter* if needed:

```
val myReporter = UncaughtExceptionReporter(ex => log.error(ex))
implicit lazy val myScheduler = Scheduler(ExecutionContext.global, myReporter)
```

Build your own *Scheduler*

- The global *Scheduler* is backed by the global *ExecutionContext* which is backed by Java's *ForkJoinPool*.
- The global *Scheduler* is a good default suitable for many scenarios. But ...
- You can create your own *Scheduler* for the needs of your application.
- Monix provides many builders to build your own.

The most generic *Scheduler* Builder

```scala
lazy val scheduledExecutor: ScheduledExecutorService = ???
lazy val executionContext: ExecutionContext = ???
lazy val exceptionReporter: UncaughtExceptionReporter = ???
lazy val executionModel: ExecutionModel = ???

lazy val scheduler = Scheduler(
  scheduledExecutor,          // schedules delays
  executionContext,           // executes Runnables or Tasks
  exceptionReporter,          // logs uncaught exceptions to some log destination
  executionModel              // specifies the ExecutionModel to use
)
```

The most generic *Scheduler* Builder

```
lazy val scheduledExecutor: ScheduledExecutorService = ???
lazy val executionContext: ExecutionContext = ???
lazy val exceptionReporter: UncaughtExceptionReporter = ???
lazy val executionModel: ExecutionModel = ???

lazy val scheduler = Scheduler(
  scheduledExecutor,          // schedules delays
  executionContext,           // executes Runnables or Tasks
  exceptionReporter,          // logs uncaught exceptions to some log destination
  executionModel              // specifies the ExecutionModel to use
)
```

- There are many overloads of this builder with good defaults for one, two or three parameters, e.g. ...

```
lazy val scheduler2 = Scheduler(scheduledExecutor, executionContext)

lazy val scheduler3 = Scheduler(executionContext)

lazy val scheduler4 = Scheduler(scheduledExecutor)

lazy val scheduler5 = Scheduler(executionContext, executionModel)

lazy val scheduler6 = Scheduler(executionModel)
```

Spezialized *Schedulers*

Spezialized *Schedulers*

- for computation (for CPU bound tasks, backed by a *ForkJoinPool*):

```
lazy val scheduler7 = Scheduler.computation(parallelism = 10)
lazy val scheduler8 =
    Scheduler.computation(parallelism = 10, executionModel = AlwaysAsyncExecution)
```

Spezialized *Schedulers*

- for computation (for CPU bound tasks, backed by a *ForkJoinPool*):

```
lazy val scheduler7 = Scheduler.computation(parallelism = 10)
lazy val scheduler8 =
    Scheduler.computation(parallelism = 10, executionModel = AlwaysAsyncExecution)
```

- for I/O (unbounded thread-pool, backed by a *CachedThreadPool*):

```
lazy val scheduler9 = Scheduler.io()
lazy val scheduler10 = Scheduler.io(name="my-io")
lazy val scheduler11 =
    Scheduler.io(name="my-io", executionModel = AlwaysAsyncExecution)
```

Spezialized *Schedulers*

- for computation (for CPU bound tasks, backed by a *ForkJoinPool*):

```
lazy val scheduler7 = Scheduler.computation(parallelism = 10)
lazy val scheduler8 =
    Scheduler.computation(parallelism = 10, executionModel = AlwaysAsyncExecution)
```

- for I/O (unbounded thread-pool, backed by a *CachedThreadPool*):

```
lazy val scheduler9 = Scheduler.io()
lazy val scheduler10 = Scheduler.io(name="my-io")
lazy val scheduler11 =
    Scheduler.io(name="my-io", executionModel = AlwaysAsyncExecution)
```

- for single thread pool (backed by a *SingleThreadScheduledExecutor*):

```
lazy val scheduler12 = Scheduler.singleThread(name = "my-single-thread-pool")
```

Spezialized *Schedulers*

- for computation (for CPU bound tasks, backed by a *ForkJoinPool*):

```
lazy val scheduler7 = Scheduler.computation(parallelism = 10)
lazy val scheduler8 =
    Scheduler.computation(parallelism = 10, executionModel = AlwaysAsyncExecution)
```

- for I/O (unbounded thread-pool, backed by a *CachedThreadPool*):

```
lazy val scheduler9 = Scheduler.io()
lazy val scheduler10 = Scheduler.io(name="my-io")
lazy val scheduler11 =
    Scheduler.io(name="my-io", executionModel = AlwaysAsyncExecution)
```

- for single thread pool (backed by a *SingleThreadScheduledExecutor*):

```
lazy val scheduler12 = Scheduler.singleThread(name = "my-single-thread-pool")
```

- for fixed size thread pool (backed by a *ScheduledThreadPool*):

```
lazy val scheduler13 =
    Scheduler.fixedPool(name = "my-fixed-size-pool", poolSize = 10)
```

Shutdown with SchedulerService

- Most *Scheduler* builders return a *SchedulerService*.
- *SchedulerService* can be used to initiate and control the shutdown process.
    - shutdown: Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
    - awaitTermination: Returns a Future that will be complete when all tasks have completed execution after a shutdown request, or the timeout occurs.
    - isShutdown: Returns true if this scheduler has been shut down.
    - isTerminated: Returns true if all tasks have completed after shut down.

Shutdown with SchedulerService - Code

```
val io: SchedulerService = Scheduler.io("my-io")

io.execute(() => {
  Thread sleep 5000L
  println(s"Running on $currentThread")
  println("Hello, world!")
})

io.shutdown()

println(s"isShutdown = ${io.isShutdown}")

val termination: Future[Boolean] = io.awaitTermination(10.seconds, global)
Await.ready(termination, Duration.Inf)

println(s"isTerminated = ${io.isTerminated}")
```

Output

```
isShutdown = true
Running on Thread: my-io-114
Hello, world!
isTerminated = true
```

14. Error Handling

Signaling of Errors - Overview

- Errors in *Task*s are signaled in the result of a *Task*. You can react on errors in the callback (in *case Left(ex)* or in *case Failure(ex)*)
- Errors in callbacks are logged to the *UncaughtExceptionReporter* of the *Scheduler* (which prints to stderr by default).
- The *UncaughtExceptionReporter* of the *Scheduler* can be overriden in order to log to a different destination.

Signaling of *Task* Errors

- Errors in the *Task* impl are signaled to the run *methods, e.g. in* runAsync*

```
val task: Task[Int] = Task(Random.nextInt).flatMap {
  case even if even % 2 == 0 =>
    Task.now(even)
  case odd =>
    throw new IllegalStateException(odd.toString)   // Error in task definition
}

implicit val scheduler: Scheduler = Scheduler.global

task runAsync println
//=> Right(-924040280)

task runAsync println
//=> Left(java.lang.IllegalStateException: 834919637)
```

Signaling of Callback Errors

- Errors in the callback are logged to the *UncaughtExceptionReporter* (stderr by default).

```scala
// Ensures asynchronous execution, just to show that
// the action doesn't happen on the current thread.
val source =
  Task(sumOfRange(1, 100))
    .delayExecution(10.seconds)

implicit val scheduler: Scheduler = Scheduler.global

task runAsync { r => throw new IllegalStateException(r.toString) }

// After 1 second, this will log the whole stack trace:
java.lang.IllegalStateException: Right(2)
      at chap14errorhandling.App02SignalingErrorInCallback$.$anonfun$new$1(App02Si
      at monix.execution.Callback$$anon$3.onSuccess(Callback.scala:145)
      // stack trace continued ...
```

Overriding the Error Logging

- Override *UncaughtExceptionReporter* to log to a different log destination.

```
import monix.execution.UncaughtExceptionReporter
import org.slf4j.LoggerFactory

val reporter = UncaughtExceptionReporter { ex =>
  val logger = LoggerFactory.getLogger(this.getClass) // log with SLF4J
  logger.error("Uncaught exception", ex)
}

implicit val global: Scheduler = Scheduler(Scheduler.global, reporter)

val source =
  Task(sumOfRange(1, 100))
    .delayExecution(10.seconds)

task.runAsync { r => throw new IllegalStateException(r.toString) }
// logs Exception with SLF4J
```

Timouts - Overview

```scala
// Triggers TimeoutException if the source does not complete in 3 seconds
val timedOut: Task[Int] = source.timeout(3.seconds)
```

```scala
// Triggers a Fallback Task if the source does not complete in 3 seconds
source.timeoutTo(
  3.seconds,
  Task.raiseError(new TimeoutException("That took too long!")) // Fallback Task
)
```

Task#timout

- triggers a *TimoutException* after the specified time.

```
val source: Task[Int] =
  Task(sumOfRange(1, 100))
    .delayExecution(10.seconds)

// Triggers TimeoutException if the source does not
// complete in 3 seconds after runAsync
val timedOut: Task[Int] = source.timeout(3.seconds)

implicit val scheduler: Scheduler = Scheduler.global

timedOut runAsync println
//=> Left(java.util.concurrent.TimeoutException: Task timed-out after 3 seconds of
```

Task#timoutTo

- falls back to another *Task* after the specified time.

```
val source: Task[Int] =
  Task(sumOfRange(1, 100))
    .delayExecution(10.seconds)

// Triggers TimeoutException if the source does not
// complete in 3 seconds after runAsync
val timedOut: Task[Int] = source.timeoutTo(
  3.seconds,
  Task.raiseError(new TimeoutException("That took too long!")) // Fallback Task
)

implicit val scheduler: Scheduler = Scheduler.global

timedOut runAsync println
//=> Left(java.util.concurrent.TimeoutException: That took too long!)
```

Error Handlng - Overview (1/2)

```scala
// handles failure with the specified (lazy) handler Function1
val f: Function1[Throwable, Task[B]] = ??? // error handler
task.onErrorHandleWith(f)
```

```scala
// handles failure  with the specified (lazy) handler PartialFunction
val pf: PartialFunction[Throwable, Task[B]] = ??? // error handler
task.onErrorRecoverWith(pf)
```

```scala
// handles failure with the specified (eager) handler Function1
val f: Function1[Throwable, B] = ??? // error handler
task.onErrorHandle(f)
```

```scala
// handles failure  with the specified (eager) handler PartialFunction
val pf: PartialFunction[Throwable, B] = ??? // error handler
task.onErrorRecoverWith(pf)
```

Error Handlng - Overview (2/2)

```
// in case of failure keeps retrying until maxRetries is reached
val f: Throwable => Task[B] = ??? // error handler
task.onErrorRestart(maxRetries = 3)
```

```
// in case of failure keeps retrying until predicate p becomes true
val p: Throwable => Boolean = ???
task.onErrorRestart(f)
```

Task#onErrorHandleWith

- provides a function *Throwable* => *Task[B]* which returns a (lazy) fallback *Task* in case an exception is thrown.

```
val source =
  Task(sumOfRange(1, 100))
    .delayExecution(10.seconds)
    .timeout(3.seconds)

val recovered = source.onErrorHandleWith {
  case _: TimeoutException =>
    // Oh, we know about timeouts, recover it
    Task.now("Recovered!")
  case other =>
    // We have no idea what happened, raise error!
    Task.raiseError(other)
}

implicit val scheduler: Scheduler = Scheduler.global

recovered.runToFuture foreach println
//=> Recovered!
```

Task#onErrorRecoverWith

- provides a *PartialFunction[Throwable, Task[B]]* which returns a (lazy) fallback *Task* in case an exception is thrown.
- avoids the 'other' case in the previous example.

```scala
val source =
  Task(sumOfRange(1, 100))
    .delayExecution(10.seconds)
    .timeout(3.seconds)

val recovered = source.onErrorRecoverWith {
  case _: TimeoutException =>
    // Oh, we know about timeouts, recover it
    Task.now("Recovered!")
}

implicit val scheduler: Scheduler = Scheduler.global

recovered.runToFuture foreach println
//=> Recovered!
```

Task#onErrorHandle

- provides a function *Throwable* => *B* which returns a (eager) fallback value in case an exception is thrown.

```
val source =
  Task(sumOfRange(1, 100))
    .delayExecution(10.seconds)
    .timeout(3.seconds)

val recovered = source.onErrorHandle {
  case _: TimeoutException =>
    // Oh, we know about timeouts, recover it
    "Recovered!"
  case other =>
    throw other // Rethrowing
}

implicit val scheduler: Scheduler = Scheduler.global

recovered.runToFuture foreach println
//=> Recovered!
```

Task#onErrorRecover

- provides a *PartialFunction[Throwable, B]* which returns a (eager) fallback value in case an exception is thrown.
- avoids the 'other' case in the previous example.

```
val source =
  Task(sumOfRange(1, 100))
    .delayExecution(10.seconds)
    .timeout(3.seconds)

val recovered = source.onErrorRecover {
  case _: TimeoutException =>
    // Oh, we know about timeouts, recover it
    "Recovered!"
}

implicit val scheduler: Scheduler = Scheduler.global

recovered.runToFuture foreach println
//=> Recovered!
```

Task#onErrorRestart

- specifies max. retries in case *Task* execution throws an exception.

```
val source = Task(Random.nextInt).flatMap {
  case even if even % 2 == 0 =>
    Task.now(even)
  case other =>
    Task.raiseError(new IllegalStateException(other.toString))
}

// Will retry 2 times for a random even number, or fail if the maxRetries is reach
val randomEven: Task[Int] = source.onErrorRestart(maxRetries = 3)

implicit val scheduler: Scheduler = Scheduler.global

randomEven runAsync println
```

Task#onErrorRestartIf

- keeps retrying in case of an exception as long as the specified predicate *Throwable => Boolean* is true.

```
val source = Task(Random.nextInt).flatMap {
  case even if even % 2 == 0 =>
    Task.now(even)
  case other =>
    Task.raiseError(new IllegalStateException(other.toString))
}

// Will keep retrying for as long as the source fails with an IllegalStateExceptio
val randomEven: Task[Int] = source.onErrorRestartIf {
  case _: IllegalStateException => true
  case _ => false
}

implicit val scheduler: Scheduler = Scheduler.global

randomEven runAsync println
```

Retry with Backoff

- *onErrorHandleWith* allows us to implement a retry with backoff.

```
val source = Task(Random.nextInt).flatMap {
  case even if even % 2 == 0 =>
    Task.now(even)
  case other =>
    Task.raiseError(new IllegalStateException(other.toString))
}

def retryBackoff[A](source: Task[A], maxRetries: Int, firstDelay: FiniteDuration):
  source.onErrorHandleWith {
    case ex: Exception =>
      if (maxRetries > 0) {
        println(s"Retrying ... maxRetries = $maxRetries, nextDelay = ${firstDelay
        // Recursive call, it's OK as Monix is stack-safe
        retryBackoff(source, maxRetries - 1, firstDelay * 2).delayExecution(firstD
      } else
        Task.raiseError(ex)
  }

val randomEven: Task[Int] = retryBackoff(source, 3, 1.second)

implicit val scheduler: Scheduler = Scheduler.global

println(Await.result(randomEven.runToFuture, 10.seconds))
```

Task#restartUntil

- restarts a *Task* until the specified predicate is true.

```
val random = Task.eval(Random.nextInt())

val predicate: Int => Boolean = _ % 2 == 0
val randomEven = random.restartUntil(predicate)

implicit val scheduler: Scheduler = Scheduler.global

randomEven.runToFuture foreach println
// prints an even Int number
```

Task#doOnFinish

- specifies an extra finish callback *Option[Throwable]* => *Task[Unit]* to be invoked when the *Task* is finished.

```scala
val task: Task[Int] = Task(1)

val finishCallback: Option[Throwable] => Task[Unit] = {
  case None =>
    println("Was success!")
    Task.unit
  case Some(ex) =>
    println(s"Had failure: $ex")
    Task.unit
}

val withFinishCallback: Task[Int] = task doOnFinish finishCallback

implicit val scheduler: Scheduler = Scheduler.global

withFinishCallback.runToFuture foreach println
//=> Was success!
//=> 1
```

15. Races

Races - Overview

```scala
// runs 2 Tasks in parallel.
// the resulting Either allows
//          to cancel the other Task or to await its completion.
val raceTask: Task[Either[(Int, Fiber[Int]), (Fiber[Int], Int)]] =
      Task.racePair(task1, task2)
```

```scala
// runs 2 Tasks in parallel and automatically cancels the losere Task.
// the resulting Either gives access to the winner's result.
val winnerTask: Task[Either[Int, Int]] = Task.race(task1, task2)
```

```scala
// runs a Seq[Task[A]] in parallel and automatically cancels the losers.
// Running it returns the winners result.
val winnerTask: Task[Int] = Task.raceMany(Seq(task1, task2))
```

Fiber

- With a *Fiber* you can cancel the other (not yet completed) *Task* or join to it (= wait for it).

```
trait Fiber[A] extends cats.effect.Fiber[Task, A] {

  // triggers the cancelation of the fiber.
  def cancel: CancelToken[Task]

  // returns a new task that will await the completion of the underlying fiber.
  def join: Task[A]
}
```

Task.racePair

- runs two *Tasks* in parallel and returns when the first is complete.
- returns an *Either* containing two tuples each containing a result value and a *Fiber* of the other *Task*.
- If the 1st *Task* completes first the *Either* is a *Left* with the result of the first *Task* and a *Fiber* for the second *Task* (analogous if the *Either* is a *Right*).

```
val random = scala.util.Random
val task1 = Task(10 + 1).delayExecution(random.nextInt(3) seconds)
val task2 = Task(20 + 1).delayExecution(random.nextInt(3) seconds)

val raceTask: Task[Either[(Int, Fiber[Int]), (Fiber[Int], Int)]] =
      Task.racePair(task1, task2)

implicit val scheduler: Scheduler = Scheduler.global
val raceFuture: CancelableFuture[Either[(Int, Fiber[Int]), (Fiber[Int], Int)]] = r

raceFuture.foreach {
  case Left((result1, fiber)) =>
    fiber.cancel
    println(s"Task 1 succeeded with result: $result1")
  case Right((fiber, result2)) =>
    fiber.cancel
    println(s"Task 2 succeeded with result: $result2")
}
```

Task.race

- runs two *Tasks* in parallel and returns after the first one completes.
- The loser is cancelled automatically.
- The resulting *Either* is a *Left* if the first one completes first, otherwise a *Right*.

```
val random = scala.util.Random
val task1 = Task(10 + 1).delayExecution(random.nextInt(3) seconds)
val task2 = Task(20 + 1).delayExecution(random.nextInt(3) seconds)

val winnerTask: Task[Either[Int, Int]] = Task.race(task1, task2)

implicit val scheduler: Scheduler = Scheduler.global

winnerTask
  .runToFuture
  .foreach(result => println(s"Winner's result: $result"))
```

Task.raceMany

- runs a *Seq[Task[A]]* in parallel and returns the first that completes.
- All loosers are cancelled automatically.

```scala
val random = scala.util.Random
val task1 = Task(10 + 1).delayExecution(random.nextInt(3) seconds)
val task2 = Task(20 + 1).delayExecution(random.nextInt(3) seconds)

val winnerTask: Task[Int] = Task.raceMany(Seq(task1, task2))

implicit val scheduler: Scheduler = Scheduler.global

winnerTask
  .runToFuture
  .foreach(result => println(s"Winner's result: $result"))
```

16. Delaying a *Task*

Delaying a *Task* - Overview

- For a *Task* you can delay the execution or the signaling of the result - either for a certain time duration or until another *Task* is complete.

```
// delays the execution of a Task
//              for a certain time duration
val delayed: Task[Int] = task.delayExecution(3.seconds)
```

```
// delays the execution of a Task
//              until another Task (the trigger) is complete
val trigger: Task[Unit] = ???
val delayed: Task[Int] = trigger.flatMap(_ => source)
```

```
// delays signaling of a Task's result
//              for a certain time duration
val delayed: Task[Int] = task.delayResult(3.seconds)
```

```
// delays signaling of a Task's result
//               until another Task (the selector) is complete
def selector(x: Int): Task[Int] = ???
val delayed: Task[Int] = task.flatMap(x => selector(x))
```

Task#delayExecution

- delays the execution of a *Task* for a certain time duration.

```
val source = Task {
  println(s"side effect on $currentThread")
  "Hello"
}

val delayed: Task[String] =
  source
    .delayExecution(2.seconds)

implicit val scheduler: Scheduler = Scheduler.global

delayed.runToFuture foreach println
```

Delay a *Task* until another *Task* is complete

- delays the execution of a *Task* until another *Task* (the trigger) is complete.

```
val source = Task {
  println(s"side effect on $currentThread")
  "Hello"
}

val delayed: Task[String] =
  trigger                 // trigger delays execution of source.
    .flatMap(_ => source)

implicit val scheduler: Scheduler = Scheduler.global

delayed.runToFuture foreach println
```

Delay a *Task* until another *Task* is complete

- delays the execution of a *Task* until another *Task* (the trigger) is complete.

```
val source = Task {
  println(s"side effect on $currentThread")
  "Hello"
}

val delayed: Task[String] =
  trigger                 // trigger delays execution of source.
    .flatMap(_ => source)

implicit val scheduler: Scheduler = Scheduler.global

delayed.runToFuture foreach println
```

- alternatively with *> (= cats.Apply.productR):

```
import cats.syntax.apply._

val delayed: Task[String] =
  trigger *> source
```

Task#delayResult

- delays signaling of a *Task*'s result for a certain time duration.

```
val source = Task {
  println(s"side effect on $currentThread")
  "Hello"
}

val delayed: Task[String] =
  source
    .delayExecution(2.second)
    .delayResult(3.seconds)

implicit val scheduler: Scheduler = Scheduler.global

delayed.runToFuture foreach println
```

Delay signaling of a *Task*'s result until another *Task* is complete

- delays signaling of a *Task*'s result until another *Task* (the selector) is complete.

```scala
val source = Task {
  println(s"side effect on $currentThread")
  Random.nextInt(5)
}

def selector(x: Int): Task[Int] =
  Task(x).delayExecution(x.seconds)

val delayed: Task[Int] =
  source
    .delayExecution(1.second)
    .flatMap(x => selector(x))    // selector delays result signaling of source.

implicit val scheduler: Scheduler = Scheduler.global

delayed.runToFuture foreach { x =>
  println(s"Result: $x (signaled after at least ${x+1} seconds)")
}
```

17. Parallelism (cats.Parallel)

The problem with *flatMap*

- The three *Task* operations are independent of each other.
- But *Task#flatMap* (for-comprehension) forces them to be sequential, (as if they were dependent).

The problem with *flatMap*

- The three *Task* operations are independent of each other.
- But *Task#flatMap* (for-comprehension) forces them to be sequential, (as if they were dependent).

```scala
def sumTask(from: Int, to: Int): Task[Int] = Task { sumOfRange(from, to) }
def factorialTask(n: Int): Task[BigInt] = Task { factorial(n) }
def fibonacciTask(cycles: Int): Task[BigInt] = Task { fibonacci(cycles) }
```

```scala
val aggregate: Task[BigInt] = for {  // sequential operations based on flatMap
  sum <- sumTask(0, 1000)
  fac <- factorialTask(10)
  fib <- fibonacciTask(10)
} yield sum + fac + fib
```

```scala
implicit val scheduler: Scheduler = Scheduler.global

aggregate runAsync printCallback

def printCallback[A]: Callback[Throwable, A] = new Callback[Throwable, A] {
  def onSuccess(result: A): Unit = println(s"result = $result")
  def onError(ex: Throwable): Unit = println(s"ERROR: ${ex.toString}")
}
```

The problem with *flatMap*

- Composition with *flatMap* (often implicitly in for-comprehensions) enforces sequential, i.e. dependent execution.
- *flatMap* prevents independent / parallel execution.
- Type class *cats.Parallel* allows to switch between sequential (monadic) and parallel (applicative) execution.

```
type ~>[F[_], G[_]] = arrow.FunctionK[F, G]

trait Parallel[M[_], F[_]] {

  // Natural Transformation from the parallel Applicative F[_] to the sequential M
  def sequential: F ~> M

  // Natural Transformation from the sequential Monad M[_] to the parallel Applica
  def parallel: M ~> F
}
```

- Monix *Task* has instance of *cats.Parallel*.
- But it also provides its own (more efficient) impl of *parZip[n]*, *parMap[n]*, *parSequnce*, *parTraverse*.

Replace *flatMap* / for-comprehension ...

Replace *flatMap* / for-comprehension ...

- with *Task#parZip3* followed by *map*

```
val aggregate: Task[BigInt] =               // potentially executed in parallel
  Task.parZip3(sumTask(0, 1000), factorialTask(10), fibonacciTask(10)).map {
    (sum, fac, fib) => sum + fac + fib
  }
```

Replace *flatMap* / for-comprehension ...

- with *Task#parZip3* followed by *map*

```
val aggregate: Task[BigInt] =                 // potentially executed in parallel
  Task.parZip3(sumTask(0, 1000), factorialTask(10), fibonacciTask(10)).map {
    (sum, fac, fib) => sum + fac + fib
  }
```

- with *Task#parMap3*

```
val aggregate: Task[BigInt] =                 // potentially executed in parallel
  Task.parMap3(sumTask(0, 1000), factorialTask(10), fibonacciTask(10)) {
    (sum, fac, fib) => sum + fac + fib
  }
```

Replace *flatMap* / for-comprehension ...

- with *Task#parZip3* followed by *map*

```scala
val aggregate: Task[BigInt] =                       // potentially executed in parallel
  Task.parZip3(sumTask(0, 1000), factorialTask(10), fibonacciTask(10)).map {
    (sum, fac, fib) => sum + fac + fib
  }
```

- with *Task#parMap3*

```scala
val aggregate: Task[BigInt] =                       // potentially executed in parallel
  Task.parMap3(sumTask(0, 1000), factorialTask(10), fibonacciTask(10)) {
    (sum, fac, fib) => sum + fac + fib
  }
```

- with *parMapN* using *cats.syntax.parallel._*

```scala
import cats.syntax.parallel._

val aggregate: Task[BigInt] =                       // potentially executed in parallel
  (sumTask(0, 1000), factorialTask(10), fibonacciTask(10)) parMapN {
    _ + _ + _
  }
```

Sequencing - Overview

Task provides three variants for sequencing (analogous to *Future.sequence*):

Sequencing - Overview

Task provides three variants for sequencing (analogous to *Future.sequence*):

- *Task#sequence*

```
// sequentially converts a *Seq[Task[A]]* to *Task[Seq[A]]*
// the order of side effects is guaranteed as well as the order of results
val taskOfSeq: Task[Seq[Int]] = Task.sequence(seqOfTask)
```

Sequencing - Overview

Task provides three variants for sequencing (analogous to *Future.sequence*):

- *Task#sequence*

```
// sequentially converts a *Seq[Task[A]]* to *Task[Seq[A]]*
// the order of side effects is guaranteed as well as the order of results
val taskOfSeq: Task[Seq[Int]] = Task.sequence(seqOfTask)
```

- *Task#gather*

```
// converts a *Seq[Task[A]]* to *Task[Seq[A]]* in parallel
// the order of side effects is not guaranteed but the order of results is
val taskOfSeq: Task[Seq[Int]] = Task.gather(seqOfTask)
```

Sequencing - Overview

Task provides three variants for sequencing (analogous to *Future.sequence*):

- *Task#sequence*

```
// sequentially converts a *Seq[Task[A]]* to *Task[Seq[A]]*
// the order of side effects is guaranteed as well as the order of results
val taskOfSeq: Task[Seq[Int]] = Task.sequence(seqOfTask)
```

- *Task#gather*

```
// converts a *Seq[Task[A]]* to *Task[Seq[A]]* in parallel
// the order of side effects is not guaranteed but the order of results is
val taskOfSeq: Task[Seq[Int]] = Task.gather(seqOfTask)
```

- *Task#gatherUnordered*

```
// converts a *Seq[Task[A]]* to *Task[Seq[A]]* in parallel
// the order of side effects is not guaranteed, neither is the order of results
val taskOfSeq: Task[Seq[Int]] = Task.gatherUnordered(seqOfTask)
```

Task#sequence

- sequentially converts a *Seq[Task[A]]* to *Task[Seq[A]]*.
- The order of side effects is guaranteed as well as the order of results.

```
val task1 = Task { println("side effect 1"); 1 }
val task2 = Task { println("side effect 2"); 2 }
val seqOfTask: Seq[Task[Int]] = Seq(task1, task2)

val taskOfSeq: Task[Seq[Int]] = Task.sequence(seqOfTask)

implicit val scheduler: Scheduler = Scheduler.global

// We always get the same ordering in the output:
taskOfSeq foreach println
//=> side effect 1
//=> side effect 2
//=> List(1, 2)

taskOfSeq foreach println
//=> side effect 1
//=> side effect 2
//=> List(1, 2)
```

Task#gather

- converts a *Seq[Task[A]]* to *Task[Seq[A]]* in parallel.
- The order of side effects is not guaranteed but the order of results is.

```
val task1 = Task { println("side effect 1"); 1 }.delayExecution(1 second)
val task2 = Task { println("side effect 2"); 2 }.delayExecution(1 second)
val seqOfTask: Seq[Task[Int]] = Seq(task1, task2)

val taskOfSeq: Task[Seq[Int]] = Task.gather(seqOfTask)

implicit val scheduler: Scheduler = Scheduler.global

// There’s potential for parallel execution:
// Ordering of effects is not guaranteed, but the results in the List are ordered.
taskOfSeq foreach println
//=> side effect 1
//=> side effect 2
//=> List(1, 2)

taskOfSeq foreach println
//=> side effect 2
//=> side effect 1
//=> List(1, 2)
```

Task#gatherUnordered

- converts a *Seq[Task[A]]* to *Task[Seq[A]]* in parallel.
- The order of side effects is not guaranteed, neither is the order of results.

```
val task1 = Task { println("side effect 1"); 1 }.delayExecution(1 second)
val task2 = Task { println("side effect 2"); 2 }.delayExecution(1 second)
val seqOfTask: Seq[Task[Int]] = Seq(task1, task2)

val taskOfSeq: Task[Seq[Int]] = Task.gatherUnordered(seqOfTask)

implicit val scheduler: Scheduler = Scheduler.global

// There’s potential for parallel execution:
// Ordering of effects is not guaranteed, results in the List are unordered too.
taskOfSeq foreach println
//=> side effect 1
//=> side effect 2
//=> List(1, 2)

taskOfSeq foreach println
//=> side effect 2
//=> side effect 1
//=> List(2, 1)
```

Traversing - Overview

Task provides three variants for traversing (analogous to *Future.traverse*):

Traversing - Overview

Task provides three variants for traversing (analogous to *Future.traverse*):

- *Task#traverse*

```
// sequentially traverses a *Seq[A]* with a function A => Task[A] to *Task[Seq[A]]
// the order of side effects is guaranteed as well as the order of results
val taskOfSeq: Task[Seq[Int]] = Task.traverse(Seq(1, 2))(i => task(i))
```

Traversing - Overview

Task provides three variants for traversing (analogous to *Future.traverse*):

- *Task#traverse*

```
// sequentially traverses a *Seq[A]* with a function A => Task[A] to *Task[Seq[A]]
// the order of side effects is guaranteed as well as the order of results
val taskOfSeq: Task[Seq[Int]] = Task.traverse(Seq(1, 2))(i => task(i))
```

- *Task#wander*

```
// traverses a *Seq[A]* with a function A => Task[A] to *Task[Seq[A]]* in parallel
// the order of side effects is not guaranteed but the order of results is
val taskOfSeq: Task[Seq[Int]] = Task.wander(Seq(1, 2))(i => task(i))
```

Traversing - Overview

Task provides three variants for traversing (analogous to *Future.traverse*):

- *Task#traverse*

```
// sequentially traverses a *Seq[A]* with a function A => Task[A] to *Task[Seq[A]]
// the order of side effects is guaranteed as well as the order of results
val taskOfSeq: Task[Seq[Int]] = Task.traverse(Seq(1, 2))(i => task(i))
```

- *Task#wander*

```
// traverses a *Seq[A]* with a function A => Task[A] to *Task[Seq[A]]* in parallel
// the order of side effects is not guaranteed but the order of results is
val taskOfSeq: Task[Seq[Int]] = Task.wander(Seq(1, 2))(i => task(i))
```

- *Task#wanderUnordered*

```
// traverses a *Seq[A]* with a function A => Task[A] to *Task[Seq[A]]* in parallel
// the order of side effects is not guaranteed, neither is the order of results
val taskOfSeq: Task[Seq[Int]] = Task.wanderUnordered(Seq(1, 2))(i => task(i))
```

Task#traverse

- sequentially traverses a *Seq[A]* with a function A => Task[A] to *Task[Seq[A]]*.
- The order of side effects is guaranteed as well as the order of results.

```scala
def task(i: Int): Task[Int] = Task { println("side effect  " + i); i }

val taskOfSeq: Task[Seq[Int]] = Task.traverse(Seq(1, 2))(i => task(i))

implicit val scheduler: Scheduler = Scheduler.global

// We always get the same ordering in the output:
taskOfSeq foreach println
//=> side effect 1
//=> side effect 2
//=> List(1, 2)

taskOfSeq foreach println
//=> side effect 1
//=> side effect 2
//=> List(1, 2)
```

Task#wander

- traverses a *Seq[A]* with a function A => Task[A] to *Task[Seq[A]]* in parallel.
- The order of side effects is not guaranteed but the order of results is.

```
def task(i: Int): Task[Int] =
      Task { println("side effect  " + i); i }.delayExecution(1 second)

val taskOfSeq: Task[Seq[Int]] = Task.wander(Seq(1, 2))(i => task(i))

implicit val scheduler: Scheduler = Scheduler.global

// There's potential for parallel execution:
// Ordering of effects is not guaranteed, but the results in the List are ordered.
// list.runToFuture.foreach(println)
taskOfSeq foreach println
//=> side effect 1
//=> side effect 2
//=> List(1, 2)

// list.runToFuture.foreach(println)
taskOfSeq foreach println
//=> side effect 2
//=> side effect 1
//=> List(1, 2)
```

Task#wanderUnordered

- traverses a *Seq[A]* with a function A => Task[A] to *Task[Seq[A]]* in parallel.
- The order of side effects is not guaranteed, neither is the order of results.

```scala
def task(i: Int): Task[Int] =
      Task { println("side effect  " + i); i }.delayExecution(1 second)

val taskOfSeq: Task[Seq[Int]] = Task.wanderUnordered(Seq(1, 2))(i => task(i))

implicit val scheduler: Scheduler = Scheduler.global

// There's potential for parallel execution:
// Ordering of effects is not guaranteed, results in the List are unordered too.
taskOfSeq foreach println
//=> side effect 1
//=> side effect 2
//=> List(1, 2)

taskOfSeq foreach println
//=> side effect 2
//=> side effect 1
//=> List(2, 1)
```

18. TaskApp

TaskApp

- allows you to create an App where you only define a *Task*
- You define the *Task* inside *TaskApp#run*.
- *TaskApp#run* takes the command line args as a *List[String]* and returns a *Task[ExitCode]*.
- No need to invoke one of the run *methods like* runAsync, runToFuture *or* foreach*
- No need to provide an implicit *Scheduler*.

TaskApp

- allows you to create an App where you only define a *Task*
- You define the *Task* inside *TaskApp#run*.
- *TaskApp#run* takes the command line args as a *List[String]* and returns a *Task[ExitCode]*.
- No need to invoke one of the run *methods like* runAsync, runToFuture *or* foreach*
- No need to provide an implicit *Scheduler*.

```
object MyApp extends TaskApp {

  def run(args: List[String]): Task[ExitCode] =

    args.headOption match {
      case Some(name) =>
        Task(println(s"Hello, $name.")).as(ExitCode.Success)
      case None =>
        Task(System.err.println("Usage: MyApp name")).as(ExitCode(2))
    }
}
```

TaskApp - Another example

```
object FibonacciApp extends TaskApp {

  def run(args: List[String]): Task[ExitCode] =

    args.headOption match {

      case None =>
        Task(System.err.println("Usage: MyApp cycles")).as(ExitCode(1))

      case Some(cyclesStr) =>
        Try(cyclesStr.toInt).toEither match {
          case Left(t) =>
            Task(System.err.println("Not a positive number")).as(ExitCode(2))
          case Right(cycles) =>
            defineWork(cycles).as(ExitCode.Success)
        }
    }

    def defineWork(cycles: Int): Task[Unit] = for {
      result <- Task { fibonacci(cycles) }
      _ <- Task(println("\n-----"))
      _ <- Task(println(s"Fibonacci for $cycles cycles is: $result."))
      _ <- Task(println("-----\n"))
    } yield ()
  }
```

19. CompletableFuture

FutureUtils

- is an object with utility functions for *scala.concurrent.Future*.
- Provides (amongst other things) interop with Java's *CompletableFuture*
- *fromJavaCompletable* converts a *CompletableFuture* to a Scala *Future*.
- *toJavaCompletable* converts a Scala *Future* to a *CompletableFuture*.

FutureUtils.fromJavaCompletable

- converts a *CompletableFuture* to a *scala.concurrent.Future*

```
implicit val ec: ExecutionContextExecutor = ExecutionContext.global

val supplier: Supplier[BigInt] = () => fibonacci(6)

def completable: CompletableFuture[BigInt] =
      CompletableFuture.supplyAsync(supplier)

val future: Future[BigInt] = FutureUtils.fromJavaCompletable(completable)

future foreach println
```

Converting *CompletableFuture* to *Task*

```
def completable: CompletableFuture[BigInt] =
      CompletableFuture.supplyAsync(() => fibonacci(6))

def future: Future[BigInt] = FutureUtils.fromJavaCompletable(completable)

val task = Task.deferFutureAction { implicit scheduler => future }

implicit val scheduler: Scheduler = Scheduler.global

task foreach println
```

FutureUtils.toJavaCompletable

- converts a *scala.concurrent.Future* to a *CompletableFuture*

```
implicit val ec: ExecutionContext = ExecutionContext.global

val future: Future[BigInt] = Future { fibonacci(6) }

val completable: CompletableFuture[BigInt] = FutureUtils.toJavaCompletable(future)

val consumer: Consumer[BigInt] = new Consumer[BigInt] {
  override def accept(value: BigInt): Unit = println(value)
}
completable.thenAccept(consumer)
```

Converting *Task* to *CompletableFuture*

```scala
val task: Task[BigInt] = Task { fibonacci(6) }

implicit val scheduler: Scheduler = Scheduler.global

val completable: CompletableFuture[BigInt] =
      FutureUtils.toJavaCompletable(task.runToFuture)

completable.thenAccept((value: BigInt) => println(value))
```

20. Resources Management

Line count example with classic Java File IO

```scala
def doCountLines(in: BufferedReader): Int = {
  var count: Int = 0
  var line: String = in.readLine()
  while (line != null) {
    count += 1
    line = in.readLine()
  }
  count
}

def countLines(file: File): Task[Int] = Task {
  val in = new BufferedReader(new FileReader(file))
  try {
    doCountLines(in)
  } finally {
    in.close()
  }
}

val task: Task[Int] = countLines(new File("README.md"))

implicit val scheduler: Scheduler = Scheduler.global
task foreach println
```

Task#bracket

- can separate resource acquisition, it's usage and it's release

```
def doCountLines(in: BufferedReader): Int = ???     // as before

def countLines(file: File): Task[Int] = {

  val acquire: Task[BufferedReader] = Task {
    new BufferedReader(new FileReader(file))
  }

  acquire.bracket { in =>
    Task { doCountLines(in) }
  } { in =>
    Task { in.close() }
  }
}

val task: Task[Int] = countLines(new File("README.md"))

implicit val scheduler: Scheduler = Scheduler.global
task foreach println
```

cats.effect.Resource

- *Resource.make* implements resource acquisition and release
- provides a method *use* that take a function for resource usage

```
def doCountLines(in: BufferedReader): Int = ???     // as before

def openFile(file: File): Resource[Task, BufferedReader] =
  Resource.make(
    Task(new BufferedReader(new FileReader(file)))
  )( in =>
    Task(in.close())
  )

def countLines(file: File): Task[Int] =
  openFile(file).use { in =>
    Task { doCountLines(in) }
  }

val task: Task[Int] = countLines(new File("README.md"))

implicit val scheduler: Scheduler = Scheduler.global
task foreach println
```

Resource.fromAutoCloseable

- simplifies *Resource.make* by omitting the release function

```scala
def doCountLines(in: BufferedReader): Int = ???     // as before

def openFile(file: File): Resource[Task, BufferedReader] =
  Resource.fromAutoCloseable(
    Task(new BufferedReader(new FileReader(file)))
  )

def countLines(file: File): Task[Int] =
  openFile(file).use { in =>
    Task { doCountLines(in) }
  }

val task: Task[Int] = countLines(new File("README.md"))

implicit val scheduler: Scheduler = Scheduler.global
task foreach println
```

21. Resources

Resources (1/2)

- Code and Slides of this Talk:
  https://github.com/hermannhueck/future-vs-monix-task
- Code and Slides for my own Implementation of IO Monad:
  https://github.com/hermannhueck/implementing-io-monad
- Monix 3.x Documentation
  https://monix.io/docs/3x/
- Monix 3.x API Documentation
  https://monix.io/api/3.0/
- Monix Task: Lazy, Async and Awesome
  Talk by Alexandru Nedelcu at Scala Days 2017
  https://www.youtube.com/watch?v=wi97X8_JQUk
- Converting Scala's Future to Task
  Video Tutorial by Alexandru Nedelcu
  https://monix.io/blog/2018/11/08/tutorial-future-to-task.html
- Presentations on Monix:
  https://monix.io/presentations

Resources (2/2)

- "Concurrency Basics" in cats-effect
  https://typelevel.org/cats-effect/concurrency/basics.html
- Daniels Spiewak's gist on Thread Pools
  https://gist.github.com/djspiewak/46b543800958cf61af6efa8e072bfd5c
- Daniels Spiewak's blog: An IO monad for cats
  https://typelevel.org/blog/2017/05/02/io-monad-for-cats.html
- Discussion about Green Threads and blocking I/O
  https://github.com/typelevel/cats-effect/issues/243
- Best Practice: "Should Not Block Threads"
  https://monix.io/docs/3x/best-practices/blocking.html
- What Referential Transparency can do for you
  Talk by Luka Jacobowitz at ScalaIO 2017
  https://www.youtube.com/watch?v=X-cEGEJMx_4

Thanks for Listening

Q & A

https://github.com/hermannhueck/future-vs-monix-task