Concurrency in Scala and on the JVM

Adam Warski, September 2023
@adamwarski / @softwaremill.social / softwaremill.com

SOFTWAREMILL

What's the problem?

1. Concurrency is hard
   - we'll focus on **local** concurrency

2. Scala is often chosen because of its concurrency offering
   - a couple of good choices

Goal #1:
how to avoid manual concurrency

Goal #2:
know your choices

The contenders

	Library	Code style	Others
Actor systems	Pekko	Imperative / Future-based	Akka
Functional effects	ZIO	Monadic	cats-effect Monix
Loom/virtual threads	Ox	Imperative / Direct	?

Use-case:
HTTP server

The flow

- Incoming "prepare a meal" HTTP request:
  - Lookup required ingredients, **race**:
    - DB query, **retry** 3 times on failure
    - cache lookup
  - For each ingredient, send demand over a WebSocket, in **parallel**
    - messages to each WebSocket must be sent **sequentially**
    - **global** processes accessed by request handlers

ZIO: what is it?

Type-safe, composable asynchronous and concurrent programming for Scala

ZIO: server

```
trait ServerSocket:
  def accept: Task[ClientSocket]

def mealServer(s: ServerSocket) =
  s.accept.flatMap { socket =>
    ...
  }.forever
```

- lazy-evaluated computation descriptions
- sequencing using `.flatMap`
- custom runtime
- `.forever` as computation description combinator

ZIO: server

```scala
def read(
  socket: ClientSocket): Task[HttpRequest] = ???

def write(resp: HttpResponse,
  socket: ClientSocket): Task[HttpRequest] = ???

def prepareMeal(
  req: HttpRequest): Task[HttpResponse] = ???

def mealServer(s: ServerSocket) =
  s.accept.flatMap { socket =>
    val handleSocket = for {
      req  <- read(socket)
      resp <- prepareMeal(req)
      _    <- write(resp, socket)
    } yield ()

    handleSocket.fork
  }.forever
```

- sequencing multiple computations using `for`
- computation definition separate from sequencing
- `.fork` to start background processes
- supervision by default

ZIO: prepareMeal logic

```
case class Ingredient(name: String)
case class Meal(name: String,
  ingredients: List[Ingredient])

def findInDB(mealName: String): Task[Meal] = ???
def findInCache(mealName: String): Task[Meal] = ???

def prepareMeal(
  req: HttpRequest): Task[HttpResponse] =

  val mealName = req.param("meal")

  val meal: Task[Meal] = ZIO.raceAll(
    findInDB(mealName).retry(
      Schedule.spaced(100.millis) &&
      Schedule.recurs(3)
    ),
    List(findInCache(mealName))
  )

  ...
```

ZIO: prepareMeal logic

```
case class Ingredient(name: String)
case class Meal(name: String,
  ingredients: List[Ingredient])

def findInDB(mealName: String): Task[Meal] = ???
def findInCache(mealName: String): Task[Meal] = ???

def prepareMeal(
  req: HttpRequest): Task[HttpResponse] =

  val mealName = req.param("meal")

  val meal: Task[Meal] = ZIO.raceAll(
    findInDB(mealName).retry(
      Schedule.spaced(100.millis) &&
      Schedule.recurs(3)
    ),
    List(findInCache(mealName))
  )

  ...
```

- `.retry` combinator
- `.raceAll` for racing computations
- computations can be interrupted
- `flatMap`s, compatible I/O are interruption points
- integrated with resource management
- test clock for reliable testing

ZIO: prepareMeal logic

```
def prepareMeal(
  req: HttpRequest,
  webSockets: Map[String, Queue[Demand]]
): Task[HttpResponse] =

  val meal: Task[Meal] = ...

  val sendDemand = meal.flatMap { m =>
    ZIO.foreachPar(m.ingredients)(i =>
      webSockets(i.name).offer(Demand(1))
    )
  }

  sendDemand.map(_ => HttpResponse(200, "OK"))
```

- `.foreachPar`: parallel computation description
- communicate with WebSockets through a message queue

ZIO: web socket process

```
trait WebSocket:
  def sendText(text: String): Task[Unit]

def startWebSocketQueue(
  ws: WebSocket): Task[Queue[Demand]] =

  Queue.bounded[Demand](16).flatMap { queue =>
    queue
      .take
      .flatMap { case Demand(amount) =>
        ws.sendText(amount.toString)
      }
      .forever
      .fork
      .map(_ => queue)
  }
```

- `.forever` + `.fork`: a never-ending process
- alternating: take a message, send over WS
- actor!

ZIO: summary

- 🍀 fast
- 🍀 uniform process description: lazy, as a value
- 🍀 fearless refactoring
- 🍀 effortless concurrency: multiple combinators
- 🍀 automatic supervision
- 🍀 testing with timers
- 🍀 principled interruption
- 🍀 flexible modelling of concurrent processes
- 🍀 fiber locals (e.g. for observability)
- 🍀 resource management
- 💔 fibers can silently die: one-way supervision
- 💔 syntax overhead, monadic
- 💔 custom control flow methods
- ⚠️ reduced usability of stack traces

Pekko: what is it?

An open-source framework for building applications that are concurrent, distributed, resilient and elastic

Pekko: server

```
trait ServerSocket:
  def accept: Future[ClientSocket]

def mealServer(s: ServerSocket, as: ActorSystem) =
  s.accept
    .map { socket =>
      ...
    }
    .flatMap(_ => mealServer(s, as))
```

- eagerly-evaluated `Futures`
- sequencing using `.flatMap`
- runtime: submit task to an executor
- forever loops through recursion

Pekko: server

```
def read(
  socket: ClientSocket): Future[HttpRequest] = ???

def write(resp: HttpResponse,
  socket: ClientSocket): Future[HttpRequest] = ???

def prepareMeal(req: HttpRequest,
  as: ActorSystem): Future[HttpResponse] = ???

def mealServer(s: ServerSocket, as: ActorSystem) =
  s.accept.map { socket =>
    for {
      req  <- read(socket)
      resp <- prepareMeal(req, as)
      _    <- write(resp, socket)
    } yield ()
  }.flatMap(_ => mealServer(s, actorSystem))
```

- sequencing multiple computations using `for`
- computations start when defined
- background processing: create `Future`
- supervision in actors, but not in `Future`s

Pekko: prepareMeal logic

```scala
case class Ingredient(name: String)
case class Meal(name: String,
  ingredients: List[Ingredient])

def findInDB(mealName: String): Future[Meal] = ???
def findInCache(name: String): Future[Meal] = ???

def prepareMeal(req: HttpRequest,
  as: ActorSystem): Future[HttpResponse] =

  val mealName = req.param("meal")

  val findInDBFuture = retry(() =>
    findInDB(mealName), attempts = 3, 100.millis)
  val findInCacheFuture = findInCache(mealName)
  val meal: Future[Meal] = raceSuccess(
    findInDBFuture, findInCacheFuture, as)

  ...
```

Pekko: prepareMeal logic

```scala
case class Ingredient(name: String)
case class Meal(name: String,
  ingredients: List[Ingredient])

def findInDB(mealName: String): Future[Meal] = ???
def findInCache(name: String): Future[Meal] = ???

def prepareMeal(req: HttpRequest,
  as: ActorSystem): Future[HttpResponse] =

  val mealName = req.param("meal")

  val findInDBFuture = retry(() =>
    findInDB(mealName), attempts = 3, 100.millis)
  val findInCacheFuture = findInCache(mealName)
  val meal: Future[Meal] = raceSuccess(
    findInDBFuture, findInCacheFuture, as)

  ...
```

- Pekko-provided `retry`
- custom `raceSuccess` method
- starting multiple futures in parallel
- computations **can't** be interrupted
- completion callbacks

Pekko: race logic

```scala
def raceSuccess[T](
    f1: Future[T],
    f2: Future[T],
    actorSystem: ActorSystem
): Future[T] =
  import actorSystem.dispatcher
  val p = Promise[T]()

  def raceBehavior2(e: Throwable) =
    Behaviors.receiveMessage[Either[Throwable, T]] {
      case Left(_) =>
        p.failure(e)
        Behaviors.stopped
      case Right(v: T) =>
        p.success(v)
        Behaviors.stopped
    }

  val raceBehavior1 =
    Behaviors.receiveMessage[Either[Throwable, T]] {
      case Left(e: Throwable) =>
        raceBehavior2(e)
      case Right(v: T) =>
        p.success(v)
        Behaviors.stopped
    }

  val raceActor = actorSystem.spawn(raceBehavior1, s"race-${UUID.randomUUID()}")
  List(f1, f2).foreach(_.onComplete {
    case Success(v) => raceActor.tell(Right(v))
    case Failure(e) => raceActor.tell(Left(e))
  })
  p.future
```

- a short-lived actor collecting responses
- straightforward, but lengthy implementation using the Pekko Typed APIs
- no combinators available out-of-the-box
- **no interruption**

Pekko: prepareMeal logic

```scala
def prepareMeal(
  req: HttpRequest,
  as: ActorSystem
): Future[HttpResponse] =

  val meal: Future[Meal] = ...

  val sendDemand = meal.map { m =>
    m.ingredients.foreach { i =>
      val wsActor = actorSystem.actorSelection(
        as("user").child(s"${i.name}-websocket")
      )
      wsActor ! Demand(1)
    }
  }

  sendDemand.map(_ => HttpResponse(200, "OK"))
```

- sending multiple messages - can be processed in parallel
- side-effecting computations
- dynamically looking up actors only in the untyped variant

Pekko: web socket process

```
trait WebSocket:
  def sendText(text: String): Future[Unit]

def webSocketBehavior(ws: WebSocket): Behavior[Demand] =
  def run(ready: Boolean,
    buffer: Vector[Demand]): Behavior[Demand | Boolean] =

    Behaviors.receive[Demand | Boolean] {
      case (ctx, msg: Demand) =>
        import ctx.executionContext
        if ready then
          ws.sendText(msg.amount.toString).onComplete(_ =>
ctx.self.tell(true))
          run(false, buffer)
        else run(false, buffer :+ msg)
      case (ctx, _: Boolean) =>
        import ctx.executionContext
        buffer match
          case head +: tail =>
            ws.sendText(head.amount.toString)
              .onComplete(_ => ctx.self.tell(true))
            run(false, tail)
          case _ => run(true, buffer)
    }

  run(true, Vector.empty).narrow[Demand]
```

- recursive implementation, returning modified behaviors
- clear, simple to understand API
- however, quite verbose
- manual `Future` integration

Pekko: summary

- 🍀 fast
- 🍀 concurrency in actors through message passing
- 🍀 simple, intuitive API to define `Behavior`s
- 🍀 automatic supervision in actors
- 🍀 very large ecosystem, both for local and distributed concurrency
- ⚠️ lazy evaluated `Behavior`s, eager `Future`s
- ⚠️ `Future`s evaluated at the moment of construction

- ⚠️ partial resource management (only in actors)
- ⚠️ some testing support
- 💔 no supervision for `Future`s
- 💔 no "future-local" values
- 💔 syntax overhead, monadic
- 💔 unusable stack traces
- 💔 no interruption
- 💔 custom control structures

Ox: what is it?

Developer-friendly structured concurrency library for the JVM

Ox: server

```
trait ServerSocket:
  def accept: ClientSocket

def mealServer(s: ServerSocket) =
  scoped {
    forever {
      val socket = s.accept
      ...
    }
  }
```

- direct style, eager evaluation
- structured concurrency: syntactical scopes determine thread lifetime
- `forever` method taking a computation by-name

Ox: server

```
def read(
  socket: ClientSocket): HttpRequest = ???

def write(resp: HttpResponse,
  socket: ClientSocket): HttpRequest = ???

def prepareMeal(
  req: HttpRequest): HttpResponse = ???

def mealServer(s: ServerSocket) =
  scoped {
    forever {
      val socket = s.accept
      fork {
        val req = read(socket)
        val resp = prepareMeal(req)
        write(resp, socket)
      }
    }
  }
```

- `fork` starts a background processes
- optional supervision
- asynchronous runtime built into the JVM

Ox: prepareMeal logic

```scala
case class Ingredient(name: String)
case class Meal(name: String,
  ingredients: List[Ingredient])

def findInDB(mealName: String): Meal = ???
def findInCache(mealName: String): Meal = ???

def prepareMeal(
  req: HttpRequest): HttpResponse =

  val mealName = req.param("meal")

  val meal: Meal = raceSuccess(
    retry(3, 100.millis)(findInDB(mealName)))(
    findInCache(mealName))

  ...
```

Ox: prepareMeal logic

```
case class Ingredient(name: String)
case class Meal(name: String,
  ingredients: List[Ingredient])

def findInDB(mealName: String): Meal = ???
def findInCache(mealName: String): Meal = ???

def prepareMeal(
  req: HttpRequest): Task[HttpResponse] =

  val mealName = req.param("meal")

  val meal: Meal = raceSuccess(
    retry(3, 100.millis)(findInDB(mealName)))(
    findInCache(mealName))

  ...
```

- `retry`, `raceSuccess` methods taking lazy-evaluated computations
- interruption using Java's interruption
- blocking operations are interruption points
- `try-finally`/`scoped` resource management

Ox: interruptions

```
fork {
  forever {
    try
      val m = nextMessage()
      processMessage(m)
    catch
      case e: Exception =>
        logger.error(e)
        redeliver(m)
  }
}
```

- interruption using Java's `InterruptedException`

Ox: interruptions

```
fork {
  forever {
    try
      val m = nextMessage()
      processMessage(m)
    catch
      case NonFatal(e) =>
        logger.error(e)
        redeliver(m)
  }
}
```

- needs discipline
- tricky integration with third-party libraries

Ox: prepareMeal logic

```
def prepareMeal(
  req: HttpRequest,
  webSockets: Map[String, Sink[Demand]]
): HttpResponse =

  val meal: Meal = ...

  par(meal.ingredients.map { i => () =>
    ingredientWebSockets(i.name).send(Demand(1))
  })

  HttpResponse(200, "OK")
```

- `par`: takes a list of lazily-evaluated computation
- communicate with WebSockets through a channel

Ox: web socket process

```
trait WebSocket:
  def sendText(text: String): Unit

def startWebSocketChannel(
  ws: WebSocket)(using Ox): Sink[Demand] =

  val c = Channel[Demand](16)
  forkDaemon {
    repeatWhile {
      c.receive() match
        case e: ChannelClosed.Error =>
         throw e.toThrowable
        case ChannelClosed.Done => false
        case Demand(amount) =>
          ws.sendText(amount.toString)
          true
    }
  }
  c
```

- `fork` a process receiving from the channel, return a sink
- alternating: take a message, send over WS
- again, actor-like
- needs to be run within a scope

Ox: summary

- 🍀 no syntax overhead
- 🍀 structured concurrency
- 🍀 effortless concurrency: multiple combinators
- 🍀 optional supervision
- 🍀 usable stack traces
- 🍀 built-in control structures
- 🍀 flexible modelling of concurrent processes
- 🍀 scope locals (e.g. for observability)

- ⚠️ mixed eager / lazy evaluation
- ⚠️ Java's exception-based interruption
- ⚠️ some resource management, but too easy to use not safely
- 💔 no type-safety for errors, I/O
- 💔 reliance on system timers
- 💔 incomplete API, experimental implementation

In summary

	Concurrency	Syntax overhead	Supervision	Interruptions	Lazy/eager	Control flow methods	Testing	Refactoring	Maturity/ ecosystem	Resource safety
ZIO	🍀	💔	⚠️ one-way - parent->child	🍀	🍀 always lazy	⚠️	🍀	🍀	🍀	🍀
Pekko	⚠️ actors & Futures, no combinators	💔	⚠️ only in actors	💔	⚠️ mixed eager/lazy	⚠️	⚠️	⚠️	🍀	⚠️
Ox	🍀	🍀	🍀 structured concurrency	⚠️	⚠️ mixed eager/lazy	🍀	💔	⚠️	💔	⚠️

Avoiding concurrency

- **High-level methods**, such as `race` or `par`
- **Actors**
- **Message-passing** - ubiquitous
- **Streaming**
  - all three libraries have a high-level streaming offering
  - always preferred to manual forks / actors
  - especially when multiple input elements need to be combined

SOFTWAREMILL

VIRTUSLAB

State of Scala survey

Links & resources

- Scalaz 8 IO vs Akka (typed) actors vs Monix (part 1)
- Implementing Raft using Project Loom
- Two types of futures
- Go-like channels using Project Loom and Scala
- ZIO, cats-effect, Akka, Pekko, Ox docs
- Effects: to be or not to be?

IO.pure("Thank you!")

@adamwarski / @softwaremill.social / softwaremill.com

SOFTWAREMILL