Scala School

Лекция 7: Асинхронные вычисления

Параллелизм в Scala. scala.concurrent.duration

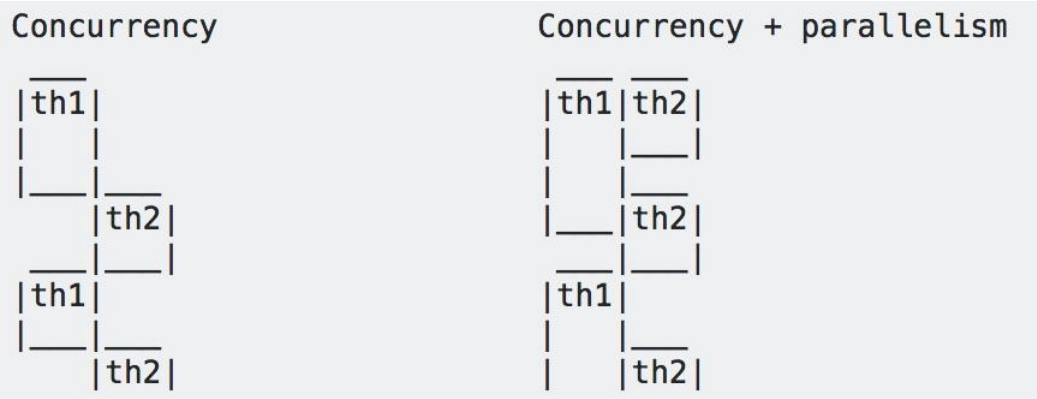
- concurrency vs parallelism
- Future
- ExecutionContext
- Duration, FiniteDuration
- Promise

concurrency vs parallelism

- Concurrency (одновременность, конкурентность) - условие, которое существует, когда два потока независимо могут добиваться какой-либо прогресса, например внутри приложения. Например, многозадачность на процессоре с одним ядром. Более общее понятие, чем паралеллизм.

- Parallelism (паралеллизм) - когда несколько задач буквально выполняются в одно и то же время. Многозадачность на процессоре с несколькими ядрами.

concurrency vs parallelism



Future

```
object Future {
  def apply[T](body: => T)(implicit ctx: ExecutionContext): Future[T]
}
```

Future

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

object Future1 extends App {
  val printFuture = Future {
    println("Hello from future")
  }
  Await.result(printFuture, Duration.Inf)
}
```

Future

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

object Future2 extends App {
  val calcFuture = Future {
    Thread.sleep(1000)
    1000
  }
  val stringFuture = calcFuture.map { i =>
    s"oh, it is $i!"
  }
  val string = Await.result(stringFuture, Duration.Inf)
  println(string)
}
```

Future

```
import scala.concurrent.ExecutionContext.Implicits.global
object Future3 extends App {
  val calcFuture = Future {
    Thread.sleep(1000); 1000
  }
  val stringFuture = Future {"Hello from second future"}
  val f = for {
    calcResult <- calcFuture
    stringResult <- stringFuture
  } yield {
    println(s"$calcResult $stringResult")
  }
  Await.result(f, Duration.Inf)
}
```

Future

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

object Future4 extends App {
  val calcFuture = Future { Thread.sleep(1000); 1000 }
  def stringFuture(i: Int) = Future { s"$i!!!" }
  val f = for {
    calcResult <- calcFuture
    stringResult <- stringFuture(calcResult)
  } yield {
    println(s"$calcResult $stringResult")
  }
  Await.result(f, Duration.Inf)
}
```

ExecutionContext

- ExecutionContext.*fromExecutor*(e: ExecutorService)
- ExecutionContext.*fromExecutorService*(e: Executor)

java.util.concurrent.Executors

- newFixedThreadPool(n)
- newSingleThreadExecutor()
- newCachedThreadPool()
- newSingleThreadScheduledExecutor()
- java.util.concurrent.ForkJoinPool
- scala.concurrent.forkjoin.ForkJoinPool

ExecutionContext

```
object Future5 extends App {
  val e1 = ExecutionContext.fromExecutorService(Executors.newFixedThreadPool(1))
  implicit val e2 = ExecutionContext.fromExecutor(Executors.newFixedThreadPool(1))

  val calcFuture = Future { Thread.sleep(1000); 1000 }(e1)
  val stringFuture = Future { Thread.sleep(1000); "Hello from second future"}(e2)
  val f = for {
    calcResult <- calcFuture
    stringResult <- stringFuture
  } yield {
    println(s"$calcResult $stringResult")
  }
  Await.result(f, Duration.Inf)
}
```

Duration, FiniteDuration

```
import scala.concurrent.duration._

val duration = Duration(100, MILLISECONDS) // FiniteDuration = 100 milliseconds
val duration = Duration(100, "millis") // FiniteDuration = 100 milliseconds

duration.toNanos // 100000000
duration < 1.second // res1: Boolean = true
duration <= Duration.Inf // res2: Boolean = true
```

Duration, FiniteDuration

- Duration.Inf
- Duration.MinusInf
- Duration.Undefined
- Duration.Zero

Callbacks

```
object Future6 extends App {
  val calcFuture = Future { Thread.sleep(1000); 1000 }
  val calcFailureFuture = Future { throw new RuntimeException("Error !!!") }
  calcFuture onComplete {
    case Success(i) =>
      println(i)
    case Failure(f) =>
      println(f.getMessage)
  }
//  deprecated calcFailureFuture onFailure
//  deprecated calcFailureFuture onSuccess
  Thread.sleep(2000)
}
```

Other

```
object Future7 extends App {
  val calcFailureFuture = Future.failed(new RuntimeException)
  val calcSuccess = Future.successful(1)
  println(calcFailureFuture.value) // Some(Failure(java.lang.RuntimeException))
  println(calcFailureFuture.isCompleted) // true
  println(calcFailureFuture.failed.value) // Some(Success(java.lang.RuntimeException))
  println(calcSuccess.value) // Some(Success(1))
}
```

Future companion object methods

- fromTry[T](result: Try[T]): Future[T]
- find[T](futures:Iterable[Future[T]])(p: T => Boolean)(implicit e: ExecutionContext): Future[Option[T]]
- traverse - Asynchronously and non-blockingly transforms a `TraversableOnce[A]` into a `Future[TraversableOnce[B]] using the provided function
- sequence - Simple version of Future.traverse. Asynchronously and non-blockingly transforms a TraversableOnce[Future[A]] into a Future[TraversableOnce[A]]. Useful for reducing many Futures into a single Future.
- firstCompletedOf[T](futures: TraversableOnce[Future[T]])(implicit e: ExecutionContext): Future[T]
- foldLeft[T, R](futures: Iterable[Future[T]])(zero: R)(op: (R, T) => R)(implicit e: ExecutionContext): Future[R]
- reduceLeft[T, R >: T](futures:Iterable[Future[T]])(op: (R, T) => R)(implicit executor: ExecutionContext): Future[R]

Future companion object methods

```
object Future8 extends App {
  val futures = for { i <- 1 to 10 } yield Future.successful(i)
  val r1 = Future.sequence(futures)
  val r2 = Future.sequence(futures :+ Future.failed(new RuntimeException))
  println(r1.value) // Some(Success(Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)))
  println(r2.value) // Some(Failure(java.lang.RuntimeException))
}
```

Promise

```
object Promise1 extends App {
  val p = Promise[Int]()
  val f = p.future
  val producer = Future {
    p success 1
  }
  val consumer = Future {
    f onSuccess {
      case r => println(r) // 1
    }
  }
  Thread.sleep(1000)
}
```

Promise

```
object Promise2 extends App {
  val f = Future { 1 }
  val p = Promise[Int]()
  p tryCompleteWith f
  println(p trySuccess 2) // false
  println(p tryFailure new RuntimeException) // false
  p.future onSuccess {
    case x => println(x) // 1
  }
  Thread.sleep(1000)
}
```

Async

```
def combined: Future[Int] = async {
  val future1 = slowCalcFuture
  val future2 = slowCalcFuture
  await(future1) + await(future2)
}
```

Async

```
def slowCalcFuture: Future[Int] = ...
val future1 = slowCalcFuture
val future2 = slowCalcFuture
def combined: Future[Int] = for {
  r1 <- future1
  r2 <- future2
} yield r1 + r2
```

Материалы

- http://docs.scala-lang.org/overviews/core/futures.html
- http://docs.scala-lang.org/sips/completed/futures-promises.html
- http://doc.akka.io/docs/akka/snapshot/scala/futures.html
- http://danielwestheide.com/blog/2013/01/09/the-neophytes-guide-to-scala-part-8-welcome-to-the-future.html
- http://danielwestheide.com/blog/2013/01/16/the-neophytes-guide-to-scala-part-9-promises-and-futures-in-practice.html
- https://ru.coursera.org/learn/parprog1
-