Concurrency in Scala

Mikhail Mutcianko, Alexey Shcherbakov

СПБгУ, СП

8 апреля 2021

Concurrent, Parallel, Asynchronous

Concurrent vs Parallel computations

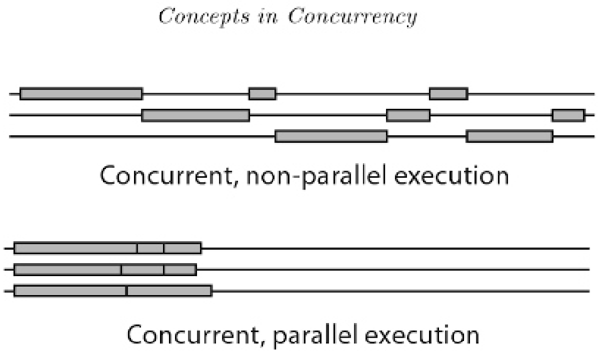
Concurrent

Concurrent computing is a form of computing in which several computations are executed during overlapping time periods — instead of sequentially, with one completing before the next.

Parallel

Parallel computations must also advance simultaneously

Concurrent vs Parallel computations



Synchronous execution

In synchronous model of execution for any statement S of a linear top-down program flow, computation of all statements that are defined before S must finish by the time S is started.

Synchronous execution

In synchronous model of execution for any statement S of a linear top-down program flow, computation of all statements that are defined before S must finish by the time S is started.

```
val a = longComputeA()
val b = longComputeB() // a is initialized

longComputeAB(a,b) // both a and b are computed
```

Asynchronous execution

Asynchronous computations drop the «happens before» requirement. Async model allows to continue the next statement even if the previous has not finished yet.

Asynchronous execution

```scala
println("starting A")
val a = Future { longComputeA(); println("A finished") }
println("starting B")
val b = Future { longComputeB(); println("B finished") }
println("starting AB")
longComputeAB(a,b)
```

Asynchronous execution

```scala
println("starting A")
val a = Future { longComputeA(); println("A finished") }
println("starting B")
val b = Future { longComputeB(); println("B finished") }
println("starting AB")
longComputeAB(a,b)
```

```
starting A
starting B
starting AB
A finished
B finished
```

Concurrency Primitives

Runnable and Callable

Low-level computation abstractions are defined as traits

```scala
// computation does NOT return a value
trait Runnable {
  def run(): Unit
}

// computation returns a value
trait Callable[V] {
  def call(): V
}
```

Thread

Scala concurrency is built on top of the Java concurrency model.

A Thread takes a Runnable. You have to call start on a Thread in order for it to run the Runnable.

```
val hello = new Thread(new Runnable {
  def run() {
    println("hello world")
  }
})

hello.start
// -> hello world
```

Thread

Scala concurrency is built on top of the Java concurrency model.

A Thread takes a Runnable. You have to call start on a Thread in order for it to run the Runnable.

```scala
val hello = new Thread(() => println("hello world"))

hello.start
// -> hello world
```

Futures

Scala Future

Future is an abstraction capturing a computation process

A computation can be in one of the following **three** states:

- unfinished
- successful
- failed

Future Methods

Common

```
def Future.apply(f: =>A): Future[A]
```

- accepts a code block
- starts computation eagerly

Future Methods

Common

```scala
def Future.apply(f: =>A): Future[A]
```

- accepts a code block
- starts computation eagerly

```scala
def onComplete(cb: Try[A] =>B): Unit
```

- provides a callback for a finished computation

Future Methods

Common

```scala
def Future.apply(f: =>A)(implicit ec: ExecutionContext): Future[A]
```

- accepts a code block
- starts computation eagerly

```scala
def onComplete(cb: Try[A] =>B)(implicit ec: ExecutionContext): Unit
```

- provides a callback for a finished computation

ExecutionContext

An *ExecutionContext* is an abstraction which implementations must provide a concrete mechanism of executing computations represented by Scala's `Future`. The intent of ExecutionContext is to lexically scope code execution

- built-in implementation:

  `scala.concurrent.ExecutionContext.Implicits.global`
- based on Java's `ForkJoinPool`
- automatically scales to the number of CPU cores

Working with results

There are several ways of processing an async computation's result:

- block current thread and get a result
- execute a callback when a computation finishes
- compose another computation

Blocking the result

An asynchronous computation can be turned into a synchronous one by using `Await.result`

This method will block the current thread until:

- the Future completes(successfully or not)
- a timeout occurs

Blocking the result

Example

```scala
val greetingFuture = Future {
  Thread.sleep(1000)
  "Hello"
}

val greeting: String = Await.result(greetingFuture, Duration.Inf)

println(s"Result: $greeting")
```

Blocking the result

Poor practice

Blocking on a future is strongly discouraged for the sake of performance and for the prevention of deadlocks. Callbacks and combinators on futures are a preferred way to use their results.

Blocking the result

Deadlock example

```scala
implicit val ec = ExecutionContext
  .fromExecutor(Executors.newFixedThreadPool(1))

def addOne(x: Int) = Future(x + 1)

def multiply(x: Int, y: Int) = Future {
  val a = addOne(x)
  val b = addOne(y)
  val result = for (r1 <- a; r2 <- b) yield r1 * r2

  // this will dead-lock
  Await.result(result, Duration.Inf)
}
```

Processing the result of a Future can be done in a completely non-blocking way by providing a callback using: `Future.onComplete[U](f: (Try[T]) => U): Unit`

```scala
val f: Future[List[String]] = Future {
  session.getRecentPosts
}

f onComplete {
  case Success(posts) => for (post <- posts) println(post)
  case Failure(t) => println("An error has occurred: " + t.getMessage)
}
```

Using callbacks

foreach

In the case where only successful results need to be handled, the `foreach` callback can be used:

```scala
val f: Future[List[String]] = Future {
  session.getRecentPosts
}

f foreach { posts =>
  for (post <- posts) println(post)
}
```

Callback hell

Chaining computations using callbacks is achieved by nesting creation of `Futures`.

This can lead to severe cases of LOP - Ladder Oriented Programming also known as «callback hell»

Callback hell

Example

```
queryDb(8612).onComplete {
  case Failure(ex: Exception) =>
    println(s"Operation failed with $ex")
  case Success(fileName: String) =>
    loadFileAsync(fileName).onComplete {
      case Failure(ex: Exception) =>
        println(s"Operation failed with $ex")
      case Success(url: String) =>
        loadPageAsync(url).onComplete {
          case Failure(ex: Exception) => println(s"Operation failed with $ex")
          case Success(text: String) => Future { ... }
      ...
}
```

In Scala a `Future[+A]` is a monad, providing the following methods:

- `def map[B](f: A => B): Future[B]`
- `def flatMap[B](f: A => Future[B]): Future[B]`
- `def withFilter(f: A => Boolean): Future[A]`

Functional composition of Futures

Example

Monadic composition style:

```
queryDb(8612)
  .flatMap(fileName  => loadFileAsync(fileName))
  .flatMap(url       => loadPageAsync(url))
  .flatMap(pageText  => println(pageText))
```

Functional composition of Futures

Example

Monadic composition style:

```
queryDb(8612)
  .flatMap(fileName  => loadFileAsync(fileName))
  .flatMap(url       => loadPageAsync(url))
  .flatMap(pageText  => println(pageText))
```

For-comprehension style:

```
for {
  fileName  <- queryDb(8612)
  url       <- loadFileAsync(fileName)
  pageText  <- loadPageAsync(url)
} println(pageText)
```

Error handling

- `def recover[U >: T](pf: PartialFunction[Throwable, U]): Future[U]`

  Creates a new future that will handle any matching throwable that this future might contain.
- `def recoverWith[U >: T](pf: PartialFunction[Throwable, Future[U]]): Future[U]`

  Same as `recover`, but composes another future instead of a value
- `def fallbackTo[U >: T](that: Future[U]): Future[U]`

  Creates a new future which holds the result of this future if it was completed successfully, or, if not, the result of the that future if that is completed successfully.
- `def failed: Future[Throwable]`

  Returns a Future with an exception as a result value if the original one has failed

Error handling

Example

```scala
Future (6 / 0) recover { case e: ArithmeticException => 0 } // result: 0

val f = Future { Int.MaxValue }
Future (6 / 0) recoverWith { case e: ArithmeticException => f } // result: Int.MaxValue

val f = Future { throw new RuntimeException("failed") }
val g = Future { 5 }
val h = f fallbackTo g
h foreach println // Eventually prints 5

val g = Future { 2 / 0 }
for (exc <- g.failed) println(exc) // result: java.lang.ArithmeticException: / by zero
```

Future aggregation

- `def traverse(in: M[A])(fn: (A) => Future[B]): Future[M[B]]`

  Asynchronously and non-blockingly transforms a `IterableOnce[A]` into a `Future[IterableOnce[B]]` using the provided function `A => Future[B]`
- `def sequence(in: M[Future[B]]): Future[M[B]]`

  Transforms a sequence of futures into a future of sequences
- `def zip(other: Future[B]): Future[(A, B)]`

  Creates a single future of Tuple2 from two futures
- `def foldLeft(futures: Iterable[Future[T]])(zero: R)(op: (R, T) => R): Future[R]`
- `def reduceLeft(futures: Iterable[Future[T]])(op: (R, T) => R): Future[R]`

Promise

Promise is an API for creating Futures with a controllable state. Promises *complete* the Futures they produce(by ”completing“ the promise)

The generated Furture state can be controlled with the following:

- `complete` / `completeWith`
- `tryComplete` / `tryCompleteWith`
- `success` / `trySuccess` / `failure` / `tryFailure`

Promise

Example

```
val p = Promise[T]()
val f = p.future

val producer = Future {
  val r = produceSomething()
  p success r
  continueDoingSomethingUnrelated()
}

val consumer = Future {
  startDoingSomething()
  f foreach { r => doSomethingWithResult() }
}
```

Promise assignment semantics

Promises have single-assignment semantics. As such, they can be completed only once.

Calling success on a promise that has already been completed (or failed) will throw an IllegalStateException

Parallel Collections

Scala parallel collections

Scala provides an easy way of converting any sequential collection to parallel with `.par`

Scala parallel collections

Scala provides an easy way of converting any sequential collection to parallel with `.par`

```scala
val list = (1 to 10000).toList
list.par.map(_ + 42)
```

"out-of-order" semantics of parallel collections lead to the following two implications:

- Side-effecting operations can lead to non-determinism
- Non-associative operations lead to non-determinism

Practice