Futures

Asynchronous Programming in Scala

Agenda

1. Futures Intro
2. Future States
3. Composing Futures
4. Other Future Operations
5. Promises
6. Adapting Java Futures (and Other Async)
7. Common Future Patterns
8. Alternatives to Futures

Futures

- Futures are a lightweight parallelism API provided in the Scala core API
- While not without some niggles, they are ubiquitous and reliable, and hence widely used

Common Imports:

```
import scala.concurrent._
```

this brings in the main Futures API, including `Future` itself

```
import scala.concurrent.duration._
```

This brings a DSL for specifying times and timeout

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

When working with futures, an ExecutionContext is required (usually implicit) to provide thread pool configuration. `Implicits.global` is handy when you don't have another source.

Creating a Future

This is quite easy using the `Future` factory method

```
val futureInt = Future(1 + 1)
```

Can also use curlies for multi-line

```
val futureResult = Future {
  val a = someLongCalculation()
  val b = someOtherCalculation(a)
  b.refineResult()
}
```

If you already have results (or a failure) you can create success or failure immediately with

```
Future.successful(result)
// or
Future.failed(someException)
// also
Future.fromTry(Try(1 / 0))
```

Some Initial Rules

- Never block or Await in a future (this includes `Thread.sleep()`) - for testing or at the very top level you sometimes have to ignore this advice, but nowhere else!
- Try not to mix `ExecutionContexts` unless you know what you are doing
- Futures start executing immediately after they are created (do if you don't want this, make it lazy with a function)
- Scala futures are not cancellable (but if you run in the context of an actor system, you can terminate that)

Future States

(for demo purposes also I am allowed to use `Await` and `Thread.sleep()`)

```
val f1 = Future { Thread.sleep(1000); 10}

f1.value        // Option[scala.util.Try[Int]] = None
f1.isCompleted  // Boolean = false
// While None is in the value, the Future is not yet resolved either way

Thread.sleep(1000)  // force a pause to wait for the future to complete

f1.value        // Option[scala.util.Try[Int]] = Some(Success(10))
f1.isCompleted  // Boolean = true
// Completed successfully, type is whatever the Future block evaluated to

val f2 = Future { 1 / 0 }
Thread.sleep(10)  // micro pause
f2.value // Option[scala.util.Try[Int]] =
         // Some(Failure(java.lang.ArithmeticException: / by zero))
```

Composing Futures

Probably the single most useful thing about Futures is that they compose with `map` and `flatMap` while remaining asynchronous:

```
val f1 = Future { Thread.sleep(1000); 10}
val f2 = f1.map(_ * 10)

f1.value        // None
f1.isCompleted  // false
f2.value        // None
f2.isCompleted  // false

Thread.sleep(1000)

f1.value        // Some(Success(10))
f1.isCompleted  // true
f2.value        // Some(Success(100))
f2.isCompleted  // true
```

`flatMap` also works asynchronously. Combining with `for` expressions is even cooler

Futures with for

Let's say we have a simple expression like this

```
val (a, b, c) = (1, 2, 3)
val s = "The answer is"

val sum = a + b + c
s"$s $sum"   // res11: String = The answer is 6
```

We can apply a simple re-writing pattern to make this fully async (and this trick always works)

```
val fRes = for {
  a <- fa  // where these are future results of a, b, c and s
  b <- fb
  c <- fc
  s <- fd
} yield {
  val sum = a + b + c  // the exact same code as before moved into yield
  s"$s $sum"
}
```

Async Evaluation

```
val fa = Future(1)
val fb = Future { Thread.sleep(1000); 2 }
val fc = Future(3)
val fd = Future { Thread.sleep(500); "The answer is"}

val fRes = for {
  a <- fa
  b <- fb
  c <- fc
  s <- fd
} yield {
  val sum = a + b + c
  s"$s $sum"
}

fRes.isCompleted      // false
fRes.value            // None

Thread.sleep(1000)
fRes.isCompleted      // true
fRes.value            // Some(Success(The answer is 6))
```

Forcing a Result

Using `for` and other combinators you can avoid blocking in almost all circumstances, but eventually you will need to get the answer. Use `Await.result` or `Await.ready` for this.

```
val success = Future( 2 / 1 )
val failure = Future( 1 / 0 )

Await.ready(failure, 1.second)

failure.value

failure.failed
```

`Await.ready` waits for the Future to be resolved one way or the other before continuing, but does not evaluate the result. If you used `Await.result` in this example, an Exception would be thrown at that point.

Other Future Operations

In addition to `map` and `flatMap` there are a number of other future operations:

```
val fa: Future[Any] = Future(10)
val fi = fa.collect {
  case i: Int => i
}

val ffi = fi.filter(_ > 11)

Await.ready(fi, 1.second)
Await.ready(ffi, 1.second)
```

```
ffi.transform(i => i * 5, {ex =>
  println(ex.getMessage)
  throw new RuntimeException("it failed to filter", ex)
})
```

More Operations

```
val f6 = Future(2)
val f7 = f6.andThen {  // for event side effects
  case Success(i) if i % 2 == 0 => println(s"it's even")
}

Await.result(f7, 1.second)  // It's even, Int: 2
```

```
f7.onComplete {
  case Success(i) => println(s"It worked, and the answer is $i")
  case Failure(ex) => println(s"It failed: ${ex.getMessage}")
}
```

```
f7.foreach(i => println(s"Got an $i"))  // simpler side effect for success
f7.failed.foreach(ex => println(ex.getMessage))  // and for failure
```

Recovering from Failures

```
val failedFuture = Future.failed(new RuntimeException("nah"))
failedFuture.fallbackTo(Future.successful(0))
```

or

```
val ff = Future.failed(new IllegalArgumentException("nope!"))

val fr = ff.recover {
  case _: IllegalArgumentException => 22
}
```

or

```
val ff2 = Future.failed(new IllegalStateException("Again, nope!"))
val fr2 = ff2.recoverWith {
  case _: IllegalArgumentException => Future.successful(22)
}
```

Dealing with Multiple Futures

```
val nums = List(1,2,3,4,5)
def square(i: Int): Future[Int] = Future(i * i)

val futs: List[Future[Int]] = nums.map(square)
val futList = Future.sequence(futs)
Await.ready(futList, 1.second)
// Future[List[Int]] = Future(Success(List(1, 4, 9, 16, 25)))

val futList2 = Future.traverse(nums)(square)
Await.ready(futList2, 1.second)
// Future[List[Int]] = Future(Success(List(1, 4, 9, 16, 25)))
```

A map over a Seq, followed by a `Future.sequence` can be replaced by a `Future.traverse`

`Future.sequence` takes any `Seq[Future[T]]` and turns it into a `Future[Seq[T]]` without blocking (also works for `Sets`)

Other Future Sequence Operations

```
val ft1 = Future { Thread.sleep(10); 10 }
val ft2 = Future { Thread.sleep(5); 5 }
val ft3 = Future { Thread.sleep(20); 20 }
val sft = List(ft1, ft2, ft3)
Await.ready(Future.firstCompletedOf(sft), 1.second)   // Future(Success(5))

Await.ready(Future.foldLeft(sft)(0)(_ + _), 1.second) // Future(Success(35))

Await.ready(Future.reduceLeft(sft)(_ + _), 1.second)  // Future(Success(35))
```

Promises

A promise is the "server" to the "client's" Future

By creating a Promise you create a socket into which you can send something

You can also obtain the Future from the Promise and hand that to someone else

When you put the value (or failure) into the Promise, the Future is resolved with that outcome

```
val promise = Promise[Int]
val future = promise.future

future.isCompleted  // false
future.value        // None

promise.success(10) // fulfill the promise

future.isCompleted  // true
future.value        // Some(Success(10))
```

A Broken Promise

```
val promise2 = Promise[Int]
val future2 = promise2.future

promise2.failure(new IllegalStateException("oops"))

future2.isCompleted  // true
future2.value        // Some(Failed(IllegalStateException: oops))
```

A Promise allows you to easily adapt another asynchronous event into a Scala Future

Working with Java's Futures

For Java's `CompletableFutures`, there is already an adapter

```
import java.util.concurrent.CompletableFuture
import java.util.function.Supplier

val supp = new Supplier[Int] {
  def get: Int = {
    Thread.sleep(500)
    10
  }
}
val cf = CompletableFuture.supplyAsync(supp)
```

```
import scala.compat.java8.FutureConverters._

val sf2 = cf.toScala

Await.result(sf2, 1.second)   // Int: 10
```

Future Patterns - Batching

```
def calc(i: Int): Future[Int] = Future {
  println(s"Calculating for $i")
  Thread.sleep(500)
  i * i
}

def processSeq(xs: Vector[Int]): Future[Vector[Int]] = {
  val allFutures: Vector[Future[Int]] = xs.map(calc)
  Future.sequence(allFutures)
}
```

- Scala's Execution Context will prevent too many threads running at once
- But you can still overwhelm other resources, or run out of memory, etc.
- Need a way to batch up Futures into groups

Future Batching - foldLeft and flatMap

Combining `foldLeft` and `flatMap` makes this fairly easy

```
def processSeqBatch(xs: Vector[Int], batchSize: Int): Future[Vector[Int]] = {
  val batches = xs.grouped(batchSize)
  val start = Future.successful(Vector.empty[Int])
  batches.foldLeft(start) {(accF, batch) =>
    for {
      acc <- accF
      batchRes <- processSeq(batch)
    } yield acc ++ batchRes
  }
}

val nums = (1 to 20).toVector

val f = processSeq(nums)
Await.result(f, 20.seconds)  // starts all 20 at once

val f2 = processSeqBatch(nums, 2)
Await.result(f2, 20.seconds) // waits for each 2 to finish before starting next
```

Future Patterns - Retrying

An example failing call that eventually succeeds:

```
import scala.util.Try

var time = 0
def resetTries(): Unit = time = 0

def calc(): Int = {
  if (time > 3) 10 else {
    time += 1
    throw new IllegalStateException("not yet")
  }
}

Try(calc())   // fail
Try(calc())   // fail
Try(calc())   // fail
Try(calc())   // fail
Try(calc())   // success(10)

resetTries() // back to not working
```

Retrying Futures (naive)

```
def fCalc(): Future[Int] = Future(calc())

val f2 = fCalc().
  fallbackTo(fCalc()).
  fallbackTo(fCalc()).
  fallbackTo(fCalc()).
  fallbackTo(fCalc())

Await.ready(f2, 10.seconds)  // success(10)
```

But this is clunky, we can do better

Retrying Futures (loop)

```
def retry[T](op: => T, retries: Int): Future[T] =
  Future(op) recoverWith { case _ if retries > 0 => retry(op, retries - 1) }

resetTries()

val f3 = retry(calc(), 3)
Await.ready(f3, 10.seconds)  // failure

resetTries()

val f4 = retry(calc(), 5)
Await.ready(f4, 10.seconds)  // success
```

Retrying with Back-off

```
object RetryBackoff extends App {
  import akka.actor._
  import akka.pattern.after

  val as = ActorSystem("timer")

  val scheduler: Scheduler = as.scheduler
  val ec: ExecutionContext = as.dispatcher

  def retryBackoff[T](op: => T, backoffs: Seq[FiniteDuration])
      (implicit s: Scheduler, ec: ExecutionContext): Future[T] =
    Future(op)(ec) recoverWith {
      case _ if backoffs.nonEmpty =>
        after(backoffs.head, scheduler)(retryBackoff(op, backoffs.tail))
    }

// ...
```

Retrying with Back-off part 2

```
  var time = 0

  def calc(): Int = {
    if (time > 3) 10 else {
      time += 1
      println("Not yet!")
      throw new IllegalStateException("not yet")
    }
  }

  val f5 = retryBackoff(calc(),
     Seq(500.millis, 500.millis, 1.second, 1.second, 2.seconds))(scheduler, ec)
  println(Await.ready(f5, 10.seconds))

  as.terminate()
  // remember to terminate the actor system or your app will never exit
}
```

Alternatives to Futures

- The Scala Futures API is well tested, ubiquitous and performant
- If you do need more, you may want to check out a `Task` implementation like those provided by Monix or FS2
- If you need even more control, see the later section on Functors, Monads and Applicative Functors
- But at the end of the day, `Future` does 99% of what you need and is always there, and usually the underlying implementation of all these others.