4.8 Asynchronous Execution: Futures

Updated: 2021-03-08

Northeastern University

Asynchronous Execution and Futures

- What does a processor/thread do most of the time?
  - It waits to be asked to do something (unless, that is, we are very smart and sufficiently determined to keep it busy).
  - One way we can try to keep our thread busy is to ensure it doesn't have to wait for slow operations. In other words, we use asynchronous calls (i.e. non-blocking) whenever we want to do two or more things in parallel.
  - The time-honored way to implement asynchronous calls (e.g. Ajax or Swing in Java7) is by defining a *callback* method. You provide a callback that will be invoked when the result is ready.
    - That's similar to waiting for the user to do something in a UI
    - But, although ultimately you will have to do some waiting, you would prefer to minimize the number of callbacks to one (per transaction).
  - In Scala, we make asynchronous calls using a *Future*. Guess what? It's a monad.

Let's design our own type: *Par*

- <u>This is an exercise in designing an algebra</u>: similar to what we did for *List* a couple of weeks ago.
- First let's think about the kind of things we want to do in parallel:
  - How about summing the elements of a *List*?

```
def sum(is: IndexedSeq[Int]): Int =
  if (is.size <= 1)
      is.headOption getOrElse 0
  else {
      val (l,r) = is.splitAt(is.length/2)
      sum(l) + sum(r)
  }
```

For example, let's split our list of integers and sum each half independently (divide and conquer).

  - Clearly, our *Par* will have to be able to hold a result of some type *T*. That's to say it will "wrap" a *T*.

Let’s design our own Par (2)

- So, let’s create a trait and some methods:

```
trait Par[T] {
  def get: T
  def unit(t: T): Par[T]
}

object Par {
  def apply[T](t: T): Par[T] = new Par[T] { def get = t; def unit(u: T): Par[T] =
Par.apply(u)}
}
```

- Now, we can rewrite our sum method:

```
def sum(is: IndexedSeq[Int]): Int =
    if (is.size <= 1)
        is.headOption getOrElse 0
    else {
        val (l,r) = is.splitAt(is.length/2)
        val sumL = Par(sum(l))
        val sumR = Par(sum(r))
        sumL.get + sumR.get
    }
```

Notice that a *val* declaration is essentially a pattern—we can declare a tuple made up of two vals: *l* and *r*.

- Note that we haven’t said anything yet about how we might <u>implement</u> the *get* or the *apply* methods.

Let’s design our own Par (3)

- This is fine, but if we substitute the right-hand-side of *get* for the invocation(s) of *get*, we basically force evaluation—but not in parallel.
- We need a way to combine the results of the two parallel computations, leaving *get* to be called later when we really need to know the answer.
- Can you think of a method where we can combine two *containers* (like *Par*), while knowing only a function that can be applied to combine the values of the containers? Sound familiar??

```
trait Par[T] {
  def get: T
  def map2(p: Par[T])(f: (T,T)=>T): Par[T]
}
object Par {
  def apply[T](t: => T): Par[T] = ???
  def sum(is: IndexedSeq[Int]): Int =
    if (is.size <= 1)
      is.headOption getOrElse 0
    else {
      val (l,r) = is.splitAt(is.length/2)
      val sumL = Par(sum(l))
      val sumR = Par(sum(r))
      val result = sumL.map2(sumR)(_+_)
      result.get
    }
}
```

Let’s design our own Par (4)

- What have we got?
- *Par* is a data structure that allows us to set up lazy calculations which can, we hope, be implemented in parallel.
  - But *Par* doesn’t know how to do this.
- What does?
  - *ExecutorService* knows.
    - It’s a Java class with a Scala wrapper (kind of)

```
class ExecutorService {
  abstract def submit[T](arg0: Callable[T]): Future[T]
}
trait Callable[T] { def call: T }
trait Future[T] {
  def get: T
  def isDone: Boolean
  // etc.
}
```

Let’s design our own Par (5)

- So, when we actually *get* the value of our *Par* object, we will have to *run* it with an *ExecutorService*.
- So, it turns out that *get* isn’t so useful and we will replace it with *run*:

```
trait Par[T] {
  def run(implicit ec: ExecutorService): T
  def map2(p: Par[T])(f: (T,T)=>T): Par[T]
}
```

Let’s pass the *ExecutorService* implicitly since it’s not really part of the logic that we want to make obvious.

- So, now we can rewrite our *sum* method:

```
trait Par[T] {
  def run(implicit ec: ExecutionContext): Future[T]
  def map2(p: Par[T])(f: (T,T)=>T): Par[T]
}

object Par {
  def apply[T](t: => T): Par[T] = ???
  def sum(is: IndexedSeq[Int]): Int =
    if (is.size <= 1)is.headOption getOrElse 0
    else {
      import scala.concurrent.ExecutionContext.Implicits.global
      val (l,r) = is.splitAt(is.length/2)
      val sumL = Par(sum(l))
      val sumR = Par(sum(r))
      val result = sumL.map2(sumR)(_+_)
      result.run.get
    }
}
```

Notice that, instead of having *run* return a *T*, we return a *Future[T]*. In this context, *Future* is a Java interface.

Also, we are passing in an *ExecutionContext*, whence we can derive an *ExecutorService*.

Let’s design our own Par (6)

- OK, this isn’t bad.
- How should we go about implementing *map2*?

```
def map2(p: Par[T])(f: (T,T)=>T): Par[T] = for (t1 <- this; t2 <- p) yield f(t1,t2)
```

- What will we need *Par* to implement for this to work?
    - *Par* needs to be a <u>monad</u>: i.e. it must implement *map* and *flatMap*
- In addition to *map2*, we’ll need something that will take an arbitrary number of segments, our old friend *sequence*:

```
def sequence[T](tps: List[Par[T]]): Par[List[T]] = ???
```

- In practice, however, there is *ParSeq*, *ParMap*, etc.
- In reality, I haven’t found much use for these *Par*-type classes.
- And, also in reality, *Future[T]* isn’t exactly like Java’s *Future* object. It is much more flexible and powerful.
- You can learn much more about this idea from *Functional Programming in Scala*.

Futures (0)

- We’ve done enough on that algebra-design exercise. Let’s now talk about the real Scala *Future* object…

Futures (1)

- *Future[T]* is a trait (and is also a monad!—no big surprise there)
  - What does that mean in practice?

- The trait has some other very useful methods:
  - *value: Option[Try[T]]*
  - *onComplete[U](f: Try[T]=>U)(implicit ec: ExecutionContext): Unit*
  - *mapTo[S : ClassTag): Future[S]*

This is more or less the equivalent of *get*

- And *Future*'s companion object has some other good methods:
  - *firstCompletedOf[T](futures: IterableOnce[Future[T]])(implicit ExecutionContext): Future[T]*
  - *sequence*, *reduce, foldLeft, fromTry, traverse, etc*..

Futures (2)

- Let's open a connection to a web page and await the result:

```
scala> val url = new java.net.URL("http://www.htmldog.com/examples/")
url: java.net.URL = http://www.htmldog.com/examples/

scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> import scala.util._
import scala.util._
```

We know that opening the connection will take a while. This statement, however, will return <u>immediately</u>. Meanwhile, we print our welcome

```
scala> val connection = Future(url.openConnection)
connection: scala.concurrent.Future[java.net.URLConnection] =
scala.concurrent.impl.Promise$DefaultPromise@439896bf

scala> println("welcome to my web crawler")
welcome to my web crawler
```

Now, we try to get the result. Oops!

```
scala> Try(connection.getInputStream)
<console>:27: error: value getInputStream is not a member of
scala.concurrent.Future[java.net.URLConnection]
       Try(connection.getInputStream)
                      ^
```

Futures (3)

- Of course, *Future* is a container! We have to actually get the *value* from it. But that value might not exist: we might have failed to open the connection.

```
scala> connection.value
res5: Option[scala.util.Try[java.net.URLConnection]] =
Some(Success(sun.net.www.protocol.http.HttpURLConnection:http://www.htmldog.com/
examples/))
```

  - Note that the type of *value* is *Option[Try[URLConnection]]*. If action is not yet complete, we get *None*. If it's complete we get either *Success(u)* or *Failure(e)*.

Futures (4)

- Continuing…
  - We can first ask *connection.isCompleted*. Or we can await the result:

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

scala> import scala.concurrent.duration._
import scala.concurrent.duration._

scala> Await.result(connection,100 millis)
res1: java.net.URLConnection =
sun.net.www.protocol.http.HttpURLConnection:http://www.htmldog.com/examples/
```

  - But we won't normally use either method here. It's better to compose all of our *Futures* together and set up a function to act accordingly (a callback, actually).

```
scala> connection.onComplete {case Success(_) => println("OK"); case _ =>
println("failed")}
OK
```

Futures (5)

- Continuing…
  - Here, we compose a couple of futures using a for-comprehension:

```
scala> import scala.io.Source
import scala.io.Source

scala> for {
     connection <- Future(url.openConnection())
     is <- Future(connection.getInputStream)
     source = Source.fromInputStream(is)
    } yield source.mkString
res18: scala.concurrent.Future[String] =
scala.concurrent.impl.Promise$DefaultPromise@efe19b1
```

It's a bit ugly having to wrap things in Future but, in practice, we will use something like Akka Http to do this sort of thing.

Futures (6)

- Review:
  - As much as possible, compose all of your *Future* objects together (use a for-comprehension or the *sequence* method);
  - As with *get* for *Option*, *Try*, you should **never** call *value* on a *Future* (instead, use a for-comprehension or set up a callback);
  - If you do call *value*, realize that the result can be any of three possibilities:
    - *None*
    - *Some(Success(x))*
    - *Some(Failure(e))*
  - Normally, you will only actually **await** the result of a single *Future* when to do otherwise will terminate your program. If you have more than one *Future* in your program then try to compose them into just one *Future* that you can await on.

Futures (7)

- Exercise (enter this into the REPL or Scastie or ScalaFiddle*):

```
import scala.collection.immutable.IndexedSeq
import scala.concurrent._
import scala.concurrent.duration._
import scala.util._
import scala.concurrent.ExecutionContext.Implicits.global
val chunk = 10000 // Try it first with chunk = 10000 and build up to 1000000
def integers(i: Int, n: Int): LazyList[Int] = LazyList.from(i) take n
def sum[N : Numeric](is: LazyList[N]): BigInt = is.foldLeft(BigInt(0))(_+implicitly[Numeric[N]].toLong(_))
def asyncSum(is: LazyList[Int]): Future[BigInt] = Future {val x = sum(is); System.err.println(s"${is.head} is done with sum $x"); x}
val xfs = for (i <- 0 to 10) yield asyncSum(integers(i * chunk, chunk))
val xsf = Future.sequence(xfs)
val xf: Future[BigInt] = for (ls <- xsf) yield ls.sum
```

- It's your job to process the result *xf* appropriately.
- What extra statement must you provide if you are compiling/running this in a *main* program?
- See also *FutureExercise* in the REPL.

*** But, if you use ScalaFiddle you will have to revert LazyList to Stream**

Future: usage

- Futures are used in many contexts:
  - In the source code for Spark (although not in your application code because of the Spark architecture);
  - interacting with Akka (HTTP, Actors or Streams);
  - *ad hoc* asynchronous calls, such as when opening a stream at a URL;
  - in Play applications;
  - Database interactions, such as Slick;
- Example: *Majabigwaduce* (map/reduce with actors)