Scala, asynchronous parallel programming

Stefan Sebastian, 242

Contents

1 Introduction

1.1 The Scala programming language

Scala is a programming language designed to integrate smoothly both object-oriented and functional programming created with the purpose of building scalable components and component systems[1]. Scala was designed to work well with some of the most popular programming languages: Java and Javascript. The syntax is very similar to that of Java, and by extension C#, making transition easier for programmers. Furthermore, Scala interoperates well with the JVM and can use most of the features of Java, including its rich ecosystem comprised of thousands of already existing and tested libraries. It is worth noting that it was designed to address most criticisms of Java. As of 2013, Scala can also be transpiled into highly performant Javascript code and run in the browser[2] and also make use of popular frameworks such as React and Angular.

Scala is a functional language, because it treats functions as values, supports nested functions and provides a concise syntax for anonymous and curried functions[1]. It is also a pure object oriented language in the sense that every values is an object, including primitives and functions. Some other notable features include: pattern matching, static typing, compiler inference of most types of variables.

1.2 Scala parallel and concurrent programming

Scala allows the programmer to use all parallel and concurrent constructs of the host language, for example Java Threads or the utility classes from the java.util.concurrent package, and also adds some specific tools. The main additions are its actor library and extension of the Future and Promise concepts. These additions will be the focus of this paper.

2 Futures and Promises

2.1 Future

The Future is one of Scala's approaches in dealing with concurrency[3]. The main motivation for it is to provide a concise way of writting asynchronous code, mainly to avoid blocking application threads during IO operations.

A Future[T] is a container type, meaning that it will eventually contain a value of type T, or an exception[4]. It encapsulates a computation, which, upon completion, will result in the respective value. A Future can only be written once, therefore its value can not be changed after the assignment caused by the end of the computation, or exception flow.

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

One way to create a Future is through the apply method on the Future companion object. A companion object is a singleton object with the same name as a class and contains methods and values specific to all instances of the companion class[5]. The

apply method takes to parameters: a computation which returns a value of type T, and which will be executed asynchronously, and an ExecutionContext, which is an implicit parameter, meaning that it is optional if defined somewhere in scope.

An ExecutionContext is responsible for executing computations and is very similar to the Java concept of Executor[3]. It can execute operations in the same thread, a new thread or a pooled thread. The scala.concurrent package comes with a default implementation, ExecutionContext.global, which is backed by a ForkJoinPool, from the Java concurrent package. The number of threads managed by it is called the parallelism level, which is set by default to the number of available processors. If needed a Java Executor can also be adapted using the method ExecutionContext.fromExecutor(...).

Listing 1: Methods for creating Future objects

```
type CoffeeBeans = String
type GroundCoffee = String
case class Water(temperature: Int)

def heatWater(water: Water): Future[Water] = Future {
   println("started heating water")
   Thread.sleep(Random.nextInt(2000))
   println("water is hot!")
   water.copy(temperature = 85)
}

def grind(beans: CoffeeBeans): Future[GroundCoffee] = Future {
   println("start grinding")
   Thread.sleep(Random.nextInt(2000))
   println("finished grinding")
   s"ground coffee of $beans"
}
```

In figure 1 there is an example of two functions which create Future objects using the Future companion object, taken from the Neophyte guide to Scala[4]. The apply method is defined inside the braces and the execution context is assumed to be implicit. After a Future instance is created, the ExecutionContext will assign a thread to it, on which the computation will be executed at a non-determinstic time.

In order to capture the result and do something with it, a callback can be added to a Future object, as demonstrated in figure 2. The callback can handle both success and exception cases.

Listing 2: Registering callbacks

```
import scala.util.{Success, Failure}
grind("arabica beans").onComplete {
   case Success(ground) => println(s"got my $ground")
   case Failure(ex) => println("could not grind")
}
```

Using callbacks can quickly lead into what is referred to as 'callback hell'[4] which means having a lot of nested functions which are difficult to read. However, Scala provides a useful alternative, which is mapping futures.

For example, in case we want to check if after heating up the water its temperature is

ok we can map a Future[Water] to a Future[Boolean] like in figure 3. The tempOk value will be assigned a Future[Boolean]. The mapping function gets executed as soon as the Future[Water] has completed successfully, however it is executed synchronously.

Listing 3: Mapping future

```
val tempOk: Future[Boolean] = heatWater(Water(25)).map { water =>
   (80 to 85).contains(water.temperature)
}
```

In the case in which we want the mapped function to also be computed asynchronously we need to keep in mind that the result will be a Future[Future[T]], which is not easy to work with. For this reason, Scala provides the flatMap utility which composes futures and returns a Future[T].

Listing 4: FlatMapping future

```
def tempOk(water: Water): Future[Boolean] = Future {
   (80 to 85).contains(water.temperature)
}
val flatFuture: Future[Boolean] = heatWater(Water(25)).flatMap {
   water => temperatureOkay(water)
}
```

Scala also provides a shorter way to compose multiple futures: the for comprehension [4]. It should be pointed out that the futures are started before the for block so they are executed concurrently, while the brew method will wait until ground and water values become available.

Listing 5: For comprehensions

```
def prepareCoffee(): Future[Coffee] = {
   val groundBeans = grind("arabica beans")
   val heatedWater = heatWater(Water(20))
   for {
      ground <- groundBeans
      water <- heatedWater
      coffee <- brew(ground, water)
   } yield coffee
}
```

2.2 Promise

Promise is a companion type for Future that allows you to complete it by setting a value[4]. This operation can only be done once. A Promise is always linked with exactly one Future instance. Even the apply method from the Future companion object creates a Promise, but shields the user from doing so explicitly.

Listing 6: Promise example

```
val promise: Promise[String] = Promise[String]()
val future = promise.future
...
```

```
val anotherFuture = Future {
   ...
   promise.success("Done")
   doSomethingElse()
}
...
future.onSuccess { case msg => startTheNextStep() }
```

The basic operations in working with Promises is illustrated in figure 6, taken from the Scala in Action book[6]. There are two Futures being created, one using the Promise object explicitly and one using the apply method. Inside the second Future the success method is invoked which will cause the onSuccess callback registered on the first Future to be executed.

2.3 Comparison with other languages

The Promise and Future approach to working with asynchronous code is available in most of the other popular programming languages, however the Scala version offers some improvements.

First of all, the Java Future class provides a very limited interface[7]. For example, it can not register a callback which results in messy code that has to do periodic polling for the result. An alternative from the java.util.concurrent package is the FutureTask, which allows the programmer to register callbacks. However, it does not provide a way to register a callback for exceptions, meaning that aspect is up to the programmer. Some ways to do that are to subclass a FutureTask and override the done() method or to surround the get() method from the Future object with a try-catch block, both approaches being much more verbose than the Scala solution.

A language which makes heavy use of this concept is Javascript, especially Node.js, which being single threaded needs to perform operations in a non-blocking way[8]. The Future objects from JS are very similar to those in Scala. An advantage of the Scala Future approach is that it provides a concise way of chaining multiple Futures and setting their dependencies using for-comprehensions, like in figure 5.

2.4 In practice

The main use of the future-based programming paradigm is to create scalable web applications[4]. The issue with traditional web servers are times spent waiting for IO operations like database queries or while interacting with other services, which causes the execution thread to block waiting for a response. Using a modern Scala web framework you can return the responses as Future[Response] instead of blocking waiting for the results. This means that all layers of the application must be asynchronous and the result should be a composition of all these Futures. This will free up the server to handle more requests and increase the scalability of the application.

Listing 7: Wrapping blocking operations with Futures to increase scalability

```
Future {
   queryDB(query)
}
```

The cases when a Future should be used over the actor model are when performing an operation concurrently and the extra utility of the actor is not necessary, or when there is a need to compose multiple concurrent operations, which is difficult to do with actors[6].

3 The Actor model

3.1 Model overview

The actor model is a computational model designed to deal with concurrency at a high level of abstraction[9]. It represents a specific implementation of the message passing concurrency model, which encourages shared nothing architectures. The building block of the model is the Actor, which is the primitive unit of computation that receives a message and handles it. This is similar to how objects work in the object-oriented paradigm, however actors never share memory, which eliminates a lot of the issues of concurrent programming.

Actors always process messages sequentially, which can be sent both synchronously or asynchronously, with the latter being more common[6]. While the actor processes a message, all the other incoming ones are stored in a queue, one for each actor, called "the mailbox". The messages themselves are immutable and are the only way to cause actors to change their internal state.

3.2 The Actor model in Scala

Scala provides an implementation for the message passing concurrency model through its actor library[6]. After Scala 2.10 the default actor library is Akka, which is also available for Java. There are other libraries but this is by far the most popular.

Actors have two main communication abstractions: send an receive. The syntax for sending a message is very simple: actor ! message. The 'message' is then added to the 'actor' mailbox which is a first-in first-out queue. In order to receive a message each actor has to define a function called 'receive' which is usually a set of patterns matching message types to specific actions.

Listing 8: Simple actor implementation

```
import akka.actor.Actor
case class Name(name: String)
class GreetingsActor extends Actor {
   def receive = {
      case Name(n) => println("Hello " + n)
   }
}
```

You can reply to messages using the sender method of an ActorRef[10] like this: sender() ! x. There is a pattern of mixing future and actors if we need to call the reply method when some asynchronous task completes and we don't want to block the executing thread.

3.2.1 Actor Systems

The infrastructure which is required for running actors is created by an actor system[6]. An actor system is a manager of one or more actors which share a common configuration. An actor can be instantiated through the system using the method actorOf which takes a configuration object and, optionally, the actor's name. The actor system can be destroyed using the 'shutdown' method, which also stops all actors, even those who still have unprocessed messages.

Listing 9: Actor system demo

```
import akka.actor.Actor
import akka.actor.ActorSystem
val system = ActorSystem("greetings")
val a = system.actorOf(Props[GreetingsActor], name="greetings-actor")
```

Actor systems are heavyweight structures, so ideally there should be one per application module: one for the database layer, one for web service calls, etc[10]. An actor system maintains a hierarchy of all actors associated with it. At the top, there is the guardian actor, which is created automatically. Each actor in a system has a parent, which is responsible for error handling, and can have multiple children.

3.2.2 Actor references

When you create actors in Akka you get back a reference called ActorRef to the actual object[6]. This means that you can't access the actor directly and mutate its state. The ActorRef is also serializable, so if an actor crashes in one node of a distributed system, you can send the reference to another node and start it there. The actor system also mainatins an actor path, which is an unique identifier for each actor, given the fact that actors are maintained in a hierarchical structure. A useful thing about the actor path is that it can also identify actors on other machines.

3.2.3 Message dispatchers

The actual job of sending and receiving messages in an actor system is done by the MessageDispatcher[6] component. Every actor system comes with a default one, however this can be overriden through configuration files and can be specialized for different actors. Every MessageDispatcher uses a thread pool internally. When receiving a message, the dispatcher adds it to the actor mailbox and immediately returns control to the sender. Sending and handling messages happens in different threads. When an actor receives a message the dispatcher checks if there is a free thread in the pool, and in case there isn't waits for one to be available. The selected thread reads messages from the mailbox and executes them by passing the receive method sequentially. A guarantee offered by the dispatcher is that a single thread always executes a given actor, which means that you can safely use mutable state inside an actor.

Listing 10: Actor API

```
def unhandled(message: Any): Unit // for unmatched messages
val self: ActorRef // reference to self
final def sender: ActorRef // reference to sender of last received
    message
```

```
val context: ActorContext // contextual information for the actor
def supervisorStrategy: SupervisorStrategy // fault handling strategy
def preStart() // when started for the first time
def preRestart() // when restarted
def postStop() // after being stopped
def postRestart() // after restart
```

3.2.4 Supervision hierarchy

Each actor is the supervisor of its children and has associated with it a SupervisorStrategy[10]. Akka encourages non-defensive programming, which is a paradigm that treats errors as a valid state of the application and handles it through fault tolerance methods, such as the supervisor strategies. Akka provides two supervisor strategies: One-for-One and All-for-One. The first one means restarting only the actor that dies, which is useful if actors are independent in the system. However, in some cases if the workflow is dependent on multiple actors interacting then restarting only one might not work. For this reason the second strategy was introduced, which restarts all the actors supervised by the supervisor.

The responsability of a supervisor is to start, stop and monitor the child actors[6]. Each actor is by default a supervisor for its child actors. You can also have the supervisor hierarchy spread across multiple machines, which greatly increases the fault tolerance of the system.

3.3 Advantages and disadvantages of the Actor Model

The actor model was designed as a high level approach for concurrent programming, and it solves most of the problems associated with the classic Java Thread based approach[10]. One of the areas which causes most bugs and performance hits in the thread model is shared mutable state. This means that multiple threads can access and modify the same variables which can lead to inconsistent state and requires synchronization mechanisms such as locks. Actors encapsulate state and the only way to modify the internal state of an actor is through the messages it receive. Furthermore, actors process messages in a synchronous fashion, meaning that the same data is never accessed concurrently. This approach already solves a lot of potential concurrency related bugs.

Another problems with synchronization is the necessity for locks. Locks are very costly and limit concurrency. Also, the caller thread is blocked, meaning it won't be able to do any meaningful work. This could be compensated by launching more threads, but threads are a costly abstraction and add significant overhead. Moreover, programming with locks requires a lot of careful consideration from the part of the programmer, because it can introduce a whole new class of bugs: deadlocks.

Most of the shared memory models were designed back when writting to a variable meant writting directly to a memory location. However, modern architectures have multiple CPU's and writes are done to cache lines. This means that in order to make changes visible between different cores they need to be shipped explicitly to the other core's cache, which causes a drop in performance.

Another issue with the above model is error handling[10]. Call stacks are local to a thread so when a thread has to delegate work to another and the callee fails we can't rely on the call stack. Actors are designed to handle errors more gracefully. Firstly if the

delegated task fails because of an error in the task than the erorr is part of the problem domain and the response presents the error case. Secondly, if the service itself encounters an internal fault, its parent actor is notified and has the responsability to restart the children, just like processes are organised in operating systems. The main idea is that children never die silently.

However, the actor model is not always the best tool for the task. A classic example of where this wouldn't be a good fit is for shared state[6]. If you want to transfer money from one account to another you can't do that only through actors, you need transactional support. Another issue is that switching to asynchronous programming introduces a time cost for the developers. It is a huge paradigm shift and message passing architectures make for hard to debug programs. Furthermore, there are some cases when you need maximum performance. In such situations using actors adds overhead and a more suitable solution is going for something low level like threads.

3.4 Comparison Erlang Actor Model implementations

One of the most popular actor model implementation is the one provided by Erlang, which is a language designed to be parallel and distributed with roots in the telecom industry, where concurrent processes are the norm.

Erlang has better fault tolerance than scala because its VM forces full isolation between actors, even having separate heaps and garbage collection[11]. This is something that can't be done in Scala since it is built on the JVM.

Erlang does the process scheduling for you, using a preemptive scheduler[11]. This means that when a process has been executed for too much time it is paused and another actor is started. However, you can not change the default scheduler, something which is highly customizable in Scala (using Akka).

The platform on which Erlang runs is OTP(Open Telecom Platform) which allows for hot swapping of code[11]. This means that a real time system can be upgraded without having downtime. In Scala this is not possible due to JVM limitations. Because of the static type system method signature can not be changed, only their content. However, an advantage of running on the JVM is that Akka can be integrated with the entire Java ecosystem, which has a tool for almost every task.

3.5 An example of an Actor based application

An example of a more complex application written using the actor model can be seen in figure11, which was taken from the Scala in Action book[6]. The problem statement is to count the number of words in each file of a directory and then sort them in ascending order. The approach taken is divide et impera, with a master actor that creates a number of slaves charged with reading a file and counting the words. The master then combines and sorts the results.

The worker actor is conceptually simpler. The receive message can handle just one type of message which contains a file that the actor should read and process. After counting the words the worker sends a message to sender containing a touple of file and number of words.

The master actor can handle two types of messages. First of all, the StartCounting message is sent at the beggining of the workflow, to initialize the process. Depending on the given parameters, the master will create child actors and read files from the required

location. Because there may be more files than actors, the FileToCount message is sent to each actor in a round robin fashion. The reply from each worker is a WordCount message, which is used to update the sortedCount map, which maintains the file names alongside their word count. When all files have been counted the actor system is terminated and the application stops.

Listing 11: Actor wordcount

```
import akka.actor.Actor
import akka.actor.Props
import akka.actor.ActorRef
import akka.actor.ActorSystem
import java.io._
import scala.io._

case class FileToCount(fileName:String)
case class WordCount(fileName:String, count: Int)
case class StartCounting(docRoot: String, numActors: Int)

class WordCountWorker extends Actor {
   def countWords(fileName:String) = {
      val dataFile = new File(fileName)
      Source.fromFile(dataFile).getLines.foldRight(0)(_.split(" ").size + _)
   }
   def receive = {
      case FileToCount(fileName:String) =>
         val count = countWords(fileName)
         sender ! WordCount(fileName, count)
   }
   override def postStop(): Unit = {
      println(s"Worker actor is stopped: ${self}")
   }
}

class WordCountMaster extends Actor {
   var fileNames: Seq[String] = Nil
   var sortedCount : Seq[(String, Int)] = Nil

   def receive = {
      case StartCounting(docRoot, numActors) =>
         val workers = createWorkers(numActors)
         fileNames = scanFiles(docRoot)
         beginSorting(fileNames, workers)
      case WordCount(fileName, count) =>
         sortedCount = sortedCount :+ (fileName, count)
         sortedCount = sortedCount.sortWith(_._2 < _._2)
         if(sortedCount.size == fileNames.size) {
            println("final result " + sortedCount)
            finishSorting()
         }
   }
   override def postStop(): Unit = {
```

```
        println(s"Master actor is stopped: ${self}")
    }

    private def createWorkers(numActors: Int) = {
        for (i <- 0 until numActors) yield
            context.actorOf(Props[WordCountWorker], name = s"worker-${i}")
    }

    private def scanFiles(docRoot: String) =
        new File(docRoot).list.map(docRoot + _)

    private[this] def beginSorting(fileNames: Seq[String], workers:
        Seq[ActorRef]) {
        fileNames.zipWithIndex.foreach( e => {
            workers(e._2 % workers.size) ! FileToCount(e._1)
        })
    }

    private[this] def finishSorting() {
        context.system.terminate()
    }
}

object Main {
    def main(args: Array[String]) {
        val system = ActorSystem("word-count-system")
        val m = system.actorOf(Props[WordCountMaster], name="master")
        m ! StartCounting("src/main/resources/", 2)
    }
}
```

4 Conclusions

Scala is a feature rich and flexibile language. Through its JVM implementation it can access a very rich ecosystem and all parallel and asynchronous libraries available for Java. On top of that, it comes with higher level constructs such as the Actor Model. The actor model implementation offers many benefits over lower level thread based approaches in the fact that it works as shared-nothing, it handles error gracefully and is very easy to scale to multiple machines. Another approach for asynchronous code is the Future Promise model, for which Scala provides a lot of syntactic sugar over other languages, making it easy to work with.

References

[1] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," 12 2008.

[2] S. Doeraene, "Scala.js: Type-directed interoperability with dynamically typed languages," p. 10, 2013.

[3] "Futures and promises." `https://docs.scala-lang.org/overviews/core/futures.html`. Accessed: 2019-05-08.

[4] "The neophyte's guide to scala." `https://danielwestheide.com/blog/2013/01/16/the-neophytes-guide-to-scala-part-9-promises-and-futures-in-practice.html`. Accessed: 2019-05-08.

[5] "Singleton objects." `https://docs.scala-lang.org/tour/singleton-objects.html`. Accessed: 2019-05-08.

[6] N. Raychaudhuri, *Scala in Action*. Greenwich, CT, USA: Manning Publications Co., 2013.

[7] "Guide to java.util.concurrent.future." `https://www.baeldung.com/java-future`. Accessed: 2019-05-09.

[8] "Javascript promises: an introduction." `https://developers.google.com/web/fundamentals/primers/promises#chaining`. Accessed: 2019-05-09.

[9] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: Updated for Scala 2.12*. USA: Artima Incorporation, 3rd ed., 2016.

[10] "Akka documentation." `https://doc.akka.io/docs/akka/2.5.5/scala/`. Accessed: 2019-05-09.

[11] "Erlang vs scala, a history of tradeoffs." `https://medium.com/@emqtt/erlang-vs-scala-5b5190326ef5`. Accessed: 2019-05-12.