62 Actors

Why Actors?

- We know that side-effects* (including I/O) and mutable state don't support referential transparency, i.e. we cannot substitute expressions for identifiers when side-effects/mutable state are involved. We cannot compose functions (e.g. *for-comprehensions*) when the functions are not R.T.
    - But just about all complex systems involve some side-effects and mutable state. So, how can we make these rogue components safe for Functional Programming?
    - We can create concepts such as I/O Monads and state-preserving concepts such as our RNG class. By doing this, we can isolate all of the side-effectual code into very clear chunks, leaving all the rest of the logic to be *composable* in our FP way.
    - What about mutable state? Another technique is to isolate mutable state in a way that other parts of the system can never observe a mutation. For instance, we want to implement *quicksort* by creating an *Array* of the elements and mutating their positions. Then we copy the array back to the original form. As long as the rest of the system can never substitute that array, we should be safe.

* **side-effect** is basically any interaction with the world outside our program and, in particular, the thread it inhabits.

Why Actors? (2)

- And, there is one other practical consideration. When we are dealing with side-effects, those actions typically take enormous amounts of time compared with "normal" programming. We typically want to interact with the world outside using a *Future*.
- There is one further golden rule of functional programming: let each method (or function) do one thing and do it well. [Actually, this should be the rule for all programming, period.] Remember our mantra:

simple, obvious, elegant.

- *Actors* embody all the properties so far mentioned (although they are admittedly not the only way to do so). But actors can do a lot more…
- Scala has its own actors but they are now deprecated in favor of…

Akka

- *Akka* actors are:
  - **message receivers and senders:**
    - that is they can communicate with the rest of the system only via messages—this helps reduce *coupling*.
  - **encapsulating:**
    - in the sense we talked about—the only handle that we, as programmers, have to an actor (except when testing) is an *ActorRef* which does not give access to any actor internals such as mutable state. Therefore the rest of the system cannot observe any side-effects or mutability directly and can therefore follow normal FP composition rules and patterns.
  - **thread-safe:**
    - in the sense that the entire processing of any one message is completed before any other message can be processed.
  - **untyped:**
    - that's to say an actor has no intrinsic knowledge about the types of its properties—all actors are essentially the same in this regard. Information about typed objects is confined to the messages sent and received.
  - **replicable:**
    - that's to say that the actor system can make copies of an actor to improve performance. These copies can be on remote systems of course.
  - **resilient:**
    - that's to say that the actor system monitors the health of actors and will restart an actor if required, potentially bubbling up to the surface any exception.
  - **lightweight:**
    - the boiler plate overhead for an actor is only about 1k bytes so it's practical to have millions of them in an application.

Akka (2)

- Overall, these properties make *Akka* the perfect system for *reactive programming*!
  - So, how do we get started? First go to http://akka.io for documentation, tutorials, patterns, etc.
  - Then add the following to your *build.sbt*

```
val akkaGroup = "com.typesafe.akka"
val akkaVersion = “2.4.1”
libraryDependencies ++= Seq(
    akkaGroup %% "akka-actor" % akkaVersion,
    akkaGroup %% "akka-testkit" % akkaVersion % "test",
    akkaGroup %% "akka-slf4j" % akkaVersion,
    "com.typesafe" % "config" % "1.3.0",
    "ch.qos.logback" % "logback-classic" % "1.0.0" % "runtime"
)
```

  - Let’s take a look at an existing system: using Akka to solve Map-Reduce problems (Majabigwaduce)

Akka (3)

- An actor receives *typed* messages via its *receive* method*:

```
/**
 * The purpose of this mapper is to convert a sequence of objects into several sequences, each of
which is
 * associated with a key…
 * …
 */

class Mapper[K1,V1,K2,W](f: (K1,V1)=>(K2,W)) extends MapReduceActor {

  override def receive = {
    case  i: Incoming[K1,V1] =>
      log.info(s"received $i")
      val wk2ts = for ((k1,v1) <- i.m) yield Try(f(k1,v1))
      sender ! prepareReply(wk2ts)
    case q =>
      super.receive(q)
  }
}

case class Incoming[K, V](m: Seq[(K,V)]) {
  override def toString = s"Incoming: with ${m.size} elements"
}
```

  - Typically, the actor will prepare a response and send it either to the sender or another actor. A logging actor is able to log the messages as they arrive.

* This is taken from Majabigwaduce

Akka (4)

```
import akka.actor.{ Actor, ActorLogging, ActorSystem, Props }
import scala.concurrent.duration._
import scala.concurrent._
import akka.util.Timeout
import akka.pattern.ask
import scala.util._
import ExecutionContext.Implicits.global

case class QuickSort() extends Actor with ActorLogging {
  override def receive = {
        case input: Seq[Int] =>
          log.info(s"received $input")
          val array = input.toArray
          Sorting.quickSort(array)
          val response = array.toSeq
          sender ! response
        case _ => println("unknown message")
  }
}

object QuickSort extends App {
 implicit val timeout: Timeout = 10.seconds
 implicit val system = ActorSystem("QuickSort")
 val quickSort = system.actorOf(Props.create(classOf[QuickSort]), "sorter")
 val ints = args map {_.toInt}
 val f = quickSort ? ints.toSeq
 f.onComplete { x => x match {
   case Success(s) => println(s"received response of sorted sequence: $s")
   case Failure(e) => System.err.println(s"exception: $e")
 }
 }
 Await.ready(f, 10.second)
 system.stop(quickSort)
 system.~~shutdown~~
}
```

Quicksort is most efficient for large *N* when it is operating on a (mutable) array.
Encapsulating array inside an *Actor* is *Referentially Transparent.*
Quicksort is one of the best-known and most efficient sorting methods: it is generally **O**(*N log N*) but in worst case is **O** (N^2).

Map-Reduce

- Map-reduce is a pattern for parallel processing:
  - it is based on the notion that you can divide a problem into *N* tasks:
    - provided that each of the tasks is truly independent;
    - where you have a “map” function which yields a “key” for each element (sub-division) of the problem;
    - and where you have a “reduce” function which evaluates all elements with key “k” in parallel with (and independent of) all other elements;
    - once all of the *N* values are available, they are collected together and combined for the final report/result;
  - it is the pattern on which *Hadoop* and *Spark* are based;
  - it is very amenable to functional programming.

Map-Reduce (2)

- The nth stage of the process looks like this:
  - $Map[K_{n-1}, V_{n-1}] \rightarrow Map[K_n, Seq[W_n]] \rightarrow Map[K_n, V_n]$
    "mapper" "reducer(s)"
  - But, typically, the input data to the first stage has no natural key so we use the fact that $Map[K_0, V_0]$ can be transformed directly to $Seq[(K_0, V_0)]$ where $Seq[(\varnothing, V_0)]$ in turn is the equivalent of $Seq[V_0]$.

But we can't *generally* go in the reverse direction. Why not?

- Assuming, then, that each (nth) stage of the pipeline works as expected, then overall, we can transform:
  - $Seq[V_0] \rightarrow Map[K_n, V_n]$

Map-Reduce (3)

- For example…
  - $Map[K_{n-1},V_{n-1}] \rightarrow Map[K_n,Seq[W_n]] \rightarrow Map[K_n,V_n]$

    "mapper" "reducer(s)"
  - But, typically, the input data to the first stage has no natural key so we use the fact that $Map[K_0,V_0]$ can be transformed directly to $Seq[(K_0,V_0)]$ where $Seq[(\varnothing,V_0)]$ in turn is the equivalent of $Seq[V_0]$.

But we can't *generally* go in the reverse direction. Why not?

- Assuming, then, that each (nth) stage of the pipeline works as expected, then overall, we can transform:
  - $Seq[V_0] \rightarrow Map[K_n,V_n]$

CountWords app

```
/**
 * CountWords: an example application of the MapReduce framework.
 * This application is a three-stage map-reduce process (the final stage is a pure reduce process).
 * Stage 1 takes a list of Strings representing URIs, converts to URIs, opens each as a stream, reading the contents
and finally returns a map of URI->Seq[String]
 * where the key is the URI of a server, and the Strings are the contents of each of the documents retrieved from that
server.
 * Stage 2 takes the map of URI->Seq[String] resulting from stage 1 and adds the lengths of the documents (in words) to
each other. The final result is a map of
 * URI->Int where the value is the total number of words read from the server represented by the key.
 * Stage 3 then sums these values together to yield a grand total.
*/
case class CountWords(resourceFunc: String => Resource)(implicit system: ActorSystem, config: Config, timeout: Timeout,
ec: ExecutionContext) extends (Seq[String] => Future[Int]) {
  override def apply(v1: Seq[String]): Future[Int] = {
    def init = Seq[String]()

    val stage1: MapReduce[String, URI, Seq[String]] = MapReduceFirstFold(
      { w: String => val u = resourceFunc(w); system.log.debug(s"stage1 map: $w"); (u.getServer, u.getContent) }, { (a:
Seq[String], v: String) => a :+ v },
      init _
    )
    val stage2: MapReduce[(URI, Seq[String]), URI, Int] = MapReducePipe(
      { (w: URI, gs: Seq[String]) => (w, (for (g <- gs) yield g.split("""\s+""").length) reduce (_ + _)) }, { (x: Int,
y: Int) => x + y },
      1
    )
    val stage3 = Reduce[Int, Int](_ + _)
    val countWords = stage1 | stage2 | stage3
    countWords.apply(v1)
  }
}
```

CountWords app

```scala
object CountWords {
  def apply(hc: HttpClient, args: Array[String]): Future[Int] = {
    val configRoot = ConfigFactory.load
    implicit val config: Config = configRoot.getConfig("CountWords")
    implicit val system: ActorSystem = ActorSystem(config.getString("name"))
    implicit val timeout: Timeout = getTimeout(config.getString("timeout"))
    import ExecutionContext.Implicits.global

    val ws = if (args.length > 0) args.toSeq else Seq("http://www.bbc.com/doc1", "http://www.cnn.com/
doc2", "http://default/doc3", "http://www.bbc.com/doc2", "http://www.bbc.com/doc3")
    CountWords(hc.getResource).apply(ws)
  }

  // TODO try to combine this with the same method in MapReduceActor
  def getTimeout(t: String): Timeout = {
    val durationR = """(\d+)\s*(\w+)""".r
    t match {
      case durationR(n, s) => new Timeout(FiniteDuration(n.toLong, s))
      case _ => Timeout(10 seconds)
    }
  }
}
```

CountWords unit test

```
class CountWordsSpec extends FlatSpec with Matchers with Futures with ScalaFutures with Inside with
MockFactory {
  "CountWords" should "work for http://www.bbc.com/ http://www.cnn.com/ http://default/" in {
    val wBBC = "http://www.bbc.com/"
    val wCNN = "http://www.cnn.com/"
    val wDef = "http://default/"
    val uBBC = new URI(wBBC)
    val uCNN = new URI(wCNN)
    val uDef = new URI(wDef)
    val hc = mock[HttpClient]
    val rBBC = mock[Resource]
    (rBBC.getServer _).expects().returning(uBBC)
    rBBC.getContent _ expects() returning CountWordsSpec.bbcText
    val rCNN = mock[Resource]
    rCNN.getServer _ expects() returning uCNN
    rCNN.getContent _ expects() returning CountWordsSpec.cnnText
    val rDef = mock[Resource]
    rDef.getServer _ expects() returning uDef
    rDef.getContent _ expects() returning CountWordsSpec.defaultText
    hc.getResource _ expects wBBC returning rBBC
    hc.getResource _ expects wCNN returning rCNN
    hc.getResource _ expects wDef returning rDef
    val nf = CountWords(hc, Array(wBBC, wCNN, wDef))
    whenReady(nf, timeout(Span(6, Seconds))) {
      case i => assert(i == 556)
    }
  }
}
```