

Scala Actors, Akka



31/10/2011 – October CZJUG, David Polách , Karel Smutný

Let's start with an illustration

- ☞ 7 volunteers, each knowing his experience with Java in whole years

```
class JavaProgrammer (  
    val yearsOfExperience: Int  
)
```

- ☞ let's sum up total Java experience

```
val volunteers = List(jarda, pepa, ...)   
val totalExperience = volunteers map  
    (_.yearsOfExperience) sum
```

Let's start with an illustration

Exercise 1

- Shared memory cell on a blackboard

```
var totalExperience = 0
```

- Update

- Acquire a sponge and a chalk resources
- Erase old value with the sponge
- Write down new value with the chalk
- Release the resources

- Each Java programmer updates the cell

```
totalExperience +=  
programmer.yearsOfExperience
```

Let's start with an illustration

Exercise 2

- ⌘ Programmers stand in a row
- ⌘ Whenever a programmer gets a card with a number
 - `val subtotal = receivedYears + myYears`
 - `nextProgrammer ! subtotal`
- ⌘ The last programmer writes the result on a blackboard
- ⌘ Cards might be prepared in advance

What have we learned?

Two approaches to concurrency:

- ∞ Shared mutable state (Exercise 1)

Multiple threads share a state, compete for access to it

- ∞ Message passing and isolated mutable state

Only one thread has access to a state, threads exchange data via immutable messages

Or even better: Pure Immutability (Exercise 2)

What will we learn about?

Two approaches to concurrency:

∞ Shared mutable state (Exercise 1)

Multiple threads share a state, compete for access to it

∞ **Message passing and isolated mutable state**

Only one thread has access to a state, threads exchange data via immutable messages

Or: Pure Immutability (Exercise 2)

What makes you an Actor?

You are able to receive messages. That's it!

- ∞ Actor processes one message at a time
- ∞ Message ordering is not determined
- ∞ Messages are asynchronous. “Send it and forget it.”

Actor model first proposed by Carl Hewitt at MIT (1970)

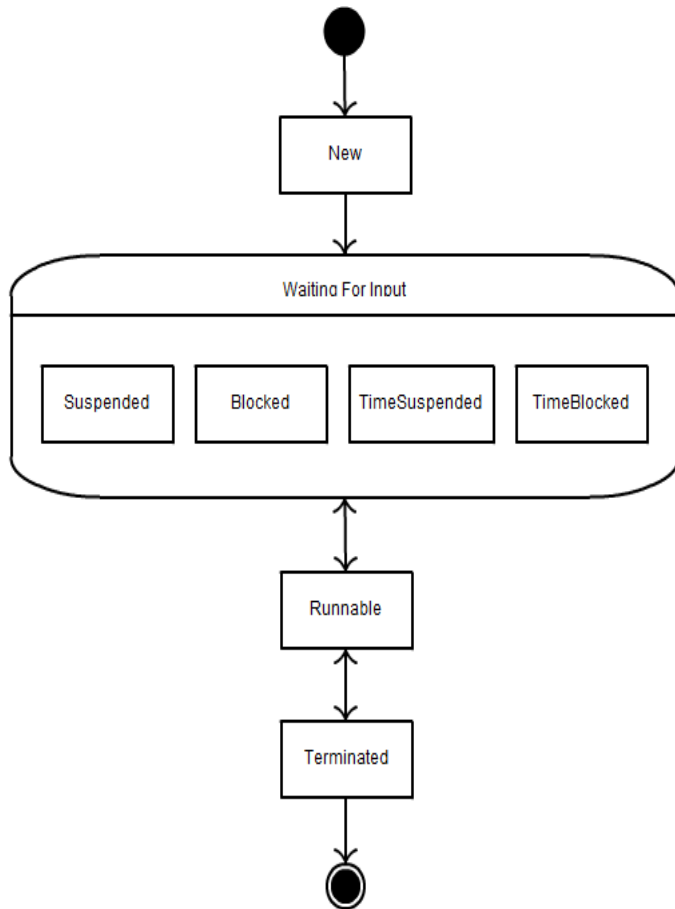
“...we use the ACTOR metaphor to emphasize the inseparability of control and data flow in our model ...”

What makes you a Scala Actor?

Implement `Actor` trait and its **def** `act(): Unit` method

- ☞ Incoming messages are served one at a time
- ☞ Message processing runs in a single thread
 - Not necessarily dedicated
- ☞ Can receive and process messages in several ways
 - Asynchronous: `actor ! message`
 - With Future: `val future = actor !! message`
 - Synchronous: `val response = actor !? message`

Actor Lifecycle



Unlike the lives of movie actors, the life of an actor object is rather boring:

- ∞ Once an actor is created, it typically starts processing incoming messages.
- ∞ Once an actor has exceeded its useful life, it can be stopped and destroyed (either of its own accord, or as a result of some “poison pill” message).

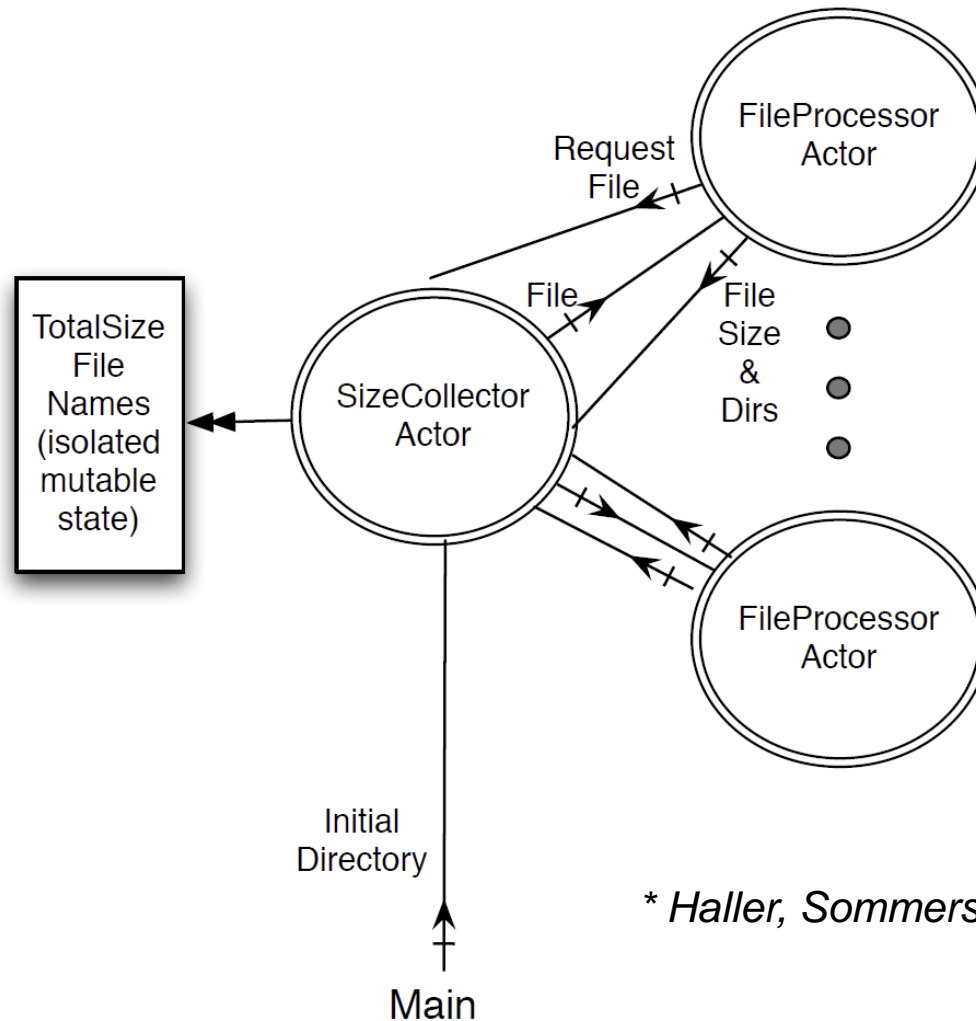
* *Haller, Sommers: Actors In Scala*

Example: Scala Programmer

```
case class Experience(years: Int, others: List[Actor])

class JavaProgrammer(val experienceYears: Int) extends Actor {
  def act() {
    react {
      case Experience(subtotalYears, others) =>
        val total = subtotalYears + experienceYears
        others match {
          case next :: remainder => next ! Experience(total, remainder)
          case Nil => println(total)
        }
    }
  }
}
```

Example: Calculating Folder Size



* *Haller, Sommers: Actors In Scala*

Example: Message Classes

```
case object RequestAFile
```

```
case class FileSize(size : Long)
```

```
case class FileToProcess(fileName : String)
```

Example: File Processor

```
class FileProcessor(val sizeCollector: Actor) extends Actor {  
  def registerToFile = sizeCollector ! RequestAFile  
  def act() {  
    registerToFile  
    loop {  
      react {  
        case FileToProcess(fileName) =>  
          val size = sizeOf(new java.io.File(fileName), dirSize)  
          sizeCollector ! FileSize(size)  
          registerToFile  
        }  
      }  
    }  
  }  
  
  def sizeOf(file: File, processDir: File => Long) =  
    if (file.isFile) file.length else processDir(file)  
  def dirSize(file: File) = file.listFiles map sizeOf(_, subdirSize) sum  
  def subdirSize(file: File) =  
    { sizeCollector ! FileToProcess(child.getPath); 0 }  
}
```

Example: SizeCollector

```
class SizeCollector extends Actor {  
  ...  
  react {  
    case RequestAFile =>  
      fileProcessors = sender :: fileProcessors  
      sendAFileToProcess()  
    case FileToProcess(fileName) =>  
      filesToProcess = fileName :: filesToProcess  
      pendingFilesToVisit += 1  
      sendAFileToProcess()  
    case FileSize(size) =>  
      totalSize += size  
      pendingFilesToVisit -= 1  
      if (pendingFilesToVisit == 0) endCalculation()  
  }  
}
```

Example: Putting it all together

```
object ConcurrentFileSize {  
  def main(args : Array[String]) {  
    val sizeCollector = new SizeCollector().start()  
    sizeCollector ! FileToProcess(args(0))  
    for (i<- 1 to 100)  
      new FileProcessor(sizeCollector).start()  
  }  
}
```

Thread vs. Event model

☞ Thread model

- Each actor has its own dedicated thread
- Suitable for small number of Actors

☞ Event model

- Actors run in threads allocated from a thread pool
- Lightweight approach with excellent **scalability**

☞ Not fully transparent to a programmer

☞ Both approaches follow different paradigm

Thread Model

- ⌘ Thread per Actor model
- ⌘ Each JVM thread may be mapped to an underlying operating system process
- ⌘ A thread-based actor waits for message by invoking wait on an object for which its thread holds the associated lock (wait/ notify)

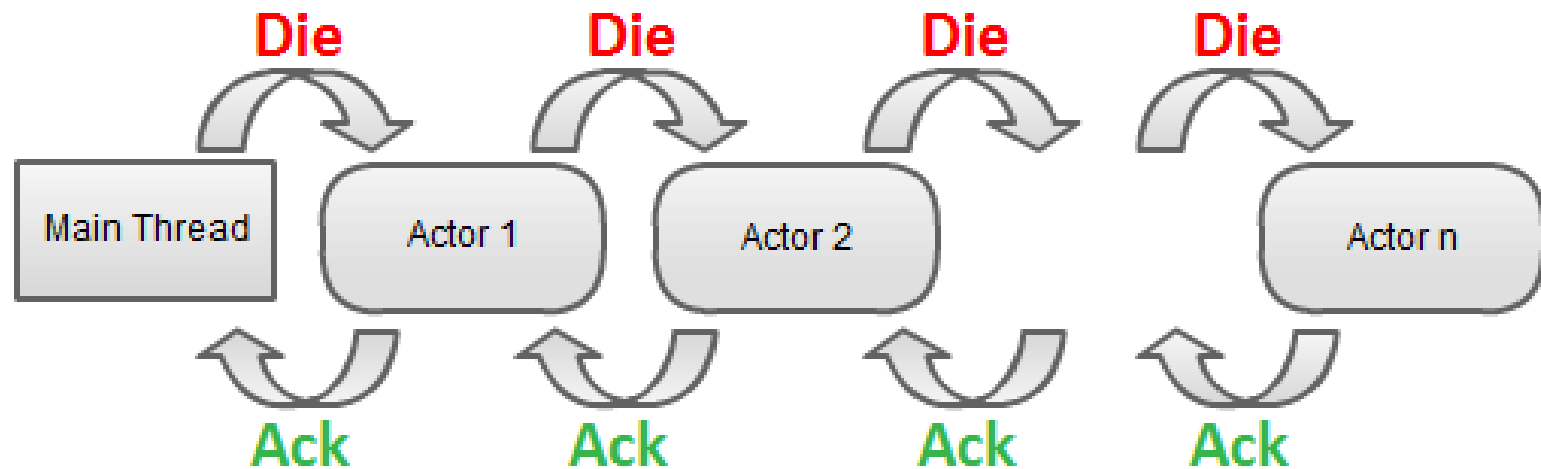
```
class ThreadActor extends Actor {  
  def act() {  
    while (true) {  
      receive {  
        case message => ...  
      }  
    }  
  }  
}
```

Event Model

- ∞ Actors are decoupled from JVM threads and just register a handler with the Actor runtime
- ∞ When a message is received, the runtime schedules event-handler for execution on a thread
- ∞ `react` handler never returns (no full stack frame)

```
class EventActor extends Actor {  
  def act() {  
    loop {  
      react {  
        case message => // serve  
      }  
      // code here is never executed  
    }  
  }  
}
```

Actor Scalability Example



Actor Scalability Example

Number Of Actors	Thread Model	Event Model
1 000	3,9 s	340 ms
10 000	245 s	880 ms
100 000	java.lang.OutOfMemory	3 s
1 000 000	guess what?	23 s

- Why would anyone even consider thread model?
 - `ThreadLocal` variables (might be used by 3rd party libraries your code depends on)
 - Java synchronization primitives
 - Actor/process never losses its stack frame

Remote Actors

- ⌘ Cooperating Actors does not have run and exchange messages within a single JVM
- ⌘ Same abstraction can be used to build a network of Actors on different JVMs / network nodes
- ⌘ Protocol based on Java serialization (messages)
- ⌘ Simple remote invocation API (`RemoteActor`)

Any limitations?

- ⌘ We must ensure messages immutability on your own
- ⌘ Actors do not prevent deadlock
- ⌘ Potential for starvation (waiting for a lost message)
- ⌘ Not all applications are well-suited

Akka

- ~~ಞ Association of Kannada Kootas of America~~
- ಞ Akka is the platform for the next generation **event-driven, scalable** and **fault-tolerant** architectures on the JVM (www.akka.io)
- ಞ Transparent remoting
- ಞ Hierarchical supervision
- ಞ STM, Transactors
- ಞ Scala & Java API

Do you want to know more?

- ☞ Join Czech Scala Enthusiasts, a first Czech Scala networking and knowledge-sharing group
- ☞ <http://www.meetup.com/czech-scala-enthusiasts/>
- ☞ We meet regularly each month in Cafe Colore, with talks and workshops about Scala
- ☞ In November we have a special foreign guest **Heiko Seeberger** from Typesafe
- ☞ You may still visit CZJUG ;)



Q & A
