Chapter 32
Further Akka Actors

32.1 Introduction

The previous chapter introduced the basic Akka Actor concepts and constructs. This chapter takes these concepts further looking at how a result can be returned by (obtained from) an Actor. It also looks at Actor hierarchies and Actor supervision.

32.2 Generating a Result from an Actor

So far all our examples have received a message and then performed some operation. This operation has not returned a result and thus this has not been an issue. However, there are many situations where we do require a result. If you look at the specification for `receive` you will see that it is actually a `PartialFunction` which returns `Unit`. Thus the `receive` method cannot return a value.

To return a result from an actor when it has finished processing a message, you can use the `sender` property to send a result back to the requesting actor. Note this assumes that the requester is also an Actor itself; however, the next section on Futures will look at how we can retrieve a result when the sender is not an actor.

In this example we are using a companion object to act as a factory for the generation of the `Props` object associated with the Actors. Thus the `Calculator` Actor has a `Calculator` companion object with a method `props` that returns the `Props` instance configured as appropriate for the actor. The calculator actor is listed below:

© Springer International Publishing AG 2018

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_32

```
  object Calculator {

  def props = Props[Calculator]
}

class Calculator extends Actor {
  def receive = {
    case x: Int => {
      println("[redacted] received: " + x)
      val result1 =
           (1 to x).map(i => i)
                   .foldLeft(x)((total, i) => total * i)
      println(
            "Calculator processing completed, " +
            "returning result")
      sender ! result1
    }
  }
}
```

The key element of interest here is the `receive` method. This has a case pattern matcher that will handle Ints. Within the body of the associated behaviour a calculation is performed based on the integer received. Once this calculation is completed the result is sent back to the Actor that installed the request using the `sender` which is always bound to the actor that issued the request (if the message sender was indeed an Actor). Thus:

```
sender ! result
```

causes in a new message to be sent from the calculator actor to whatever actor issued the initial request.

The invoking actor, the `Controller` Actor, also has a companion object with a `props` factory method. In this case a string is passed in as a parameter to the constructor on the `Controller`. Thus the `Props` construction process requires a different syntax. This syntax allows information to be passed to the actors constructor. To do this we use an alternative `Props` factory method. This version of the Props factory takes the type of class to configure (the actor class) and the parameters to pass into the constructor of that class. Note any number of parameters can be passed here, as this is a variable length parameter list.

```
object Controller {
  // String controller passed as a parameter to the
  // contructor of the controller class
  def props = Props(classOf[Controller], "Controller")
}

class Controller(name: String) extends Actor {
  def receive = {
    case "start" => {
      println(name + ": starting calculator")
      context.actorOf(Calculator.props) ! 4
      println(name + ": message set")
    }
    case result: Int => {
      println(name + " received: " + result)
      context.stop(self)
    }
  }
}
```

The `Controller` actor itself has two case statements in its `receive` method. One matches the string "start" which obtains a reference to the Calculator actor and sends it a message containing the integer '4'. The other case condition matches an `Int` which is assumed to be the result of the calculator's calculations and is printed out. In this case, once it has done that it stops itself (and as the `Calculator` was created by the `Controller` as a child of the controller) it also stops the Calculator actor.

The simple application that exercises these actors is shown below:

```
object ReplyTest extends App {
   println("Starting ReplyTest application")
   val system = ActorSystem("mysystem")
   val controller = system.actorOf(Controller.props)
   controller ! "start"
   println("End of ReplyTest body")
   system terminate
}
```

This program obtains a reference to the `Controller` actor from the `ActorSystem` and sends it the message string "start".

Note that the three imports for these three elements are the usual:

```
import akka.actor.Actor
import akka.actor.Props
import akka.actor.ActorSystem
```

The output of this program is:

```
Starting ReplyTest application
End of ReplyTest body
Controller: starting calculator
Controller: message set
Calculator received: 4
Calculator processing completed, returning result
Controller received: 96
```

Notice that the two Strings printed by the main application body are at the start of the output. Thus the main program completes before the controller and the calculator start their message processing. This clearly indicates that the `Controller` and the `Calculator` are executing in their own threads of execution.

32.3 Futures

The previous example illustrates how one actor can return a result to another actor. However, what happens if the sending code is not part of an actor? One answer is to use a `scala.concurrent.Future`. A `Future` is a object which offers to return the result of some process or calculation, at a point in the future, allowing the current code to continue processing. Futures can be used in many different situations, such as reading data from a database, executing some long-running process, requesting data from a remote service.

The key to a `scala.concurrent.Future` is that it can take an actor, and the message to send to the actor, and wrap the resulting response message in a form that the requesting code can access. As in the last section a simple `Calculator` Actor will be used. This Actor again has a companion object to generate the properties for the `Calculator` and again the `receive` message handles Integers and returns a result by sending that result back to the `sender`.

```
  object Calculator2 {
    def props = Props(classOf[Calculator2])
  }

  class Calculator2 extends Actor {
    def receive = {
      case x: Int => {
        println("Calculator2 ->  received: " + x)
        val result1 =
          (1 to x).map(i => i)
            .foldLeft(x)((total, i) => total * i)
        println("Calculator2 ->  processing completed, " +
                " returning result")
        sender ! result1    }
    }
  }
```

The code that invokes the above behaviour however uses a Future:

```
import scala.concurrent.ExecutionContext
import scala.concurrent.Future
import scala.concurrent.duration._

import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props
import akka.pattern.ask
import akka.util.Timeout

import scala.util.{Success,Failure}

object FutureTest extends App {
  import scala.concurrent.ExecutionContext.Implicits.global
  println("FutureTest -> Starting ReplyTest application")
  val system = ActorSystem("mysystem")
  val calc = system.actorOf(Calculator2.props)

  val future: Future[Int] =
              ask(calc, 4)(Timeout(5 seconds)).mapTo[Int]
  println("FutureTest -> Obtained : " + future)
  future onComplete {
    case Success(i: Int) =>
           println(s"FutureTest -> Result from future: $i")
    case Failure(ex) => println(s"Failed: $ex")
  }
  println("FutureTest -> Finished")
  system.terminate
  System.exit(0)
}
```

There are a number of elements to note about this listing including:

1. An *implicit* declaration is used to provide an execution context for the future processing. This is required as Actors run within an execution context and thus the Future must also have an execution context. This implicit statement provides a suitable execution context that can be *implicitly* made available to the Future invocation. This is implemented as:

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

2. The `ask` function for the `akka.pattern` package is used to create the Future which will send the message containing the integer '4' to the `calc` Actor. It is a multiple parameter list function and thus the first parameter list takes the actor and the message and the second parameter list takes a timeout. This could also have been declared implicitly.
3. The `ask` function generates a `Future` which returns `Any` type of elements, whereas we know that the `Calculator` returns an Integer as a result and thus the `Future` returned by the `ask` function is then mapped to one that will work with an integer result using (`mapTo[Int]`).
4. The resulting Future is then stored in the val '`future`'. The main program then continues on and prints out a message. It then defines the behaviour to be invoked once a result is generated—note it does not wait for that result to be made available before continuing on.
5. The main programme then prints out the final string saying it has finished.
6. Once the calculation performed by the calculator has completed, the future `onComplete` behaviour is executed. Pattern matching is then used to determine whether the Actor completed successfully or whether in fact there was a failure generated. If the actor completed successfully then the future will contain the value to be retrieved in the Success object; if instead it failed (e.g. because an exception was thrown) then it will contain a Failure object.

The result of executing this program is shown below:

```
FutureTest -> Starting ReplyTest application
Calculator2 -> received: 4
Calculator2 -> processing completed, returning result
FutureTest -> Obtained: Future(<not completed>)
FutureTest -> Finished
FutureTest -> Result from future: 96
```

32.4 Dispatchers

In Akka it is a dispatcher (specifically a Message Dispatcher) that provides the foundation for an Akka Actor. That is a Message Dispatcher handles how any request received by the actor is processed.

Notice that the dispatcher triggers an execution as well as determining how the request (or message) will be processed. This is because an implementation of the `MessageDispatcher` is also an `ExecutionContext`. This means that it can execute arbitrary code such as that used to define the behaviour of the actor or to trigger a future.

In many (most) cases the default dispatcher will be more than adequate; however, it is possible to change an actors' dispatcher. To do this you need to have an alternative dispatcher available. This can be provided by defining an appropriate dispatcher in one of the Akka configuration files. There are two executors available with the default dispatcher. These are the *fork-join-executor* and the *thread-pool-executor*. The first executor forks a separate thread for each request to the Actor and then waits to rejoin that thread before continuing. The thread pool executor uses a pool of threads to process multiple requests in parallel across multiple instances of an Actor.

For example, the following dispatcher configuration defines a new thread-pool-based executor called jeh-pool-dispatcher.

```
akka {
 actor {
  jeh-pool-dispatcher {
   # Dispatcher is the name of the
   # event-based dispatcher
   type = Dispatcher
   # What kind of ExecutionService to use
   executor = "thread-pool-executor"
   # Configuration for the thread pool
   thread-pool-executor {
    # minimum number of threads to cap
    # factor-based core number to
    core-pool-size-min = 3
    # maximum number of threads to cap
    # factor-based number to
    core-pool-size-max = 3
   }
   # Throughput defines the maximum number
```

```
      # of messages to be processed per actor
      # before the thread jumps to the next actor.
      throughput = 5
    }
  }
}
```

The above specification states that the thread pool has an initial size of 3, with a maximum of 3 threads. Note that there are numerous properties available including a property to specify how long to keep alive a thread when it is not being used (default 60 s).

A thread pool is a set of underlying threads that can be used by instances of an actor to process messages. When a message is received, the Actor will attempt to obtain a new thread from the thread pool. If one is available (not being used) then that thread will be returned from the pool and processing of a message can start. If no threads are available then the request will wait until one thread is returned to the pool and is thus available to service a request. Note that it is not uncommon for a thread pool to have a minimum and a maximum size—thus when demand is low threads can be removed and at busy times the number of threads in the pool can grow (to some predefined maximum). This provides some control over the number of concurrent requests that can be processed at any one time. However, care needs to be taken when configuring such pools to ensure that a suitable set of defaults can be used otherwise the performance of the system can suffer due to the lack of threads (or thread starvation).

Also note some general rules of thumb regarding *throughput*:

- If you have a case where the number of threads is equal to the number of actors using the dispatcher, set the *throughput* number extremely high, like 1000.
- If your actors perform tasks that will take some time to complete and you need fairness to avoid starvation of other actors sharing the pool, set the *throughput* to 1.
- For general usage, start with the default value of 5 and tune this value for each dispatcher so that you get reasonable performance characteristics without the risk of making actors wait too long to handle messages in their mailboxes.

A significant point of confusion relating to threads and thread pools is that a specific instance of an Actor is *always* a singled threaded process. That is, every Actor instance, when it receives a request, will process that request from start to finish before moving onto the next request in its in box. However, multiple instances of an actor can be created and can be sent requests in parallel. The underlying dispatcher determines whether these are queued up for processing (the default behaviour) or whether the instances of an actor can run concurrently. The following simple application creates four instances of the `LongProcessor` actor and sends 4 messages to the actors. However, the `ActorSystem` uses the earlier configuration file and thus only allows three threads to be used at any one

time by a single *type* of Actor. Thus the first three actors obtain a thread for execution, while the fourth will have to wait until a thread becomes free:

```
package com.jjh.akka
import akka.actor.Props
import akka.actor.ActorSystem
import akka.actor.Actor

class LongProcessor extends Actor {
  def receive = {
    case s:String => {
      for (i <- 1 to 5) print(s)
      println
      Thread.sleep(2000)
      for (i <- 1 to 5) print(s)
      println
    }
  }
}

object AkkaConfigTest extends App {
   val props =
    Props[LongProcessor]
    .withDispatcher("akka.actor.jeh-pool-dispatcher")
   val system = ActorSystem("mysystem")
   val actor1 = system.actorOf(props, "actor1")
   val actor2 = system.actorOf(props, "actor2")
   val actor3 = system.actorOf(props, "actor3")
   val actor4 = system.actorOf(props, "actor4")
   println("Sending messages")
   actor1 ! "A"
   actor2 ! "B"
   actor3 ! "C"
   actor4 ! "D"
   println("Messages sent")
   system.terminate
}
```

Notice that the path to the dispatcher configuration is formed from the elements wrapping that definition. Thus the format in the configuration file wraps *akka* around *actor*, which is around the *jeh-pool-dispatcher*, and thus the path is `akka.actor.jeh-pool-dispatcher`. Also note that the `withDispatcher` method on the Props type is used to load the configuration information associated without custom pool-based dispatcher.

The output from this application is a mix of the output from the instances of the `LongProcessor` actor handling the "A", "B" and "C" strings. The instance handling the "D" string has to wait until a thread becomes free and thus does not start to run until after the instance processing the "C" string completed.

```
Sending messages
Messages sent
CCCCC
BBBBB
AAAAA
CCCCBBAABC
BB
DDDDD
AAA
DDDDD
```

32.5 Actor Hierarchies

A number of times in this chapter we have seen the term 'child' actor. However, we have not really defined what a child actor is. Within the Actor model there can be relationships between one Actor and another. If one Actor is created from within the context of another actor, then this Actor is a child of the originating Actor, while the originating Actor is referred to as a parent of the child actor. In addition any actor can create child actors. Thus there can be a hierarchy of actors where any particular actor can have a parent Actor and any number of child actors. This is illustrated in Fig. 32.1.

In addition any Actor can also have grandchildren and great grandchildren depending on the level of the hierarchy.

As an example, the following listing illustrates an Actor that creates two child actors and then sends messages to both the children. Note that these child actors are created using the Actors own context rather than the top-level ActorSystem:

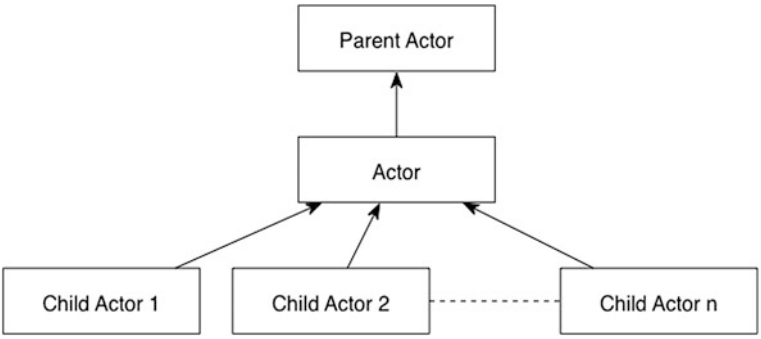

Fig. 32.1 Parent–Child relationships for an actor

```
class Supervisor extends Actor {
  var children = ArrayBuffer[ActorRef]()

  override def preStart(): Unit = {
    children += context.actorOf(Props(classOf[ChildActor],
"child1"))
    children += context.actorOf(Props(classOf[ChildActor],
"child2"))
  }
  def receive = {
    case "run" => {
      println("Supervisor run")
      children.foreach(a => a ! "msg")
    }
    case "error" => {
      println("Supervisor error")
      children.foreach(a => a ! "error")
    }
  }
}
```

In this case the `prestart` lifecycle method is being used to create the children. Child actors are created whenever the Actors `context` is used to create the actor using the `actorOf` method, rather than the `ActorSystem`. As this context is available in the `receive`, `prestart`, `preRestart`, etc., method, children can be created wherever appropriate within the Actor.

32.6 Actor Supervision

Other than an interesting concept, why is it significant that an Actor be considered a child of another actor? The answer relates to supervision of Actors. Within the Actor model supervision describes a dependency relationship between actors: the

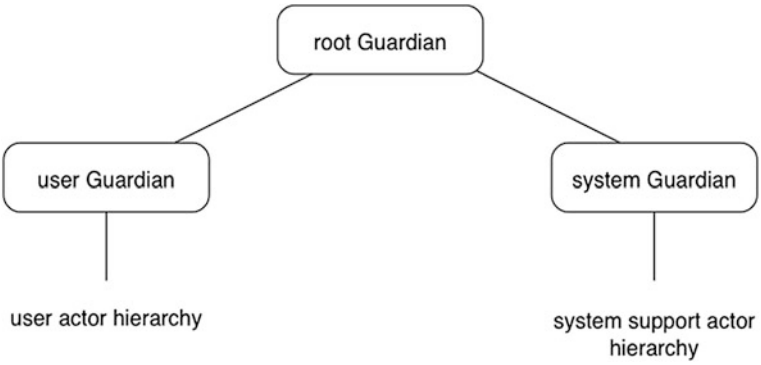

Fig. 32.2 Root of Guardianship hierarchy for all actors

supervisor or parent of an actor delegates tasks to its subordinates and therefore must respond to their failures or errors. When a subordinate or child detects a failure (i.e. it throws an exception) it suspends itself and any of its subordinates (or children) and sends a message to its supervisor, indicating the failure (i.e. the exception).

At this point the supervisor can determine what action to take, which includes:

1. **Resume** the subordinate keeping the current state of that subordinate.
2. **Restart** the subordinate that clears out the current state by creating a new Actor instance within the Actor incarnation.
3. **Stop** the subordinate permanently.
4. **Escalate** the failure to its parent.

Note that each of these is an object defined within the `SupervisorStrategy` type. The Supervisor strategy uses the object to determine what action is being requested by the programme.

This hierarchy of supervision extends right up to the top of the Actor framework. Thus an actor can receive failures which were generated by its children, grand children or great grand children. At the root of this hierarchy is the Root Guardian. The Root Guardian has two children; the User Guardian and the System Guardian (see Fig. 32.2). All user created actors are the responsibility of the User Guardian. In practice you would normally handle the errors yourself to ensure a clean and stable system rather than allowing them to propagate up to the User Guardian.

The Supervisor actor class presented in the previous section can now be modified to specify how it will deal with certain types of failure. This is achieved by specifying a *Supervision Strategy* and is implemented by overriding the default Supervision Strategy. Thus the Supervisor actor can now be defined as:

```
class Supervisor extends Actor {
  var children = ArrayBuffer[ActorRef]()

// Restart the child actors child when RuntimeException is
// thrown.
// After 3 restarts within 1 minute it will be stopped.
 override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 3,
                      withinTimeRange = 1 minute) {
         case _: RuntimeException =>
             { println("Supervisor Restarting"); Restart}
     }

override def preStart(): Unit = {
  children +=
     context.actorOf(Props(classOf[ChildActor], "child1"))
  children +=
     context.actorOf(Props(classOf[ChildActor], "child2"))
}
def receive = {
  case "run" => {
    println("Supervisor run")
    children.foreach(a => a ! "msg")
  }
  case "error" => {
    println("Supervisor error")
    children.foreach(a => a ! "error")
  }
}
}
```

Note that a `OneForOneStrategy` is being used here. There are actually two classes of Supervision Strategy available in Akka:

- **OneForOneStrategy**—which only applies the following action to the failed child. In our case this means that only the failed child is restarted.
- **AllForOneSrategy**—which applies the action it defines to all children.

These are defined in the `akka.actor` package along with other Supervision Strategy types such as the Restart action. Both the strategies presented above take information on the number of times a child can be restarted in a particular time period. In this example, the child can be restarted three times within 1 min. After this the child will be stopped. Both strategies also use pattern matching to determine what action to take when an exception is thrown. In our case whenever a `RuntimeException` is thrown by one of the children it will be handled by restarting that child.

There is nothing special about the actors that will become child actors other than the fact that they can throw a `RuntimeException` if the message "error" is sent to them:

```
class ChildActor(name: String) extends Actor {
  override def preStart(): Unit =
                { println(name + " preStart") }
  override def postStop(): Unit =
                { println(name + " postStop") }
  override def preRestart(reason: Throwable,
                          message: Option[Any]): Unit ={
    super.preRestart(reason, message)
    println(name + " preRestart")
  }
  override def postRestart(reason: Throwable): Unit = {
    println(name + " postRestart")
    super.postRestart(reason)
  }
  def receive = {
    case "error" => throw new RuntimeException(name)
    case _ => println(name + " received msg")
  }
}
```

The imports for this program are:

```
import scala.collection.mutable.ArrayBuffer
import scala.concurrent.duration.DurationInt

import akka.actor.Actor
import akka.actor.ActorRef
import akka.actor.ActorSystem
import akka.actor.OneForOneStrategy
import akka.actor.Props
import akka.actor.SupervisorStrategy.Restart
import akka.actor.actorRef2Scala
```

The sample program which uses the Supervisor looks like any other Actor program with nothing to indicate that the Supervisor is monitoring the child actors (or indeed that there are child actors involved):

```
object ParentChildActorTest extends App {
  val props = Props[Supervisor]
  val system = ActorSystem("mysystem")
  val actor = system.actorOf(props, "supervisor")
  actor ! "run"
  Thread.sleep(100)
  actor ! "error"
  actor ! "run"
  system terminate
}
```

The output from executing these classes is presented below:

```
Supervisor run
child1 preStart
child1 received msg
child2 preStart
child2 received msg
Supervisor error
Supervisor run
Supervisor Restarting
Supervisor Restarting
child2 postStop
child2 preRestart
child1 postStop
child1 preRestart
child2 postRestart
child1 postRestart
child2 preStart
child1 preStart
child1 received msg
child2 received msg
```

Note you can see the `preRestart` and `postRestart` methods producing output as well as the `preStart` and `postStop` methods' output in this example. This is because the supervisor is restarting the ChildActors following an exception being thrown by the children.

32.7 Good Practices

This section provides some guidelines on good practices when developing Akka Actor applications:

1. *Actors should not block while processing a message*. Such blocking will cause performance issues and may lead to deadlock between communicating Actors.
2. *Communicate with actors only* via *messages*. Do not try to define additional methods through which you will invoke actor functionality.
3. *Do not share state*. Scala Akka does not prevent actors from sharing state, so it is (unfortunately) very easy to do. Any shared mutable object represents state including objects in messages.
4. *Prefer immutable messages*. Ensure that all the data within your messages is immutable. Mutable data in a message is shared state.
5. *Make messages self-contained*. When you get a response from an actor, it may not be obvious what is being responded to. If the request is immutable, it is very inexpensive to include the original request as part of the response. The use of *case* classes often makes messages more readable.
6. *Factory methods*. Provide a factory method for the Actors property configuration:

```
object Calculator {
  def props = Props[Calculator]
}

class Calculator extends Actor {
  def receive = {
    case x: Int => {
      println("Calculator received: " + x)
      val result1 =
        (1 to x).map(i => i)
                .foldLeft(x)((total, i) => total * i)
      println("Calculator processing completed, " +
              "returning result")
      sender ! result1
    }
  }
}
```

Online References

Introduction to Actor Model—http://en.wikipedia.org/wiki/Actor_model
Akka Actor homepage—http://akka.io/