Akka Actor Basics Tour II

Kason Chan

This talk was compiled with tut

```
import akka.actor.{ActorRef, ActorSystem, DeadLetter, Inbox, Props
import com.typesafe.config.ConfigFactory

import scala.concurrent.Await
import scala.concurrent.duration._
```

Recap

- Akka
- Actor Model
- Actor System
  - How to create and terminate it
- Actor
  - An actor is a container for state, behavior, a mailbox, child actors and a supervisor strategy
  - How to handle messages should be processed (Define an actor class and receive method)
  - How to create and kill one in the actor system
  - How to handle incoming message
  - What to do before actor start and after actor is terminated
  - Messages which cannot be delivered are delivered to deadLetters

Recap

- Partial Function

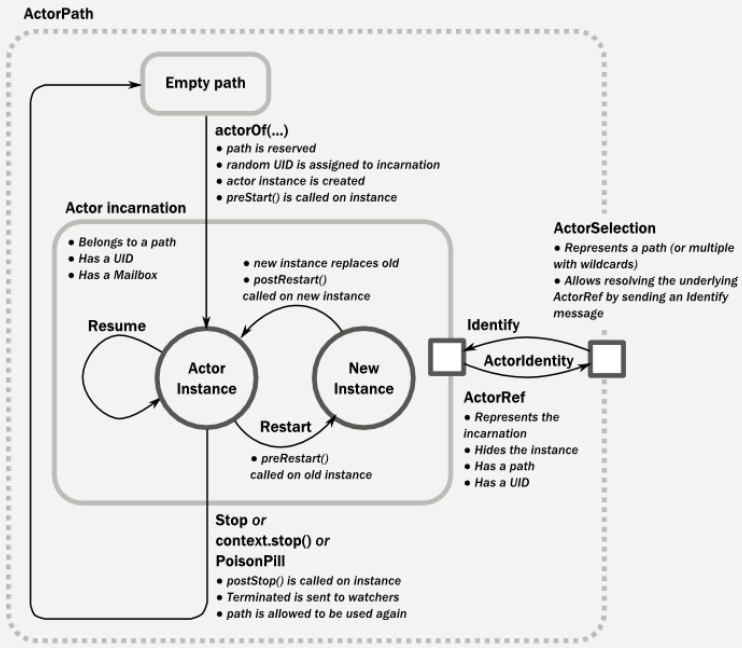
```
def receive: PartialFunction[Any, Unit] = {
    case msg: Msg =>
      log.info(msg)
    case any @ _ =>
      log.warning(any.toString)
  }
```

- A function works for every argument of the defined type.
- A function defined as Any => Unit takes Any type and returns Unit type.

- Props is a configuration class to specify options for the creation of actors, think of it as an immutable and thus freely shareable recipe for creating an actor including associated deployment information.

```
val you: ActorRef = system.actorOf(Props[Worker], "you")
```

Actor Lifecycle



Props

- How to create an actor with default constructor values? This is a recommended practice.

```
object MagicActor {
  def props(magicNumber: Int): Props = Props(new MagicActor(magicN
}

class MagicActor(magicNumber: Int) extends Actor {
  def receive = {
    case x: Int => sender() ! (x + magicNumber)
  }
}

// Create an actor named magicActor
val magicActor: ActorRef = system.actorOf(MagicActor.props(1), "ma
```

Recommended Practices

- Declare messages an actor can receive in the companion object of the Actor, which makes easier to know what it can receive

```
object MyActor {
  case class Greeting(from: String)
  case object Goodbye
}

class MyActor extends Actor with ActorLogging {
  import MyActor._
  def receive = {
    case Greeting(greeter) => log.info(s"I was greeted by $greeter
    case Goodbye           => log.info("Someone said goodbye to me
  }
}
```

ActorOf vs ActorSelection

- actorOf only ever creates a new actor, and it creates it as a direct child of the context on which this method is invoked (which may be any actor or actor system).

```
val you: ActorRef = system.actorOf(Props[Worker], "you")
```

- actorSelection only ever looks up existing actors when messages are delivered, i.e. does not create actors, or verify existence of actors when the selection is created.

```
// Select all ActorRef starting with you and send a Hey you messag
system.actorSelection("*/you*") ! "Hey you"
```

Send Messages

Messages can be sent to an actor through one of the following methods:

- ! means “fire-and-forget”, e.g. send a message asynchronously and return immediately. Also known as tell.

```
val msg: String = "Hello World!"
worker ! msg
i.tell(msg, you)
```

- ? sends a message asynchronously and returns a Future representing a possible reply. Also known as ask. Message ordering is guaranteed on a per-sender basis.

```
case object Ping
case object Pong
import akka.pattern.ask
me ? Ping // Success(Pong)
ask(i, Ping) // Success(Pong)
```

Send Messages

- Replies can be pipe to another actor:

```
import akka.pattern.pipe
val result: Future[Any] = me ? Ping
result2.pipeTo(you)
pipe(result2) to me
```

- Messages can be forward to another actor:

```
val msg: String = "Hello World!"
worker forward msg
```

Send Messages

- Messages can be set to be sent by a timer:

```
import akka.actor.Timers

object MyActor {
  private case object TickKey
  private case object FirstTick
  private case object Tick
  private case object LaterTick
}

class MyActor extends Actor with Timers {
  import MyActor._
  timers.startSingleTimer(TickKey, FirstTick, 500.millis)

  def receive = {
    case FirstTick =>
      // do something useful here
      timers.startPeriodicTimer(TickKey, Tick, 1.second)
    case Tick =>
    // do something useful here
  }
}
```

Terminate an Actor

- Kill message causes the actor to throw a ActorKilledException, triggering a failure.

```
worker ! Kill
```

- PoisonPill message will stop the actor when the message is processed.

```
worker ! PoisonPill
```

- stop self will stop the actor itself.

```
context stop self
```

- stop worker at the application.

```
system stop worker
```

Terminate an Actor and Exceptions

- Termination of an actor proceeds in two steps:

    1. the actor suspends its mailbox processing and sends a stop command to all its children, then it keeps processing the internal termination notifications from its children until the last one is gone
    2. Terminating itself. (invoking postStop, dumping mailbox, publishing Terminated on the DeathWatch, telling its supervisor)

- If an exception is thrown while a message is being processed, nothing happens to the mailbox.

- If the actor is restarted, the same mailbox will be there. All messages on that mailbox will be kept the same.

Supervision Strategies

- 2 classes of supervision strategies which come with Akka: OneForOneStrategy and AllForOneStrategy
  - OneForOneStrategy applies the obtained directive only to the failed child. It is the default strategy if none is specified explicitly.
  - AllForOneStrategy applies it to all siblings as well.

```
class Supervisor extends Actor with ActorLogging {

val maxNrOfRetries = 3

override val supervisorStrategy: OneForOneStrategy =
  OneForOneStrategy(maxNrOfRetries = maxNrOfRetries,
                    withinTimeRange = 1 minute) {
    case _: Exception => Escalate
}

...
```

Routing

- We can send messages to a router to efficiently route them to destination actors, known as its routees.
- A Router can be used inside or outside of an actor
- We can manage the routees yourselves or use a self contained router actor with configuration capabilities.
- Akka provides us the following routing logics:

```
akka.routing.RoundRobinRoutingLogic
akka.routing.RandomRoutingLogic
akka.routing.SmallestMailboxRoutingLogic
akka.routing.BroadcastRoutingLogic
akka.routing.ScatterGatherFirstCompletedRoutingLogic
akka.routing.TailChoppingRoutingLogic
akka.routing.ConsistentHashingRoutingLogic
```

Routing

```
class Supervisor extends Actor with ActorLogging {

  private val router: Router = {
    val routees = Vector.fill(5) {
      val r = context.actorOf(Props[Worker])
      context watch r
      ActorRefRoutee(r)
    }
    Router(RoundRobinRoutingLogic(), routees)
  }

  def receive: PartialFunction[Any, Unit] = {
    case any @ _ =>
      log.info(s"${sender().path} ${any.toString}")
      router.route(any, self)
  }

}
```

Demo

Recap

- Actor Lifecycle
- How to create an actor with default constructor values?
- ActorOf vs ActorSelection
- Send Messages - !, ?, pipeTo and forward
- Terminate Actors - Kill, PoisonPill, stop
- Supervision Strategies - OneForOneStrategy vs AllForOneStrategy
- Routing with different logics

Post Readings

- Actor Systems: http://doc.akka.io/docs/akka/current/scala/general/actor-systems.html
- What is an Actor? http://doc.akka.io/docs/akka/current/scala/general/actors.html
- Actors: http://doc.akka.io/docs/akka/current/scala/actors.html
- Message Delivery Reliability: http://doc.akka.io/docs/akka/current/scala/general/message-delivery-reliability.html
- One-For-One Strategy vs. All-For-One Strategy: http://doc.akka.io/docs/akka/snapshot/scala/general/supervision.ht for-one-strategy-vs-all-for-one-strategy
- Routing: http://doc.akka.io/docs/akka/snapshot/scala/routing.html

References

- Akka Docs: http://akka.io/docs/
- Reactive Messaging Patterns with Actor Model: https://github.com/VaughnVernon/ReactiveMessagingPatterns_Acto
- Akka in Acton: https://www.manning.com/books/akka-in-action

Thank you

- Q&A/Comments/Suggestions?