Area progettuale

Luca Scalzotto

September 2017

1 Introducing Akka

This work aims to investigate on some of the features provided by Akka framework. Akka present a good solution for concurrent programming. It also simplifies the scaling out and the scaling up of the systems.
Akka is centered on actors. Actors can receive messages one at a time and execute some behavior whenever a message is received. Messages are simple data structures that can't be changed after they've been created (immutable).
Actors are the building blocks of the system. An actor is a lightweight process that has only four operations:

- *Send* - an actor can only communicate with another actor by sending it messages. Sending messages is always asynchronous (fire and forget style).
- *Create* - an actor can create other actors; this automatically creates a hierarchy of actors.
- *Become* - an actor can have a different behavior depending on the state he is in.
- *Supervise* - any actor can be a supervisor, but only for actors that it creates itself. The supervisor decides what should happen when components fail in the system.

2 Test-driven development

Testing actors is more difficult than testing normal objects for many reasons:

- *Timing* - sending messages is asynchronous, so it's difficult to know when to assert expected values in the unit test.
- *Asynchronicity* - actors are meant to be run in parallel on several threads. Multi-threaded tests are more difficult than single-threaded tests and require concurrency primitives to synchronize results from various actors.
- *Statelessness* - an actor hides its internal state and doesn't allow access to this state.
- *Collaboration/Integration* - it's difficult to test the behavior of a group of actors that collaborate to reach a goal (integration test).

Luckily, Akka provides the akka-testkit module. This module contains a number of testing tools that makes testing actors a lot easier:

- *Single-threaded unit testing*
- *Multi-threaded unit testing*
- *Multiple JVM testing*

3 Fault tolerance

Akka simplifies the building of a fault-tolerant application. Akka provides two separate flows: one for normal logic and one for fault recovery logic. The normal flow consists of actors that handle normal messages; the recovery flow consists of actors that monitor the actors in the normal flow. Actors that monitor other actors are called supervisors. Instead of catching exceptions in an actor, we'll just let it crash. The actor code for handling messages only contains normal processing logic and no error handling or fault recovery logic, which keeps things much clearer. The mailbox for a crashed actor is suspended until the supervisor in the recovery flow has decided what to do with the exception. A supervisor doesn't "catch exceptions"; it decides what should happen with the crashed actors that it supervises base on the cause of the crash. The supervisor has four options:

- *Restart* - the actor must be re-created from its Props. After it's restarted, the actor will continue to process messages.
- *Resume* - the same actor instance should continue to process messages; the crash is ignored.
- *Stop* - the actor must be terminated. It will no longer take part in processing messages.
- *Escalate* - the supervisor doesn't know that to do with it and escalates the problem to its parent, which is also a supervisor.

3.1 Example

```
package it.infinity.main

import akka.actor.Actor
import akka.actor.Props
import akka.actor.OneForOneStrategy
import akka.actor.SupervisorStrategy._

class FirstActor extends Actor {

  override def preStart() = {
    context.actorOf(Props[SecondActor], name = "secondActor");
  }

  override def receive = {
    case msg: String =>
      println("Start handling message...");
      Thread.sleep(7000);
      println("Message: " + msg);
    case n: Integer =>
      throw ( new IllegalArgumentException("") );
  }

  override def supervisorStrategy = OneForOneStrategy() {
    case _: IllegalArgumentException =>
      println("Actor resumed.");
      Resume;
  }

}
```

```
package it.infinity.main

import akka.actor.Actor

class SecondActor extends Actor {

  override def receive = {
    case msg: String =>
      println("Message: " + msg);
    case n: Integer =>
      throw ( new IllegalArgumentException("") );
  }

}
```

```
package it.infinity.main

import akka.actor.ActorSystem
import akka.actor.Props
import akka.actor.PoisonPill
import akka.actor.DeadLetter

object Main {

  def main(args: Array[String]): Unit = {
    val system = ActorSystem("actorSystem");
    val firstActor = system.actorOf(Props[FirstActor], name = "firstActor")
    val secondActor = system.actorSelection("user/firstActor/secondActor");
    firstActor ! "Hello";
    Thread.sleep(1000);
    secondActor ! 3;
    println("Message sent");
    // Thread.sleep(3000);
    // secondActor ! "ciao";


    Thread.sleep(7000);
    system.terminate();
  }

}
```

4 Futures

Futures are extremely useful and simple tools for combining functions asynchronously. Actors provide a mechanism to build a system out of concurrent objects, futures provide a mechanism to build a system out of asynchronous functions. A future makes it possible to process the result of a function without ever waiting in the current thread for the result. It's a read-only placeholder for a function result (a success or failure) that will be available at some point in the future.

4.1 Use cases

- you don't want to block (wait on the current thread) to handle the result of a function
- calling a function once-off and handling the result at some point in the future
- combining many once-off functions and combining the results

- calling many competing functions and only using some of the results, for instance only the fastest response
- calling a function and returning a default result when the function throws an exception so the flow can continue
- pipe-lining the kind of functions, where one function depends on one or more results of other functions

4.2 Examples

Here are three examples of futures usage:

1. simple example of future usage
2. futures pipe-line
3. futures complex combination

```
package it.infinity.main

import scala.concurrent.ExecutionContext
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import scala.util.Success
import scala.util.Failure

object Application {

  def main(args: Array[String]): Unit = {
    simple();
    println("");
    pipeline();
    println("");
    combination();
  }

  private def simple(): Unit = {
    val future: Future[String] = Future {
      Thread.sleep(3000); // long-time operation
      "Hello world!";
    }

    future.onComplete {
      case Success(x) => println(x)
      case Failure(exception) => println(exception);
    }

    println("Waiting...");
    Thread.sleep(5000); // doing something else
  }

  private def pipeline(): Unit = {
    val future1: Future[String] = Future {
      Thread.sleep(3000); // long-time operation
      "Hello world!";
    }
```

```
    val future2: Future[Int] = future1.map { x =>
      Thread.sleep(3000); // long-time operation
      1;
    }

    future2.onComplete {
      case Success(x) => println(x)
      case Failure(exception) => println(exception);
    }

    println("Waiting...");
    Thread.sleep(7000); // doing something else
  }

  private def combination(): Unit = {
    val first = 1;
    val second = 2;
    val third = 3;
    val fourth = 4;

    val future1: Future[Int] = Future {
      Thread.sleep(5000); // long-time operation
      first;
    }

    val future2: Future[Int] = Future {
      Thread.sleep(6000); // long-time operation
      second;
    }

    val future3: Future[Int] = Future { third }
    val future4: Future[Int] = Future { fourth }

    val future5 = Future.firstCompletedOf(List(future1, future2));
    val future6 = future3.zip(future4).map {
      case (a, b) => a + b;
    }

    val future7 = future5.zip(future6).map {
      case (a, b) => a + b;
    }

    future7.onComplete {
      case Success(x) => println(x);
      case Failure(exception) => println(exception);
    }

    println("Waiting for result...");
    Thread.sleep(7000); // doing something else
  }

}
```

Look also at: future.onSuccess(), future.onFailure(), future.recover().

5 Finite-state machine modeling

There are many reason for using stateless components when implementing a system to avoid all kind of problem, like restoring state after an error. But in most cases, there are components within a system that need state to be able to provide the required functionality. A possible solution is to use the become functionality. Another possibility, is using finite state machine modeling. We'll see both in the following paragraphs.

5.1 Become

Akka supports hot-swapping the Actor's message loop (e.g. its implementation) at run-time: invoke the *context.become* method from within the *Actor*. *become* takes a *PartialFunction[Any, Unit]* that implements the new message handler. The hotswapped code is kept in a stack which can be pushed (*become* method) and popped (*unbecome* method).
N.B.: typically used when it is necessary to define a simple behavior (maximum two states).

5.2 Finite-state machine

A finite-state machine (FSM) is a common language-independent modeling technique. FSMs can model a number of problems; common applications are communication protocols, language parsing, and even business application problems. What they encourage is isolation of state; transitions from one state to another are caused by the arrival of predefined events and are understood as atomic operations.

Here is an example of finite-state machine that recognizes the regular expression *ab*a*:

```
package it.infinity.main

import akka.actor.Actor
import akka.actor.FSM

case class Message(x: String) { }
case class PrintMessage() { }

trait State { }

case object InitialState extends State;
case object FirstState extends State;
case object FinalState extends State;
case object ErrorState extends State;

case class StateData(str: String);

class MyActor extends Actor with FSM[State, StateData] {
  startWith(InitialState, StateData(""));

  when(InitialState) {
    case Event(message: Message, data: StateData) => {
      (message.x) match {
        case "a" => goto(FirstState) using StateData(data.str + "a");
        case "b" => goto(ErrorState) using StateData("");
      }
    }
    case Event(printMessage: PrintMessage, data: StateData) => {
      println("INFO: " + stateName + ", " + data.str);
      stay();
    }
```

```
}

when(FirstState) {
  case Event(message: Message, data: StateData) => {
    (message.x) match {
      case "a" => goto(FinalState) using StateData(data.str + "a");
      case "b" => stay() using StateData(data.str + "b");
    }
  }
  case Event(printMessage: PrintMessage, data: StateData) => {
    println("INFO: " + stateName + ", " + data.str);
    stay();
  }
}

when(FinalState) {
  case Event(message: Message, data: StateData) => {
    (message.x) match {
      case "a" => goto(ErrorState) using StateData("");
      case "b" => goto(ErrorState) using StateData("");
    }
  }
  case Event(printMessage: PrintMessage, data: StateData) => {
    println("INFO: " + stateName + ", " + data.str);
    stay();
  }
}

when(ErrorState) {
  case Event(message: Message, data: StateData) => {
    stay() using StateData("");
  }
  case Event(printMessage: PrintMessage, data: StateData) => {
    println("INFO: " + stateName + ", " + data.str);
    stay();
  }
}

whenUnhandled {
  case Event(event, stateData) => println("unhandled"); stay();
}

onTransition {
  case x -> y => {
    println("transition from " + x + " to " + y);
  }
}

onTermination {
  case StopEvent(FSM.Normal, state, stateData) => {
    println("Normal termination: " + state + ", " + stateData);
  }
}

initialize();
```

}

N.B.: the FSM trait implicitly defines an actor within it.

6 Distributed system with Akka

Distributed computing is hard, notoriously hard. Akka (in particular, the akka-remote module) provides a simple and elegant solution for communicating between actors across the network. Most network technologies use a blocking remote procedure call (RPC), which tries to mask the difference between calling an object locally or remotely. This style of communication works for point-to-point connections, but isn't a good solution for large-scale networks. Akka takes a different approach when it comes to scaling out applications across the network: you have transparency of remoting actors and you can continue to work with messages as before (in this case they'll be sent to remote actors). To get a reference to an actor on a remote node, you have to look up the actor by its path:

akka.tcp://back-end@host:port/user/actorPath

In addition, a configuration file need to be created in order to allow the remote look up.

6.1 Example

CLIENT

```
package it.infinity.model

import com.typesafe.config.ConfigFactory

import akka.actor._
import java.io.File

object Application {

  def main(args: Array[String]): Unit = {
    val config = ConfigFactory.parseFile(new File("./local/application.conf"));
    val system = ActorSystem("local", config);
    val localActor = system.actorOf(Props[LocalActor], name = "localactor")
  }

  class LocalActor extends Actor {
    var counter = 0

    override def preStart() {
      val selection = context.actorSelection("akka.tcp://remote@192.168.1.127:2555/us
      selection.tell(Message("Hello"), self);
    }

    def receive = {
      case msg: String =>
        println("LocalActor received message: " + msg)
        if (counter < 5) {
          sender ! "Hello back to you"
          counter += 1
        }
    }
  }
```

```
  case class Message(val content: String) {}
}
```

SERVER

```
package it.infinity.model

import akka.actor.Actor
import akka.actor.Props
import akka.actor.ActorSystem
import com.typesafe.config.ConfigFactory
import java.io.File

object Application {

  def main(args: Array[String]): Unit = {
    val config = ConfigFactory.parseFile(new File("./remote/application.conf"));
    val system = ActorSystem("remote", config);
    val remoteActor = system.actorOf(Props[RemoteActor], name = "remoteactor")
  }

  class RemoteActor extends Actor {
    def receive = {
      case msg: String ⇒
        println("string: " + msg);
        sender ! msg;
      case Message(content) ⇒ println("message: " + content); sender ! content;
    }
  }

  case class Message(val content:String) { }

}
```

6.2 Configuration files

CLIENT

```
akka {
    stdout-loglevel = "OFF"
    loglevel = "OFF"

    actor {
        provider = "akka.remote.RemoteActorRefProvider"
    }
    remote {
        enabled-transports = ["akka.remote.netty.tcp"]
        netty.tcp {
            hostname = localhost
            port = 0
        }
    }
}
```

SERVER

```
akka {
    stdout-loglevel = "OFF"
    loglevel = "OFF"

    actor {
        provider = "akka.remote.RemoteActorRefProvider"
    }
    remote {
        enabled-transports = ["akka.remote.netty.tcp"]
        netty.tcp {
            hostname = 192.168.1.127
            port = 2555
        }
    }
}
```

7 Message channels

Akka has defined a generalized interface: the *EventBus*, which can be implemented to create a publish-subscribe channel. An *EventBus* is generalized so that it can be use for all implementations of a public-subscribe channel. In the generalized form, there are three entities:

- *Event*: this is the type of all events published on that bus.
- *Subscriber*: this is the type of subscriber allowed to register on that event bus (e.g. *ActorRef*).
- *Classifier*: this defines the classifier to be used in selecting subscribers for dispatching events.

Here is the complete interface of the *EventBus*:

```
package akka.event

trait EventBus {
  type Event
  type Classifier
  type Subscriber

  def subscribe(subscriber: Subscriber, to: Classifier): Boolean

  def unsubscribe(subscriber: Subscriber, from: Classifier): Boolean

  def unsubscribe(subscriber: Subscriber): Unit

  def publish(event: Event): Unit
}
```

The whole interface has to be implemented, and because most implementations need the same functionality, Akka also has a set of composable traits implementing the *EventBus* interface, which can be used to easily create your own implementation of *EventBus*. Akka has three composable traits that can help you track of the subscribers. All these traits are still generic, so they can be used with any entities you have defined.

- *LookupClassification*: this trait uses the most basic classification. It maintains a set of subscribers for each possible classifier and extracts a classifier from each event. It extracts a classifier using the *classify* method, which should be implemented by the custom *EventBus* implementation.

- *SubchannelClassification*: this trait is used when classifiers form a hierarchy and it is desired that subscription be possible not only at the leaf nodes, but also at the higher nodes.
- *ScanningClassification*: this trait is a more complex one; i can be used when classifiers have an overlap. This means that one *Event* can be part of more classifiers.

In the following examples we'll use the *LookupClassification*. This trait implement the *subscribe* and *unsubscribe* methods of the *EventBus* interface. But it also introduce new abstract methods that need to be implemented in our class:

- *classify(event: Event): Classifier* - this is used for extracting the classifier from the incoming events.
- *publish(event: Event, subscriber: Subscriber): Unit* - this method will be invoked for each event for all subscribers that registered themselves for the events classifier.
- *mapSize: Int* - this returns the expected number of the different classifiers. This is used for the initial size of an internal data structure.

7.1 Example 1 - local

```
package it.infinity.main

import akka.actor.Actor

class MyActor extends Actor {

  def receive = {
    case message: String ⇒
      println("Message: " + message + ", " + self.path.name);
  }

}
```

```
package it.infinity.main

import akka.event.EventBus
import akka.event.LookupClassification
import javax.sql.StatementEvent
import akka.event.ActorEventBus
import akka.actor.ActorRef

class MessageBus extends EventBus with LookupClassification with ActorEventBus {
  type Event = String
  type Classifier = Boolean
  override type Subscriber = ActorRef

  def mapSize() = 2

  protected def classify(event: String) = {
    !event.isEmpty();
  }

  protected def publish(event: String, subscriber: Subscriber) {
    subscriber ! event;
  }

}
```

```
package it.infinity.main

import akka.actor.ActorSystem
import akka.actor.Props

object Main {

  def main(args: Array[String]): Unit = {
    val system = ActorSystem("actorSystem");
    val actor1 = system.actorOf(Props[MyActor], name = "actor1")
    val actor2 = system.actorOf(Props[MyActor], name = "actor2")

    val messageBus = new MessageBus();
    messageBus.subscribe(actor1, false);
    messageBus.subscribe(actor2, true);

    messageBus.publish("Ciao");
    messageBus.publish("");

    Thread.sleep(3000);

    system.terminate();
  }

}
```

7.2 Example 2 - remote

The EventBus itself is local, meaning that events are not automatically transferred to EventBuses on other systems, but you can subscribe any ActorRef you want, including remote ones. You only need an actor on the node where the EventBus is that takes care of subscribing remote actors and publishing their messages.

7.2.1 Client

```
package it.infinity.model

import com.typesafe.config.ConfigFactory

import akka.actor._
import java.io.File

object Application {

  def main(args: Array[String]): Unit = {
    val file = new File("");
    val config = ConfigFactory.parseFile(new File("./local/application.conf"));
    val system = ActorSystem("local", config);
    val local1 = system.actorOf(Props[LocalActor], name = "local1")
    val local2 = system.actorOf(Props[LocalActor], name = "local2")
    val connection = "akka.tcp://remote@localhost:2555/user/remoteActor";
    val remote = system.actorSelection(connection);

    remote ! Subscribe(local1, 0);
    remote ! Subscribe(local2, 0);
```

```
    remote ! Publish("Hello actor!");
  }

}

class LocalActor extends Actor {

  def receive = {
    case message: String => {
      println("(" + self.path.name + ")" + " " + message);
    }
  }

}

case class Subscribe(val actorRef: ActorRef, val n: Int) { }
case class Publish(val content: String) { }
```

7.2.2 Server

```
package it.infinity.model

import akka.event.EventBus
import akka.event.LookupClassification
import javax.sql.StatementEvent
import akka.event.ActorEventBus
import akka.actor.ActorRef

class MessageBus extends EventBus with LookupClassification with ActorEventBus {
  type Event = String
  type Classifier = Int
  override type Subscriber = ActorRef

  def mapSize() = 1

  protected def classify(event: String) = {
    0;
  }

  protected def publish(event: String, subscriber: Subscriber) {
    subscriber ! event;
  }

}

package it.infinity.model

import akka.actor.Actor
import akka.actor.ActorRef

class RemoteActor extends Actor {
  private val messageBus: MessageBus = new MessageBus();

  def receive = {
    case Subscribe(actorRef, n) => messageBus.subscribe(actorRef, n);
    case Publish(message) => messageBus.publish(message);
```

```
  }
}

case class Subscribe(val actorRef: ActorRef, val n: Int) { }
case class Publish(val content: String) { }
```

8 REST Api

The akka-http module provides an API to build HTTP clients and servers. The REST architectural style can be used for defining the API of the HTTP service. Defining routes is done by using directives. A directive is a rule that the received HTTP request should match. A directive has one or more of the following functions:

- Transforms the request
- Filters the request
- Completes the request

Directives are small building blocks out of which you can construct arbitrarily complex route and handling structures. The generic form is this:

$$name(arguments)\ \{\ extractions = \ldots\ \}$$

akka-http has a lot of predefined directives like get, post, put, delete. The route needs to complete the HTTP request with an HTTP response for every matched pattern.

8.1 Example

```
package it.infinity.main

import java.io.File

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import scala.io.StdIn

import com.typesafe.config.ConfigFactory

import akka.actor.ActorSystem
import akka.http.scaladsl.Http
import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport.sprayJsonMarshaller
import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport.sprayJsonUnmarshalle
import akka.http.scaladsl.marshalling.ToResponseMarshallable.apply
import akka.http.scaladsl.model.StatusCodes
import akka.http.scaladsl.server.Directive.addByNameNullaryApply
import akka.http.scaladsl.server.Directive.addDirectiveApply
import akka.http.scaladsl.server.Directives.LongNumber
import akka.http.scaladsl.server.Directives._enhanceRouteWithConcatenation
import akka.http.scaladsl.server.Directives._segmentStringToPathMatcher
import akka.http.scaladsl.server.Directives.as
import akka.http.scaladsl.server.Directives.complete
import akka.http.scaladsl.server.Directives.entity
import akka.http.scaladsl.server.Directives.get
import akka.http.scaladsl.server.Directives.onComplete
import akka.http.scaladsl.server.Directives.onSuccess
```

```
import akka.http.scaladsl.server.Directives.path
import akka.http.scaladsl.server.Directives.pathPrefix
import akka.http.scaladsl.server.Directives.post
import akka.http.scaladsl.server.Route
import akka.http.scaladsl.server.RouteResult.route2HandlerFlow
import akka.http.scaladsl.server.directives.OnSuccessMagnet.apply
import akka.stream.ActorMaterializer
import spray.json.DefaultJsonProtocol.LongJsonFormat
import spray.json.DefaultJsonProtocol.StringJsonFormat
import spray.json.DefaultJsonProtocol.jsonFormat2

object Application {

  implicit val itemFormat = jsonFormat2(Item);

  private val data = new Data();

  def main(args: Array[String]): Unit = {
    val file = new File("./application.conf");
    val config = ConfigFactory.parseFile(file);
    implicit val system = ActorSystem("system", config);
    implicit val materializer = ActorMaterializer();

    val route =
      get {
        pathPrefix("item" / LongNumber) { id =>
          val maybeItem: Future[Option[Item]] = findItem(id);

          onSuccess(maybeItem) {
            case Some(item) => complete(maybeItem);
            case None => complete(StatusCodes.NotFound)
          }
        }
      } ~
      post {
        path("create") {
          entity(as[Item]) { item =>
            val saved: Future[Unit] = saveItem(item);
            onComplete(saved) {
              done => complete(StatusCodes.Created);
            }
          }
        }
      }

    val bindingFuture = Http().bindAndHandle(route, "localhost", 8080);
    println("Server online at 'http://localhost:8080'");
    println("Presso RETURN to stop...");
    StdIn.readLine();

    bindingFuture.map { serverBinding => serverBinding.unbind() }
    bindingFuture.onComplete { serverBinding => system.terminate() }
  }

  def findItem(id: Long) = {
    val future = Future { data.getItem(id) }
```

```
    future;
  }

  def saveItem(item: Item) = {
    val future = Future { data.addItem(item) }
    future;
  }
}
```

9 Conclusions

Fault tolerance	”let it crash”, supervision strategy	...
Finite-state machine	become/unbecome, FSM trait	...
Distributed system	configuration files	...
Event	EventBus	...
Rest	Rest API (route, directives, path)	...
Transaction	Transactor	...