EBook Gratis

APRENDIZAJE akka

Free unaffiliated eBook created from **Stack Overflow contributors.**

#akka

Tabla de contenido

Acerca de 1

Capítulo 1: Empezando con akka 2

Observaciones 2

Examples 2

Instalación o configuración 2

Capítulo 2: Actor DSL 4

Examples 4

Simple actor DSL 4

Cambio de contexto 4

Gestión del ciclo de vida 4

Actores anidados 5

Supervisión 5

Soporte de alijo 5

Capítulo 3: Akka HTTP 7

Introducción 7

Examples 7

Servidor HTTP Akka: Hello World (Scala DSL) 7

Capítulo 4: akka-streams formas personalizadas 8

Observaciones 8

Examples 8

TwoThreeShape 8

Capítulo 5: Corrientes de Akka 10

Examples 10

Akka Streams: Hola Mundo 10

Akka-Streams: subflujos 12

Capítulo 6: Despachadores 13

Examples 13

Despachador predeterminado 13

Configuración del despachador para un actor 13

Capítulo 7: Hola Mundo 15

Examples 15
Akka hola mundo (Scala) 15
Implementación de actor simple 16
Akka Hello World (Java 8) 17
Capítulo 8: Inyectando dependencias en un actor. 19
Examples 19
Actor de primavera 19
Capítulo 9: Supervisión y Seguimiento en Akka 22
Observaciones 22
Examples 22
¿Qué es la supervisión? 22
Estrategias de supervisión 24
¿Qué es el monitoreo? 25
Repositorio de código 26
Creditos 27

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: akka

It is an unofficial and free akka ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official akka.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con akka

Observaciones

Akka es un conjunto de herramientas de código abierto y tiempo de ejecución que simplifica la construcción de aplicaciones concurrentes y distribuidas en la JVM. Implementa el modelo de actor conocido de Erlang.

También debe mencionar cualquier tema importante dentro de akka, y vincular a los temas relacionados. Dado que la Documentación para akka es nueva, es posible que deba crear versiones iniciales de los temas relacionados.

Examples

Instalación o configuración

1. Instale JDK 8 (Windows , Linux) y establezca la ruta (Windows).
2. Instale Scala (Linux), Para Windows visite http://www.scala-lang.org/download/ descargue e instale la distribución binaria, establezca la variable de entorno para scala en PATH que está en `\scala\bin` .
3. Instalación del activador Typesafe (contiene Scala, Akka, Play, SBT) + andamios y plantillas de proyectos. Para un inicio rápido descargue el `mini-package` .
4. Extraiga el activador Typesafe y establezca el PATH en `activator-xxxx-minimal\bin` (incluye los scripts de bash y bat para ejecutar el activador).
5. Es hora de crear un proyecto de muestra e importarlo en su IDE favorito.

- Tipo `activator new` en cmd / terminal.

```
Fetching the latest list of templates...

Browse the list of templates: http://lightbend.com/activator/templates
Choose from these featured templates or enter a template name:
  1) minimal-akka-java-seed
  2) minimal-akka-scala-seed
  3) minimal-java
  4) minimal-scala
  5) play-java
  6) play-scala
(hit tab to see a list of all templates)
>
```

- Puedes elegir `4` porque el ejemplo de Hello World está basado en Scala.
- Importe el proyecto a su IDE favorito y comience con el ejemplo de Hello World .
- Hecho !.

1. Descargue la distribución akka-2.0.zip de Akka desde http://akka.io/downloads/

2. Descomprima akka-2.0.zip en cualquier directorio. (Ejemplo: /home/USERNAME/tools/akka-2.0) Le gustaría tener instalado Akka.

3. Establecer el `AKKA_HOME`

4. Para linux

```
# First got to the installed location
cd /home/USERNAME/tools/akka-2.0

# Export the location as AKKA_HOME
export AKKA_HOME=`pwd`

# Check if PATH is Exported.
echo $AKKA_HOME
/home/USERNAME/tools/akka-2.0
```

5. Para ventanas

```
# First got to the installed location
C:\USERNAME\akka> cd akka-2.0

# Set the location as AKKA_HOME
C:\USERNAME\akka\akka-2.0> set AKKA_HOME=%cd%

# Check if PATH is Exported.
C:\USERNAME\akka\akka\akka-2.0> echo %AKKA_HOME%
C:\USERNAME\akka\akka-2.0
```

Lea Empezando con akka en línea: https://riptutorial.com/es/akka/topic/2041/empezando-con-akka

Capítulo 2: Actor DSL

Examples

Simple actor DSL

Para crear actores simples sin crear una nueva clase, puede usar:

```
import akka.actor.ActorDSL._
import akka.actor.ActorSystem

implicit val system = ActorSystem("demo")

val a = actor(new Act {
  become {
    case "hello" ⇒ sender() ! "hi"
  }
})
```

Cambio de contexto

Las dos formas posibles de emitir un context.become (reemplazar o agregar el nuevo comportamiento) se ofrecen por separado para habilitar una notación de recibos anidados sin desorden:

```
val a = actor(new Act {
  become { // this will replace the initial (empty) behavior
    case "info" ⇒ sender() ! "A"
    case "switch" ⇒
      becomeStacked { // this will stack upon the "A" behavior
        case "info"   ⇒ sender() ! "B"
        case "switch" ⇒ unbecome() // return to the "A" behavior
      }
    case "lobotomize" ⇒ unbecome() // OH NOES: Actor.emptyBehavior
  }
})
```

Gestión del ciclo de vida

Los ganchos del ciclo de vida también se exponen como elementos DSL, donde las invocaciones posteriores de los métodos que se muestran a continuación reemplazarán el contenido de los ganchos respectivos:

```
val a = actor(new Act {
  whenStarting { testActor ! "started" }
  whenStopping { testActor ! "stopped" }
})
```

Lo anterior es suficiente si el ciclo de vida lógico del actor coincide con los ciclos de reinicio (es decir, cuándo se ejecuta la Detención antes de un reinicio y cuándo se inicia luego). Si eso no se

desea, use los siguientes dos ganchos:

```
val a = actor(new Act {
  become {
    case "die" ⇒ throw new Exception
  }
  whenFailing { case m @ (cause, msg) ⇒ testActor ! m }
  whenRestarted { cause ⇒ testActor ! cause }
})
```

Actores anidados

También es posible crear actores anidados, es decir, nietos, como este:

```
// here we pass in the ActorRefFactory explicitly as an example
val a = actor(system, "fred")(new Act {
  val b = actor("barney")(new Act {
    whenStarting { context.parent ! ("hello from " + self.path) }
  })
  become {
    case x ⇒ testActor ! x
  }
})
```

Supervisión

También es posible asignar una estrategia de supervisión a estos actores con lo siguiente:

```
superviseWith(OneForOneStrategy() {
  case e: Exception if e.getMessage == "hello" ⇒ Stop
  case _: Exception                            ⇒ Resume
})
```

Soporte de alijo

Por último, pero no menos importante, hay un poco de magia de conveniencia incorporada, que detecta si la clase de tiempo de ejecución del subtipo de actor dado estáticamente extiende el rasgo RequiereMessageQueue a través del rasgo Stash (esta es una manera complicada de decir que la nueva Ley con Stash no funciona porque su tipo de tiempo de ejecución borrado es solo un subtipo anónimo de Ley). El propósito es utilizar automáticamente el tipo de buzón basado en deque apropiado requerido por Stash. Si quieres usar esta magia, simplemente extiende ActWithStash:

```
val a = actor(new ActWithStash {
  become {
    case 1 ⇒ stash()
    case 2 ⇒
      testActor ! 2; unstashAll(); becomeStacked {
        case 1 ⇒ testActor ! 1; unbecome()
      }
  }
})
```

Lea Actor DSL en línea: https://riptutorial.com/es/akka/topic/2392/actor-dsl

Capítulo 3: Akka HTTP

Introducción

Akka HTTP es un servidor HTTP liviano y una biblioteca cliente, que utiliza akka-streams bajo el capó

Examples

Servidor HTTP Akka: Hello World (Scala DSL)

La siguiente aplicación iniciará una escucha del servidor HTTP en el puerto 8080 que devuelve `Hello world` en `GET /hello/world`

```
import akka.actor.ActorSystem
import akka.http.scaladsl.Http
import akka.http.scaladsl.server.Directives._
import akka.http.scaladsl.server._
import akka.stream.ActorMaterializer

import scala.concurrent.Await
import scala.concurrent.duration.Duration

object HelloWorld extends App {

  implicit val system = ActorSystem("ProxySystem")
  implicit val mat = ActorMaterializer()

  val route: Route = get {
    path("hello" / "world") {
      complete("Hello world")
    }
  }

  val bindingFuture = Http().bindAndHandle(Route.handlerFlow(route), "127.0.0.1", port = 8080)

  Await.result(system.whenTerminated, Duration.Inf)

}
```

Lea Akka HTTP en línea: https://riptutorial.com/es/akka/topic/10108/akka-http

Capítulo 4: akka-streams formas personalizadas

Observaciones

akka proporciona algunas formas predefinidas, que probablemente deberían ajustarse al 99.9% de su uso. crear una nueva forma solo se debe hacer en casos muy raros. Las formas predefinidas son:

- `Source` - 1 salida, sin entradas
- `Sink` - 1 entrada, sin salidas
- `Flow` - 1 entrada, 1 salida
- `BidiFlow` - 2 entradas, 2 salidas
- `Closed` - no hay entradas, no hay salidas
- `FanInN` - `N` entradas (`N` <= 22), 1 salida
- `FanOutN` - `N` salidas (`N` <= 22), 1 entrada
- `UniformFanIn` - cualquier número de entradas del mismo tipo, 1 salida
- `UniformFanOut` - cualquier número de salidas del mismo tipo, 1 entrada
- `Amorphous` - cualquier cantidad de entradas o salidas, pero sin tipo.

Examples

TwoThreeShape

un ejemplo simple de cómo definir una forma personalizada con 2 entradas y 3 salidas.

```
case class TwoThreeShape[-In1, -In2, +Out1, +Out2, +Out3](
          in1: Inlet[In1@uncheckedVariance],
          in2: Inlet[In2@uncheckedVariance],
          out1: Outlet[Out1@uncheckedVariance],
          out2: Outlet[Out2@uncheckedVariance],
          out3: Outlet[Out3@uncheckedVariance]) extends Shape {

  override val inlets: immutable.Seq[Inlet[_]] = List(in1, in2)
  override val outlets: immutable.Seq[Outlet[_]] = List(out1, out2, out3)

  override def deepCopy(): TwoThreeShape[In1, In2, Out1, Out2, Out3] =
    TwoThreeShape(in1.carbonCopy(),
                  in2.carbonCopy(),
                  out1.carbonCopy(),
                  out2.carbonCopy(),
                  out3.carbonCopy())

  override def copyFromPorts(inlets: immutable.Seq[Inlet[_]], outlets:
immutable.Seq[Outlet[_]]): Shape = {
    require(inlets.size == 2, s"proposed inlets [${inlets.mkString(", ")}] do not fit
TwoThreeShape")
    require(outlets.size == 3, s"proposed outlets [${outlets.mkString(", ")}] do not fit
TwoThreeShape")
```

```
    TwoThreeShape(inlets(0), inlets(1), outlets(0), outlets(1), outlets(2))
  }
}
```

Un ejemplo de uso para esta forma extraña: una etapa que pasará a través de elementos de 2 flujos, mientras mantiene una proporción de cuántos elementos pasaron en los flujos:

```
def ratioCount[X,Y]: Graph[TwoThreeShape[X,Y,X,Y,(Int,Int)],NotUsed] = {
  GraphDSL.create() { implicit b =>
    import GraphDSL.Implicits._

    val x = b.add(Broadcast[X](2))
    val y = b.add(Broadcast[Y](2))
    val z = b.add(Zip[Int,Int])

    x.out(1).conflateWithSeed(_ => 1)((count,_) => count + 1) ~> z.in0
    y.out(1).conflateWithSeed(_ => 1)((count,_) => count + 1) ~> z.in1

    TwoThreeShape(x.in,y.in,x.out(0),y.out(0),z.out)
  }
}
```

Lea akka-streams formas personalizadas en línea: https://riptutorial.com/es/akka/topic/3282/akka-streams-formas-personalizadas

Capítulo 5: Corrientes de Akka

Examples

Akka Streams: Hola Mundo

Akka Streams le permite crear fácilmente una secuencia que aproveche el poder del marco Akka sin definir explícitamente los comportamientos y mensajes de los actores. Cada corriente tendrá al menos una `Source` (origen de los datos) y al menos un `Sink` (destino de los datos).

```
import akka.actor.ActorSystem
import akka.stream.ActorMaterializer
import akka.stream.scaladsl.{Sink, Source}
import java.io.File

val stream = Source(Seq("test1.txt", "test2.txt", "test3.txt"))
  .map(new File(_))
  .filter(_.exists())
  .filter(_.length() != 0)
  .to(Sink.foreach(f => println(s"Absolute path: ${f.getAbsolutePath}")))
```

En este ejemplo rápido tenemos una `Seq` de nombres de archivos que ingresamos en la secuencia. Primero los asignamos a un `File` , luego filtramos los archivos que no existen, luego los archivos cuya longitud es 0. Si un archivo pasó por los filtros, se imprime en la `stdout` .

Akka streams también te permite hacer streams de forma modular. Puedes crear `Flow` con los módulos parciales de tu flujo. Si tomamos el mismo ejemplo también podríamos hacer:

```
import akka.actor.ActorSystem
import akka.stream.ActorMaterializer
import akka.stream.scaladsl.{Sink, Source}
import java.io.File

implicit val actorSystem = ActorSystem("system")
implicit val actorMaterializer = ActorMaterializer()

val source = Source(List("test1.txt", "test2.txt", "test3.txt"))
val mapper = Flow[String].map(new File(_))
val existsFilter = Flow[File].filter(_.exists())
val lengthZeroFilter = Flow[File].filter(_.length() != 0)
val sink = Sink.foreach[File](f => println(s"Absolute path: ${f.getAbsolutePath}"))

val stream = source
  .via(mapper)
  .via(existsFilter)
  .via(lengthZeroFilter)
  .to(sink)

stream.run()
```

En esta segunda versión podemos ver que el `mapper` , `existsFilter` , `lengthZeroFilter` son `Flow` s. Puedes componerlos en transmisión usando el método a `via` . Esta capacidad le permitiría

reutilizar sus piezas de código. Una cosa importante a mencionar es que los `Flow` s pueden ser sin estado o con estado. En el caso de estado, debe tener cuidado al reutilizarlos.

También puedes pensar en los streams como `Graphs` . Akka Streams también proporciona un potente `GraphDSL` para definir secuencias complicadas de una manera sencilla. Siguiendo con el mismo ejemplo podríamos hacer:

```
import java.io.File
import akka.actor.ActorSystem
import akka.stream.{ActorMaterializer, ClosedShape}
import akka.stream.scaladsl.{Flow, GraphDSL, RunnableGraph, Sink, Source}

implicit val actorSystem = ActorSystem("system")
implicit val actorMaterializer = ActorMaterializer()

val graph = RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val source = Source(List("test1.txt", "test2.txt", "test3.txt"))
  val mapper = Flow[String].map(new File(_))
  val existsFilter = Flow[File].filter(_.exists())
  val lengthZeroFilter = Flow[File].filter(_.length() != 0)
  val sink = Sink.foreach[File](f => println(s"Absolute path: ${f.getAbsolutePath}"))

  source ~> mapper ~> existsFilter ~> lengthZeroFilter ~> sink

  ClosedShape
})

graph.run()
```

También es posible crear un flujo agregado utilizando `GraphDSL` . Por ejemplo, si nos gustaría combinar el mapeador y dos filtros en uno, podríamos hacerlo:

```
val combinedFlow = Flow.fromGraph(GraphDSL.create() { implicit builder =>
  import GraphDSL.Implicits._

  val mapper = builder.add(Flow[String].map(new File(_)))
  val existsFilter = builder.add(Flow[File].filter(_.exists()))
  val lengthZeroFilter = builder.add(Flow[File].filter(_.length() != 0))

  mapper ~> existsFilter ~> lengthZeroFilter

  FlowShape(mapper.in, lengthZeroFilter.out)
})
```

Y luego usarlo como un bloque individual. `combinedFlow` sería un `FlowShape` o un `PartialGraph` . Nos podemos por ejemplo con `via` :

```
val stream = source
  .via(combinedFlow)
  .to(sink)

stream.run()
```

O usando el `GraphDSL` :

```
val graph = RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val source = Source(List("test1.txt", "test2.txt", "test3.txt"))
  val sink = Sink.foreach[File](f => println(s"Absolute path: ${f.getAbsolutePath}"))

  source ~> combinedFlow ~> sink

  ClosedShape
})

graph.run()
```

Akka-Streams: subflujos

Puede `groupBy` dinámicamente un flujo en múltiples `groupBy` usando `groupBy` . Las etapas continuas se aplican a cada subflujo hasta que las vuelva a combinar utilizando `mergeSubstreams` .

```
val sumByKey: Flow[(String, Int), Int, NotUsed] =
  Flow[(String, Int)].
    groupBy(Int.maxValue, _._1).  //forks the flow
    map(_._2).                    //this is applied to each subflow
    fold(0)(_ + _).
    mergeSubstreams               //the subflow outputs are merged back together
```

Lea Corrientes de Akka en línea: https://riptutorial.com/es/akka/topic/7394/corrientes-de-akka

Capítulo 6: Despachadores

Examples

Despachador predeterminado

Un Akka MessageDispatcher es lo que hace que los Actores Akka "marquen", es el motor de la máquina, por así decirlo. Todas las implementaciones de MessageDispatcher también son un ExecutionContext, lo que significa que se pueden utilizar para ejecutar código arbitrario, por ejemplo, futuros.

Cada ActorSystem tendrá un despachador predeterminado que se utilizará en caso de que no se configure nada más para un Actor. El distribuidor predeterminado se puede configurar y, de forma predeterminada, es un `Dispatcher` con el `default-executor` especificado. Si se crea un ActorSystem con un `ExecutionContext` pasado, este ExecutionContext se utilizará como el ejecutor predeterminado para todos los despachadores en este ActorSystem. Si no se proporciona ExecutionContext, `akka.actor.default-dispatcher.default-executor.fallback` al ejecutor especificado en `akka.actor.default-dispatcher.default-executor.fallback` . Por defecto, este es un `fork-join-executor` , que ofrece un rendimiento excelente en la mayoría de los casos.

Configuración del despachador para un actor

Entonces, en caso de que quiera darle a su Actor un despachador diferente al predeterminado, necesita hacer dos cosas, de las cuales la primera es configurar el despachador en su `application.conf` :

```
my-dispatcher {
  # Dispatcher is the name of the event-based dispatcher
  type = Dispatcher
  # What kind of ExecutionService to use
  executor = "fork-join-executor"
  # Configuration for the fork join pool
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 2
    # Parallelism (threads) ... ceil(available processors * factor)
    parallelism-factor = 2.0
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 10
  }
  # Throughput defines the maximum number of messages to be
  # processed per actor before the thread jumps to the next actor.
  # Set to 1 for as fair as possible.
  throughput = 100
}
```

Y aquí hay otro ejemplo que usa el "thread-pool-executor":

```
my-thread-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 = 2
    # No of core threads ... ceil(available processors * factor)
    core-pool-size-factor = 2.0
    # maximum number of threads to cap factor-based number to
    core-pool-size-max = 10
  }
  # Throughput defines the maximum number of messages to be
  # processed per actor before the thread jumps to the next actor.
  # Set to 1 for as fair as possible.
  throughput = 100
}
```

Luego puede definir el despachador que usará para su actor dentro de su configuración, por ejemplo,

```
akka.actor.deployment {
  /myactor {
    dispatcher = my-dispatcher
  }
}
```

y crea este actor con el nombre especificado en la configuración:

```
import akka.actor.Props
val myActor = context.actorOf(Props[MyActor], "myactor")
```

O puedes buscar a tu despachador con:

```
import akka.actor.Props
val myActor =
  context.actorOf(Props[MyActor].withDispatcher("my-dispatcher"), "myactor1")
```

Lea Despachadores en línea: https://riptutorial.com/es/akka/topic/3228/despachadores

Capítulo 7: Hola Mundo

Examples

Akka hola mundo (Scala)

1. Añadir dependencia akka-actor (ejemplo SBT)

```
libraryDependencies += "com.typesafe.akka" % "akka-actor_2.11" % "2.4.8"
```

2. Crear clases de actores:

Actor para la salida de cadena:

```
class OutputActor extends Actor {
  override def receive: Receive = {
    case message => println(message)
  }
}
```

Actor para modificar la cadena:

```
class AppendActor(outputActor: ActorRef) extends Actor {
  override def receive: Receive = {
    case message: String =>
      val changed = s"Hello, $message!"
      outputActor ! changed

    case unknown =>
      println(s"unknown message: $unknown")
  }
}
```

3. Crear sistemas de actores y enviar mensajes.

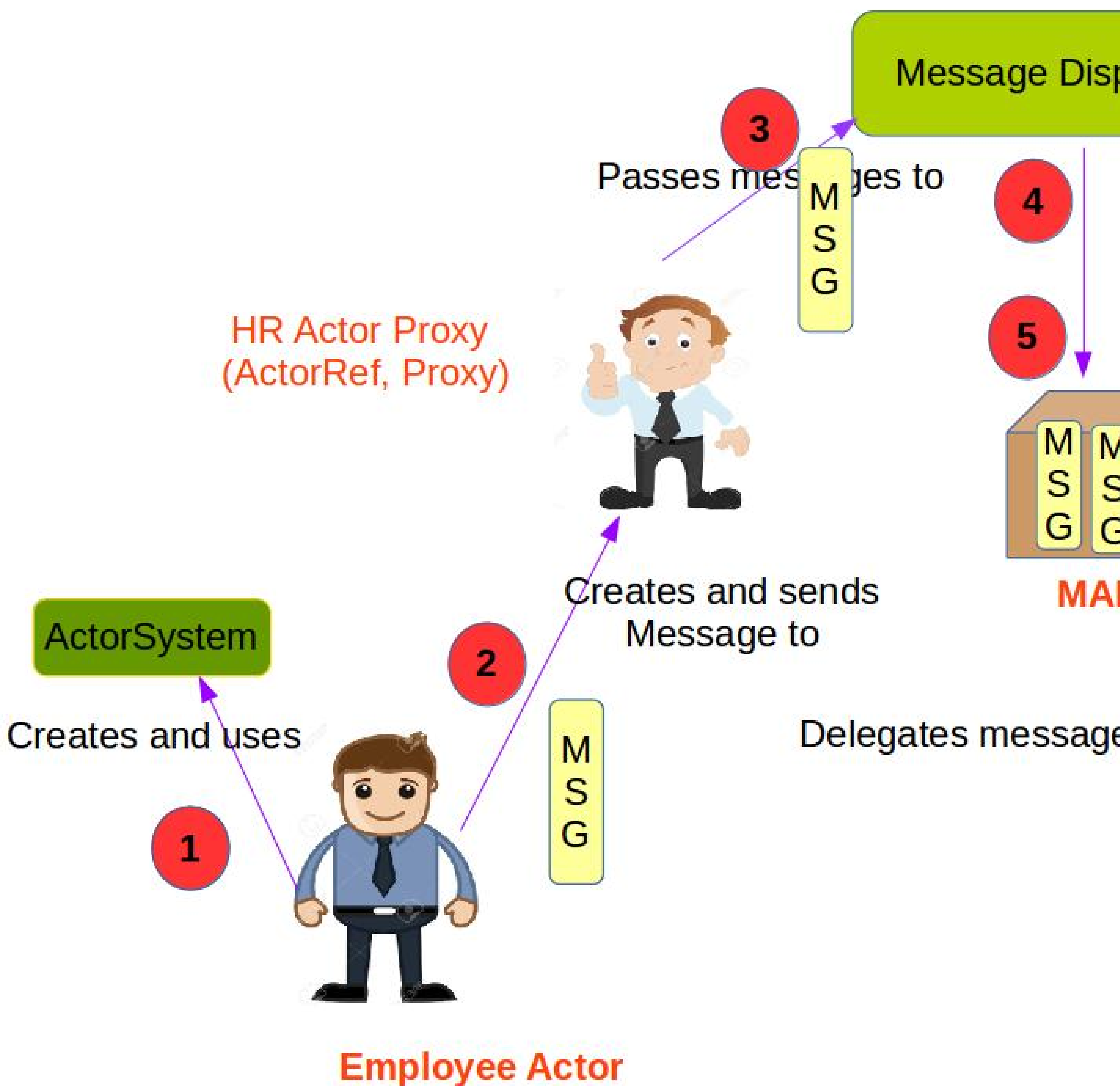
```
object HelloWorld extends App {
  val system = ActorSystem("HelloWorld")
  val outputActor = system.actorOf(Props[OutputActor], name = "output")
  val appendActor = system.actorOf(Props(classOf[AppendActor], outputActor), name =
"appender")

  appendActor ! "Akka" // send test message
  Thread.sleep(500) // wait for async evaluation
  system.terminate() // terminate actors system
}
```

Salida del programa:

> Hola, Akka!

Implementación de actor simple

Considere la posibilidad de una comunicación entre un empleado y su departamento de recursos humanos.



En términos generales, estos se explican en los siguientes seis pasos cuando se pasa un mensaje al actor:

1. Empleado crea algo llamado un `ActorSystem` .

2. Utiliza el ActorSystem para crear algo llamado como `ActorRef` . El mensaje (MSG) se envía a `ActorRef` (un proxy para HR Actor).
3. La referencia del actor pasa el mensaje a un `Message Dispatcher` .
4. El Dispatcher pone en cola el mensaje en el `MailBox` de `MailBox` del actor de `MailBox` .
5. El despachador luego coloca el `Mailbox` en un hilo (más sobre esto en la siguiente sección).
6. El `MailBox` correo `MailBox` cola un mensaje y eventualmente lo delega al método de `receive` del Actor de Recursos Humanos real.

```
    /** The Main Program consider it as a Employee Actor that is sending the requests  **/

    object EmployeeActorApp extends App{
     //Initialize the ActorSystem
      val actorSystem=ActorSystem("HrMessageingSystem")

     //construct the HR Actor Ref
      val hrActorRef=actorSystem.actorOf(Props[HrActor])

     //send a message to the HR Actor
      hrActorRef!Message

     //Let's wait for a couple of seconds before we shut down the    system
      Thread.sleep (2000)

     //Shut down the ActorSystem.
      actorSystem.shutdown()

    }


    /** The HRActor reads the message sent to it and performs action based on the message Type
**/
    class HRActor extends Actor {
       def receive  = {
            case s: String if(s.equalsIgnoreCase(“SICK”)) => println("Sick Leave applied”)
            case s: String if(s.equalsIgnoreCase(“PTO”)) => println("PTO applied “)
}
```

}

Akka Hello World (Java 8)

Agregue esta dependencia a su proyecto POM:

```
<dependency>
        <groupId>com.typesafe.akka</groupId>
        <artifactId>akka-actor_2.11</artifactId>
        <version>2.4.4</version>
</dependency>
```

Crear un actor

```
public class HelloWorldActor extends AbstractActor {

    public HelloActor() {
        receive(ReceiveBuilder
            .match(SayHello.class, this::sayHello)
            .match(SayBye.class, this::sayBye)
            .build());
    }

    private void sayHello(final SayHello message) {
        System.out.println("Hello World");
    }

    private void sayHello(final SayBye message) {
        System.out.println("Bye World");
    }

    public static Props props() {
        return Props.create(HelloWorldActor.class);
    }
}
```

Crea una prueba de Junit para el actor.

```
public class HelloActorTest {

    private ActorSystem actorSystem;

    @org.junit.Before
    public void setUp() throws Exception {
        actorSystem = ActorSystem.create();
    }

    @After
    public void tearDown() throws Exception {
        JavaTestKit.shutdownActorSystem(actorSystem);
    }

    @Test
    public void testSayHello() throws Exception {
        new JavaTestKit(actorSystem) {
            {
                ActorRef helloActorRef = actorSystem.actorOf(HelloWorldActor.props());
                helloActorRef.tell(new SayHello(), ActorRef.noSender());
                helloActorRef.tell(new SayBye(), ActorRef.noSender());
            }
        };
     }
 }
```

Lea Hola Mundo en línea: https://riptutorial.com/es/akka/topic/3283/hola-mundo

Capítulo 8: Inyectando dependencias en un actor.

Examples

Actor de primavera

Debido a la forma muy específica de instanciación del actor, inyectar dependencias en una instancia de actor no es trivial. Para intervenir en la creación de instancias de actores y permitir que Spring inyecte dependencias, se deben implementar un par de extensiones akka. El primero de ellos es un `IndirectActorProducer` :

```
import akka.actor.Actor;
import akka.actor.IndirectActorProducer;
import java.lang.reflect.Constructor;
import java.util.Arrays;
import org.springframework.beans.factory.config.AutowireCapableBeanFactory;
import org.springframework.context.ApplicationContext;

/**
 * An actor producer that lets Spring autowire dependencies into created actors.
 */
public class SpringWiredActorProducer implements IndirectActorProducer {

    private final ApplicationContext applicationContext;
    private final Class<? extends Actor> actorBeanClass;
    private final Object[] args;

    public SpringWiredActorProducer(ApplicationContext applicationContext, Class<? extends
Actor> actorBeanClass, Object... args) {
        this.applicationContext = applicationContext;
        this.actorBeanClass = actorBeanClass;
        this.args = args;
    }

    @Override
    public Actor produce() {
        Class[] argsTypes = new Class[args.length];
        for (int i = 0; i < args.length; i++) {
            if (args[i] == null) {
                argsTypes[i] = null;
            } else {
                argsTypes[i] = args[i].getClass();
            }
        }
        Actor result = null;
        try {
            if (args.length == 0) {
                result = (Actor) actorBeanClass.newInstance();
            } else {
                try {
                    result = (Actor)
actorBeanClass.getConstructor(argsTypes).newInstance(args);
                } catch (NoSuchMethodException ex) {
```

```
                    // if types of constructor don't match exactly, try to find appropriate
constructor
                    for (Constructor<?> c : actorBeanClass.getConstructors()) {
                        if (c.getParameterCount() == args.length) {
                            boolean match = true;
                            for (int i = 0; match && i < argsTypes.length; i++) {
                                if (argsTypes[i] != null) {
                                    match =
c.getParameters()[i].getType().isAssignableFrom(argsTypes[i]);
                                }
                            }
                            if (match) {
                                result = (Actor) c.newInstance(args);
                                break;
                            }
                        }
                    }
                }
            }
            if (result == null) {
                throw new RuntimeException(String.format("Cannot find appropriate constructor
for %s and types (%s)", actorBeanClass.getName(), Arrays.toString(argsTypes)));
            } else {

applicationContext.getAutowireCapableBeanFactory().autowireBeanProperties(result,
AutowireCapableBeanFactory.AUTOWIRE_BY_TYPE, true);
            }
        } catch (ReflectiveOperationException e) {
            throw new RuntimeException("Cannot instantiate an action of class " +
actorBeanClass.getName(), e);
        }
        return result;
    }

    @Override
    public Class<? extends Actor> actorClass() {
        return (Class<? extends Actor>) actorBeanClass;
    }

}
```

Este productor crea una instancia de un actor e inyecta dependencias antes de devolver la instancia de actor.

Podemos preparar `Props` para la creación de un actor utilizando `SpringWiredActorProducer` la siguiente manera:

```
Props.create(SpringWiredActorProducer.class, applicationContext, actorBeanClass, args);
```

Sin embargo, sería mejor envolver esa llamada en el siguiente bean spring:

```
import akka.actor.Extension;
import akka.actor.Props;
import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.stereotype.Component;
```

```
/**
 * An Akka Extension to inject dependencies to {@link akka.actor.Actor}s with
 * Spring.
 */
@Component
public class SpringProps implements Extension, ApplicationContextAware {

    private volatile ApplicationContext applicationContext;

    /**
     * Creates a Props for the specified actorBeanName using the
     * {@link SpringWiredActorProducer}.
     *
     * @param actorBeanClass The class of the actor bean to create Props for
     * @param args arguments of the actor's constructor
     * @return a Props that will create the named actor bean using Spring
     */
    public Props create(Class actorBeanClass, Object... args) {
        return Props.create(SpringWiredActorProducer.class, applicationContext,
actorBeanClass, args);
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws
BeansException {
        this.applicationContext = applicationContext;
    }

}
```

Puede crear automáticamente `SpringProps` cualquier `SpringProps` donde se `SpringProps` un actor (incluso en el propio actor) y crear actores con cables de la siguiente manera:

```
@Autowired
private SpringProps springProps;
//...
actorSystem.actorOf(springProps.create(ActorClass.class), actorName);
//or inside an actor
context().actorOf(springProps.create(ActorClass.class), actorName);
```

Suponiendo que `ActorClass` extienda `UntypedActor` y tenga propiedades anotadas con `@Autowired` , esas dependencias se inyectarán justo después de la `@Autowired` instancias.

Lea Inyectando dependencias en un actor. en línea: https://riptutorial.com/es/akka/topic/4717/inyectando-dependencias-en-un-actor-

Capítulo 9: Supervisión y Seguimiento en Akka

Observaciones

Referencias: akka.io/docs

Echa un vistazo a mi blog: https://blog.knoldus.com/2016/08/07/supervision-and-monitoring-in-akka/

Examples

¿Qué es la supervisión?

Describe una relación de dependencia entre los actores, el padre y la relación con el niño. El padre es único porque ha creado al niño actor, por lo que el padre es responsable de reaccionar cuando ocurren fallas en su hijo.

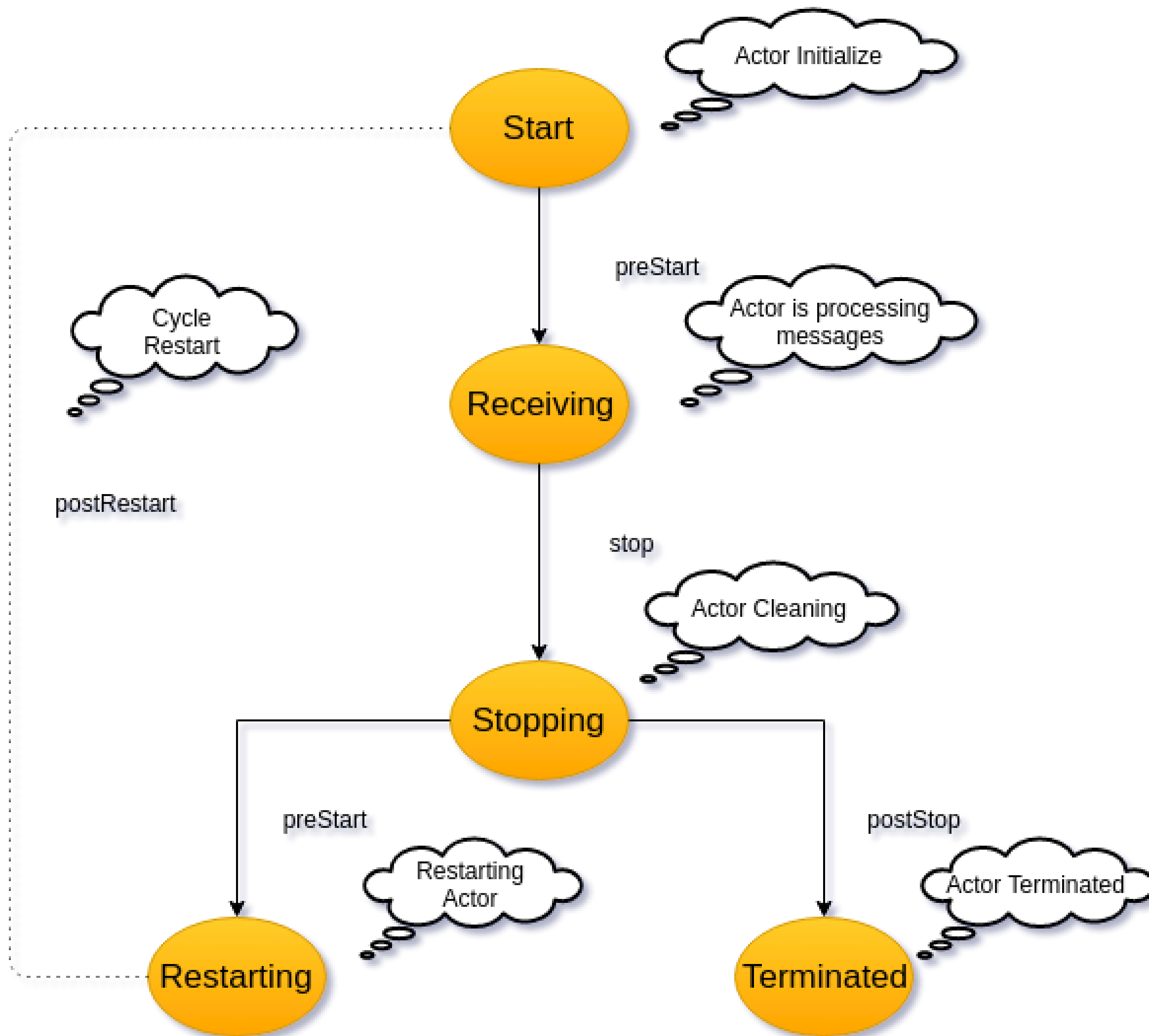
Y el padre decide qué opción necesita ser seleccionada. Cuando un padre recibe la señal de falla de su hijo, dependiendo de la naturaleza de la falla, el padre decide entre las siguientes opciones:

Curriculum vitae: El padre comienza el actor infantil manteniendo su estado interno.

Reinicio: el padre inicia al actor secundario borrando su estado interno.

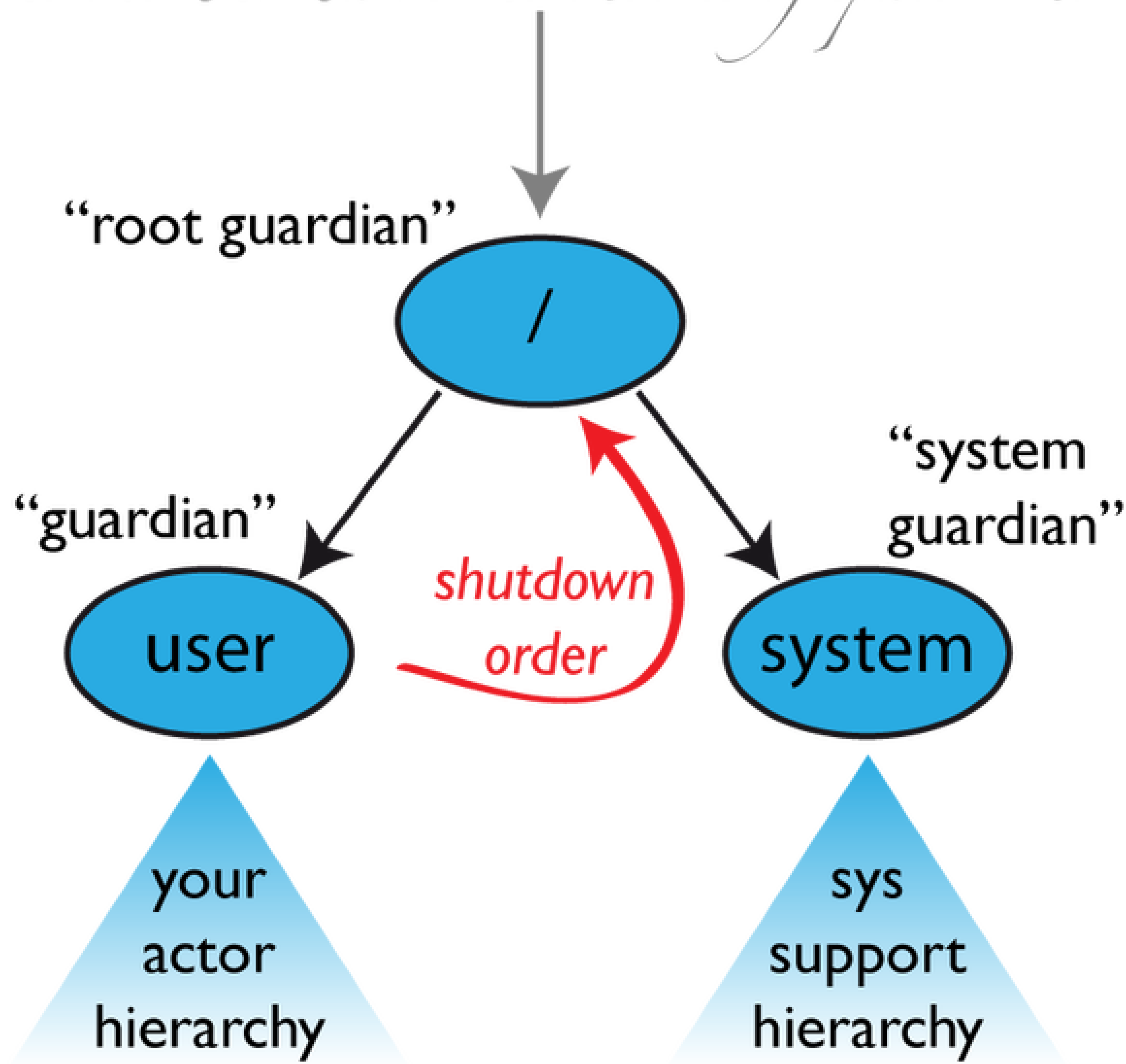
Stop: Detener al niño permanentemente.

Escalado: Aumente la falla al fallar a sí mismo y propague la falla a su padre.



Ciclo de vida akka

Siempre es importante ver una parte de la jerarquía de supervisión, que explica la opción de escalado. Cada supervisor debe cubrir con todos los posibles casos de fallo.



Sistema de actores: Fuente: doc.akka.io

/ usuario: **El usuario Guardián Actor**

El actor creado con system.actorOf () es hijo del actor tutor de usuario. Cuando el usuario guardian termina, todos los actores creados por el usuario serán terminados también. Los actores creados por el usuario de nivel superior están determinados por el actor tutor del usuario y cómo serán supervisados. Root Guardian es el supervisor del usuario guardian.

/ raíz: **The Root Guardian**

El actor guardián de la raíz es el padre de todo sistema de actores. Supervisa al usuario guardián actor y al sistema guardián actor.

Estrategias de supervisión

Hay dos tipos de estrategias de supervisión que seguimos para supervisar a cualquier actor:

1. Estrategia Uno por Uno

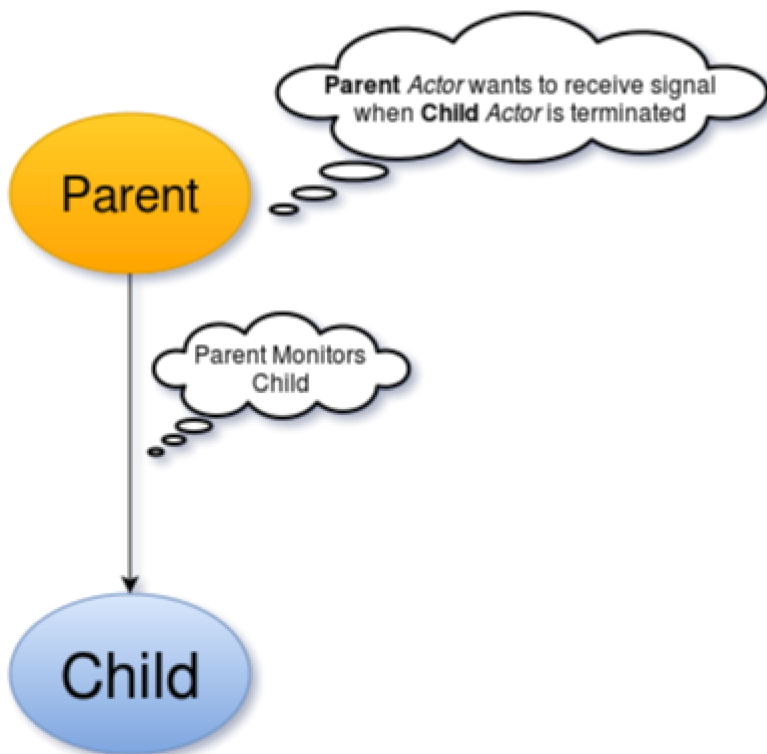
2. Estrategia única para todos

```
case object ResumeException extends Exception

case object StopException extends Exception
case object RestartException extends Exception

override val supervisorStrategy =
 OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 second){
 case ResumeException => Resume
 case RestartException => Restart
 case StopException => Stop
 case _: Exception => Escalate
 }
```

¿Qué es el monitoreo?

La monitorización del ciclo de vida en Akka se suele denominar DeathWatch.

Por lo tanto, el monitoreo se utiliza para vincular a un actor con otro para que pueda reaccionar a la terminación del otro actor, en contraste con la supervisión que reacciona ante el fracaso.



Vigilancia

El monitoreo es particularmente útil si un supervisor no puede simplemente reiniciar a sus hijos y tiene que terminarlos, por ejemplo, en caso de errores durante la inicialización del actor. En ese caso, debe monitorear a esos niños y recrearlos o programarse para reintentar esto más adelante.

Repositorio de código

Supervisión y Seguimiento

Lea Supervisión y Seguimiento en Akka en línea: https://riptutorial.com/es/akka/topic/7831/supervision-y-seguimiento-en-akka

Creditos

S. No	Capítulos	Contributors
1	Empezando con akka	12Sandy, Community, noelyahan
2	Actor DSL	Martin Seeler
3	Akka HTTP	Cyrille Corpet, Konrad 'ktoso' Malawski
4	akka-streams formas personalizadas	gilad hoch
5	Corrientes de Akka	Cyrille Corpet, hveiga
6	Despachadores	Martin Seeler
7	Hola Mundo	12Sandy, Cortwave, Fabien Benoit-Koch, Konrad 'ktoso' Malawski, tmbo
8	Inyectando dependencias en un actor.	Oleg Kurbatov
9	Supervisión y Seguimiento en Akka	Prabhat Kashyap