

Software-Architektur

— Actors —

Prof. Dr. Oliver Braun

Letzte Änderung: 11.07.2017 15:12

Actor Model

- ▶ 1973 von Carl Hewitt, Peter Bishop und Richard Steiger vorgeschlagenes Modell für nebenläufige Berechnungen
 - ▶ Carl Hewitt, Peter Bishop, Richard Steiger: A Universal Modular Actor Formalism for Artificial Intelligence. In: Proceeding IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence. 1973, S. 235–245
- ▶ Actors sind ein wesentlicher Bestandteil der Programmiersprache Erlang
 - ▶ in den 80er Jahren von der Firma Ericsson für Telefonswitche entwickelte funktionale Programmiersprache
 - ▶ die Sprache in der WhatsApp auf FreeBSD-Servern Milliarden von Messages täglich verarbeitet
- ▶ Actors sind auch in Scala und Java über Akka nutzbar

- ▶ Actors sind unabhängige, nebenläufige Einheiten
- ▶ Sie kommunizieren über Messages, die in einer Mailbox abgelegt werden
- ▶ der Actor kann sich eine Message aus der Mailbox holen und darauf reagieren:
 - ▶ er kann Nachrichten an sich oder andere (ihm bekannte) Actors senden
 - ▶ er kann neue Actors erzeugen
 - ▶ er kann das eigene Verhalten ändern

Der Actor in Akka

- ▶ ein Actor kapselt Zustand und Verhalten
- ▶ er kommuniziert ausschließlich über asynchrone Nachrichten und besitzt für den Empfang eine Mailbox

- ▶ der Actor kann nur über eine Actor-Referenz angesprochen werden
 - ▶ diese Objekte können als Parameter weiter gegeben werden
 - ▶ sie verhindern den direkten Zugriff auf den Actor und seinen Zustand

Zustand

- ▶ ein Actor hat einen Zustand den er verwaltet (Objektvariablen)
- ▶ um den Zugriff anderer Actors auf seinen Zustand zu verhindern, läuft jeder Akka-Actor in einem eigenen, abgeschirmten, leichtgewichtigen Thread
- ▶ wenn ein Actor “abstürzt” wird er von seinem Supervisor neu gestartet
 - ▶ mit initialem Zustand
 - ▶ oder Replay von persistierten Messages

- ▶ jedes mal, wenn eine Nachricht verarbeitet wird, wird Sie gegen das Verhalten des Actors gematcht
- ▶ das Verhalten ist eine Funktion/Methode des Actors
- ▶ das Verhalten kann über die Methoden `become` und `unbecome` geändert werden
- ▶ ein Restart setzt zunächst wieder das Anfangsverhalten ein

- ▶ Kommunikation findet nur asynchron statt, in dem einer ActorRef eine Message geschickt wird
- ▶ diese landet in der Mailbox des dazugehörigen Actors
- ▶ die einzige Garantie über die Reihenfolge ist, dass Nachrichten vom **gleichen** Sender zeitlich geordnet sind
- ▶ verschiedene Mailbox-Implementierungen
 - ▶ FIFO (default)
 - ▶ priorisierte Mailbox
- ▶ in Akka muss immer die nächste Nachricht verarbeitet werden
 - ▶ in manchen Actor-Framework gibt es die Möglichkeit nach der nächsten passenden Nachricht zu suchen

- ▶ jeder Actor kann weitere Actors erzeugen
 - ▶ für die er automatisch Supervisor ist
- ▶ die Liste der Kinder wird im Kontext des Actors verwaltet
- ▶ Erzeugung und Stoppen der Kinder erfolgt asynchron
- ▶ die Supervisor Strategie legt fest, wie ein Actor auf das Abstürzen seiner Kinder reagiert

- ▶ wenn ein Actor abstürzt, kann sein Supervisor auf verschiedene Arten reagieren
 - ▶ er startet ihn wieder mit dem letzten Zustand
 - ▶ er startet ihn wieder mit einem neuen initialen Zustand
 - ▶ er stoppt ihn permanent
 - ▶ er eskaliert den Fehler in dem er “sich selbst abstürzen läßt”

Supervisor-Hierarchie

- ▶ `/user`: der Guardian Actor
 - ▶ Wurzel der von der Anwendung erzeugten Actors
- ▶ `/system`: der System Guardian
 - ▶ Wurzel der *System Support Hierarchy*
- ▶ `/`: der Root Guardian
 - ▶ Supervisor von `/system` und `/user`

Supervising Strategies

`OneForOneStrategy` es wird nur der betroffene Actor neu gestartet

`AllForOneStrategy` es werden alle Kinder neu gestartet

ActorRefs und Actor-Paths

- ▶ eine Actor-Referenz referenziert einen konkreten Actor über dessen Lebenszyklus
- ▶ ein Actor-Pfad repräsentiert einen Namen hinter dem sich ein Actor verbergen kann
 - ▶ kann ohne dazugehörigen Actor erzeugt werden
 - ▶ kann nacheinander verschiedene Actors repräsentieren
- ▶ Beispiele für Pfade

```
"akka://my-sys/user/service-a/worker1"
```

```
"akka.tcp://my-sys@host.example.com:5678/user/service-b"
```

Erzeugen und Finden von Actors

- ▶ Erzeugen direkt unterhalb von /user mit `ActorSystem.actorOf`
- ▶ Erzeugen unterhalb des aktuellen Actors mit `ActorContext.actorOf`
- ▶ Zugriff über den Pfad mit `ActorSystem.actorSelection` bzw. `context.actorSelection`
 - ▶ z.B.

```
context.actorSelection("../brother")  
context.actorSelection("/user/serviceA")
```

Actors und das Java Memory Model

Zugriff durch mehrere Threads auf Shared Memory auf zwei Weisen

- ▶ wenn eine Nachricht an einen Actor gesendet wird
 - ▶ solange Nachrichten unveränderlich sind (und das sollen sie sein), gibt es keine Probleme
 - ▶ wenn Nachrichten nicht richtig unveränderlich implementiert sind, kann es vorkommen, dass der Empfänger partiell initialisierte Nachrichten sieht
- ▶ wenn ein Actor beim Verarbeiten einer Nachricht seinen Zustand verändert und kurz darauf, beim Verarbeiten einer weiteren Nachricht, wieder darauf zugreift
 - ▶ es ist nicht sichergestellt, dass ein Actor immer im selben Thread ausgeführt wird

Eine Actor-Klasse

```
import akka.actor.{Actor, ActorLogging,
                  ActorSystem, Props}

class MyActor extends Actor with ActorLogging {
  def receive = {
    case "test" => log.info("received test")
    case _      =>
      log.info("received unknown message")
  }
}
```


Props für Actors

- ▶ Props ist eine Konfigurationsklasse um Optionen für die Actor-Erzeugung zu spezifizieren
- ▶ Beispielnutzung:

```
import akka.actor.Props

val props1 = Props[MyActor]
// NO!
val props2 = Props(new ActorWithArgs("arg"))
val props3 = Props(classOf[ActorWithArgs],
                    "arg")
```

- ▶ Version 2 nur außerhalb von einem Actor nutzen, oder besser:
gar nicht

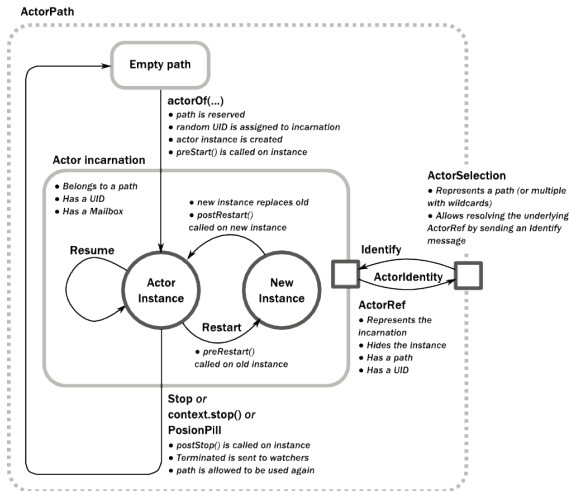
Empfohlenes Vorgehen

```
object DemoActor {
  def props(magicNumber: Int): Props =
    Props(new DemoActor(magicNumber))
}
class DemoActor(magicNumber: Int) extends Actor {
  def receive = {
    case x: Int => sender() ! (x + magicNumber)
  }
}
class SomeOtherActor extends Actor {
  // Props(new DemoActor(42)) would not be safe
  context.actorOf(DemoActor.props(42), "demo")
  // ...
}
```

Actor API

- ▶ der Actor-Trait definiert eine abstrakte Methode: `receive`
- ▶ wenn eine Message in `receive` nicht matcht, wird `unhandled` aufgerufen (nützlich z.B. zum debuggen)
- ▶ `self` = Referenz auf die ActorRef des Actors
- ▶ `sender` = Referenz auf den Sender der letzten Message
- ▶ `supervisorStrategy` ist redefinierbar
- ▶ `context`
 - ▶ Kinder erzeugen (`actorOf`)
 - ▶ Referenzen auf ActorSystem, Vater/Mutter-Actor, Kinder, ...

Actor Lifecycle



Quelle: <http://akka.io/>

Unveränderliche Nachrichten

- ▶ Messages können grundsätzlich beliebige Objekte sein
- ▶ Messages müssen unveränderbar (*immutable*) sein
 - ▶ dies kann der Scala-Compiler nicht überprüfen!
 - ▶ daher ist dies eine einzuhaltende Konvention
- ▶ veränderbarer Zustand in Messages würde zu unvorhersehbarem Verhalten führen
- ▶ primitive Datentypen und Strings sind unveränderlich
- ▶ Collections in Scala auch (wenn nicht explizit veränderliche importiert)
- ▶ mit Case-Klassen eigene Messages definieren

- ▶ zwei Arten Nachrichten zu senden:
 1. ! aka tell: *fire-and-forget*
 2. ? aka ask: gibt ein Future für die Antwort zurück

Tell: Fire-forget

- ▶ bevorzugte Nachrichtenübermittlung
- ▶ blockiert nicht
- ▶ skalierbar
- ▶ Beispiel

```
actorRef ! msg
```

- ▶ wenn das von einem Actor gesendet wurde, kann mit

```
sender() ! reply
```

einfach geantwortet werden

Ask: Send-And-Receive-Future

- ▶ der Sender sendet z.B.

```
val future = actorRef ? msg
```

- ▶ der Aufruf blockiert nicht
- ▶ der Empfänger muss eine Antwort mit

```
sender() ! reply
```

senden um das Future mit einem Wert zu füllen

Timeouts

- ▶ für ask wird ein interner Actor erzeugt, der die Antwort behandelt
- ▶ um keine Ressource Leaks zu erzeugen muss ein Timeout gesetzt werden
- ▶ wenn der Timeout abläuft, wird der interne Actor zerstört und eine AskTimeoutException geworfen
- ▶ der Timeout kann explizit oder implizit gesetzt werden

```
import scala.concurrent.duration._
import akka.pattern.ask
val future = myActor.ask("hello")(5 seconds)
```

```
import scala.concurrent.duration._
import akka.util.Timeout
import akka.pattern.ask
implicit val timeout = Timeout(5 seconds)
val future = myActor ? "hello"
```

Forward Messages

- ▶ Messages mit `tell` weiter senden, setzt den Absender auf den aktuellen Sender
- ▶ Absender erhalten mit

```
actorRef forward msg
```

Actors stoppen

- ▶ stop von system oder context aufrufen, z.B.

```
context stop otherActor
```

- ▶ an den Actor eine PoisonPill schicken

```
otherActor ! PoisonPill
```

- ▶ akka.pattern.gracefulStop gibt Future zurück →
Warten auf Stopp möglich

Verhalten ändern mit become

Ersetzen des Verhaltens auf dem *behavior stack*.

```
class HotSwapActor extends Actor {
  import context._
  def angry: Receive = {
    case "foo" => sender() ! "I am already angry?"
    case "bar" => become(happy)
  }

  def happy: Receive = {
    case "bar" => sender() ! "I am already happy :-)"
    case "foo" => become(angry)
  }

  def receive = {
    case "foo" => become(angry)
    case "bar" => become(happy)
  }
}
```

Hin und zurück mit become und unbecome

Auf den *behavior stack* pushen und wieder poppen.

```
case object Swap
class Swapper extends Actor with ActorLogging {
  import context._

  def receive = {
    case Swap =>
      log.info("Hi")
      become({
        case Swap =>
          log.info("Ho")
          unbecome()
      }, discardOld = false)
  }
}
```

Weiteres Beispiel

- ▶ siehe Livecoding-Repo
- ▶ erzeugt mit

```
activator new Akka minimal-akka-scala-seed  
und weiter angepasst
```