Reaktive Programmierung
Vorlesung 7 vom 08.05.19
Actors in Akka

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

Fahrplan

- Einführung
- Monaden und Monadentransformer
- Nebenläufigkeit: Futures and Promises
- Aktoren I: Grundlagen
- Aktoren II: Implementation
- Meta-Programmierung
- Bidirektionale Programmierung
- Reaktive Ströme I
- Reaktive Ströme II
- Funktional-Reaktive Programmierung
- Software Transactional Memory
- Eventual Consistency
- Robustheit und Entwurfsmuster
- Theorie der Nebenläufigkeit, Abschluss

Aktoren in Scala

- Eine kurze Geschichte von Akka:
  - 2006: Aktoren in der Scala Standardbücherei (Philipp Haller, `scala.actors`)
  - 2010: Akka 0.5 wird veröffentlich (Jonas Bonér)
  - 2012: Scala 2.10 erscheint ohne `scala.actors` und Akka wird Teil der Typesafe Platform
- Auf Akka aufbauend:
  - Apache Spark
  - Play! Framework
  - Akka HTTP (Früher Spray Framework)

Akka

- Akka ist ein Framework für Verteilte und Nebenläufige Anwendungen
- Akka bietet verschiedene Ansätze mit Fokus auf Aktoren
- Nachrichtengetrieben und asynchron
- Location Transparency
- Hierarchische Aktorenstruktur

Rückblick

- Aktor Systeme bestehen aus Aktoren
- Aktoren
  - haben eine Identität,
  - haben ein veränderliches Verhalten und
  - kommunizieren mit anderen Aktoren ausschließlich über unveränderliche Nachrichten.

Aktoren in Akka

```
trait Actor {
  type Receive = PartialFunction[Any, Unit]

  def receive: Receive

  implicit val context: ActorContext
  implicit final val self: ActorRef
  final def sender: ActorRef

  def preStart()
  def postStop()
  def preRestart(reason: Throwable, message: Option[Any])
  def postRestart(reason: Throwable)

  def supervisorStrategy: SupervisorStrategy
  def unhandled(message: Any)
}
```

Aktoren Erzeugen

```
object Count

class Counter extends Actor {
  var count = 0
  def receive = {
    case Count ⇒ count += 1
  }
}
```

Aktoren Erzeugen

```
object Count

class Counter extends Actor {
  var count = 0
  def receive = {
    case Count ⇒ count += 1
  }
}
```

```
val system = ActorSystem("example")
```

Aktoren Erzeugen

```
object Count

class Counter extends Actor {
  var count = 0
  def receive = {
    case Count ⇒ count += 1
  }
}
```

```
val system = ActorSystem("example")
```

Global:

```
val counter = system.actorOf(Props[Counter], "counter")
```

Aktoren Erzeugen

```
object Count

class Counter extends Actor {
  var count = 0
  def receive = {
    case Count ⇒ count += 1
  }
}
```

```
val system = ActorSystem("example")
```

Global:

```
val counter = system.actorOf(Props[Counter], "counter")
```

In Aktoren:

```
val counter = context.actorOf(Props[Counter], "counter")
```

Nachrichtenversand

```
object Counter { object Count; object Get }

class Counter extends Actor {
  var count = 0
  def receive = {
    case Counter.Count ⇒ count += 1
    case Counter.Get   ⇒ sender ! count
  }
}
```

```
val counter = actorOf(Props[Counter], "counter")
```

```
counter ! Count
```

“!” ist asynchron – Der Kontrollfluss wird sofort an den Aufrufer zurückggegeben.

Eigenschaften der Kommunikation

- Nachrichten von einer Aktor identität zu einer anderen kommen in der Reihenfolge des Versands an. (Im Aktorenmodell ist die Reihenfolge undefiniert)
- Abgesehen davon ist die Reihenfolge des Nachrichtenempfangs undefiniert.
- Nachrichten sollen unveränderlich sein. (Das kann derzeit allerdings nicht überprüft werden)

Verhalten

```
trait ActorContext {
  def become(behavior: Receive, discardOld: Boolean = true):
      Unit
  def unbecome(): Unit
  ...
}
```

```
class Counter extends Actor {
  def counter(n: Int): Receive = {
    case Counter.Count ⇒ context.become(counter(n+1))
    case Counter.Get   ⇒ sender ! n
  }
  def receive = counter(0)
}
```

Nachrichten werden sequenziell abgearbeitet.

Modellieren mit Aktoren

Aus “Principles of Reactive Programming” (Roland Kuhn):

- Imagine giving the task to a group of people, dividing it up.
- Consider the group to be of very large size.
- Start with how people with different tasks will talk with each other.
- Consider these “people” to be easily replaceable.
- Draw a diagram with how the task will be split up, including communication lines.

Beispiel

Aktorpfade

- Alle Aktoren haben eindeutige absolute Pfade. z.B. `"akka://exampleSystem/user/countService/counter1"`
- Relative Pfade ergeben sich aus der Position des Aktors in der Hierarchie. z.B. `"../counter2"`
- Aktoren können über ihre Pfade angesprochen werden

```
context.actorSelection("../sibling") ! Count
context.actorSelection("../*") ! Count // wildcard
```

- `ActorSelection` $\neq$ `ActorRef`

Location Transparency und Akka Remoting

- Aktoren in anderen Aktorsytemen auf anderen Maschinen können über absolute Pfade angesprochen werden.

```
val remoteCounter = context.actorSelection(
  "akka.tcp://otherSystem@214.116.23.9:9000/user/counter")

remoteCounter ! Count
```

- Aktorsysteme können so konfiguriert werden, dass bestimmte Aktoren in einem anderen Aktorsystem erzeugt werden

```
src/resource/application.conf:

> akka.actor.deployment {
>   /remoteCounter {
>     remote = "akka.tcp://otherSystem@127.0.0.1:2552"
>   }
> }
```

Supervision und Fehlerbehandlung in Akka

- `OneForOneStrategy` vs. `AllForOneStrategy`

```
class RootCounter extends Actor {
  override def supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 10,
                      withinTimeRange = 1 minute) {
      case _: ArithmeticException      ⇒ Resume
      case _: NullPointerException     ⇒ Restart
      case _: IllegalArgumentException ⇒ Stop
      case _: Exception                ⇒ Escalate
    }
}
```

Aktorsysteme Testen

- Um Aktorsyteme zu testen müssen wir eventuell die Regeln brechen:

```
val actorRef = TestActorRef[Counter]
val actor = actorRef.underlyingActor
```

- Oder: Integrationstests mit `TestKit`

```
"A counter" must {
  "be able to count to three" in {
    val counter = system.actorOf[Counter]
    counter ! Count
    counter ! Count
    counter ! Count
    counter ! Get
    expectMsg(3)
  }
}
```

Event-Sourcing (Akka Persistence)

- Problem: Aktoren sollen Neustarts überleben, oder sogar dynamisch migriert werden.
- Idee: Anstelle des Zustands, speichern wir alle Ereignisse.

```
class Counter extends PersistentActor {
  var count = 0
  def receiveCommand = {
    case Count ⇒
      persist(Count)(_ ⇒ count += 1)
    case Snap ⇒ saveSnapshot(count)
    case Get ⇒ sender ! count
  }
  def receiveRecover = {
    case Count ⇒ count += 1
    case SnapshotOffer(_,snapshot: Int) ⇒ count = snapshot
  }
}
```

akka–http (ehemals Spray)

- Aktoren sind ein hervorragendes Modell für **Webserver**
- akka–http ist ein **minimales** HTTP interface für Akka

```
val serverBinding = Http(system).bind(
    interface = "localhost", port = 80)

...

val requestHandler: HttpRequest ⇒ HttpResponse = {
  case HttpRequest(GET, Uri.Path("/ping"), _, _, _)  ⇒
      HttpResponse(entity = "PONG!")
  ...
}
```

- Vorteil: Vollständig in Scala implementiert, keine Altlasten wie *Jetty*

Bewertung

- Vorteile:
  - Nah am Aktorenmodell (Carl-Hewitt-approved)
  - keine Race Conditions
  - Effizient
  - Stabil und ausgereift
  - Umfangreiche Konfigurationsmöglichkeiten
- Nachteile:
  - Nah am Aktorenmodell $\Rightarrow$ receive ist untypisiert
  - Aktoren sind nicht komponierbar
  - Tests können aufwendig werden
  - Unveränderlichkeit kann in Scala nicht garantiert werden
  - Umfangreiche Konfigurationsmöglichkeiten

Zusammenfassung

- Unterschiede Akka / Aktormodell:
  - Nachrichtenordnung wird pro Sender / Receiver Paar garantiert
  - Futures sind keine Aktoren
  - `ActorRef` identifiziert einen eindeutigen Aktor
  - Die Regeln können gebrochen werden (zu Testzwecken)
- Fehlerbehandlung steht im Vordergrund
- Verteilte Aktorensystem können per Akka Remoting miteinander kommunizieren
- Mit Event-Sourcing können Zustände über Systemausfälle hinweg wiederhergestellt werden.