Reaktive Programmierung
Vorlesung 6 vom 27.05.14: Futures and Promises

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2014

Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
  - Futures and Promises
  - Reaktive Datenströme
  - Funktionale Reaktive Programmierung
  - Aktoren
  - Aktoren und Akka
- Teil III: Fortgeschrittene Konzepte

Implizite Fehlerbehandlung mit Ausnahmen

- Die Signatur einer Methode verrät nichts über mögliche Fehler

```
case class Robot(pos: Int, battery: Int) {
  def move(n: Int): Robot = {
    if (n <= 0) this
    else if (battery > 0) {
      Thread.sleep(1000);
      Robot(pos + 1, battery - 1).move(n-1)
    } else throw new LowBatteryException
} }
```

- Problem bei der Kombination:
  - Wir müssen `try` und `catch` benutzen
  - Kombination wird umständlich, und kombiniert mit Seiteneffekten unmöglich
  - Beispiel: `Robot` als veränderliche Klasse

Der Datentyp `Try`

- Macht Fehler explizit (materializes errors):

```
sealed abstract class Try[+T] {
  def flatMap[U](f: T ⇒ Try[U]): Try[U] = this match {
    case Success(x) ⇒ try f(x) catch { case NonFatal(ex) ⇒
        Failure(ex) }
    case fail: Failure ⇒ fail }
  def unit[U]= Try }

case class Success[T](x: T) extends Try[T]
case class Failure(ex: Throwable) extends Try[Nothing]

object Try {
  def apply[T](expr: ⇒ T): Try[T] =
  try Success(expr)
  catch { case NonFatal(ex) ⇒ Failure(ex) } }
```

- Ist `Try` eine Monade?

Der Datentyp `Try`

- Macht Fehler explizit (materializes errors):

```
sealed abstract class Try[+T] {
  def flatMap[U](f: T ⇒ Try[U]): Try[U] = this match {
    case Success(x) ⇒ try f(x) catch { case NonFatal(ex) ⇒
        Failure(ex) }
    case fail: Failure ⇒ fail }
  def unit[U]= Try }

case class Success[T](x: T) extends Try[T]
case class Failure(ex: Throwable) extends Try[Nothing]

object Try {
  def apply[T](expr: ⇒ T): Try[T] =
  try Success(expr)
  catch { case NonFatal(ex) ⇒ Failure(ex) } }
```

- Ist `Try` eine Monade? Nein, `e flatMap f != f e`

Explizite Fehlerbehandlung

- `Try` macht Fehler explizit:

```
case class Robot(pos: Int, battery: Int) {
  def move(n: Int): Try[Robot] = Try {
    def mv(r: Robot, n: Int): Robot = {
      if (n <= 0) this
      else if (battery > 0) {
          Thread.sleep(1000);
          mv(Robot(pos+1, battery- 1), n-1)
        } else throw new LowBatteryException }
    mv(this, n)
}   }


for { atCheckpoint ← robot.move(3)
      atGoal ← atCheckpoint.move(2) } yield atGoal
```

Explizite Fehlerbehandlung

- `Try` macht Fehler explizit:

```
case class Robot(pos: Int, battery: Int) {
  def move(n: Int): Try[Robot] = Try {
    def mv(r: Robot, n: Int): Robot = {
      if (n <= 0) this
      else if (battery > 0) {
          Thread.sleep(1000);
          mv(Robot(pos+1, battery- 1), n-1)
        } else throw new LowBatteryException }
    mv(this, n)
}   }
```

```
for { atCheckpoint ← robot.move(3)
      atGoal ← atCheckpoint.move(2) } yield atGoal
```

- Aber gibt es hier noch mehr unsichtbare Besonderheiten?

Explizite Fehlerbehandlung

- `Try` macht Fehler explizit:

```
case class Robot(pos: Int, battery: Int) {
  def move(n: Int): Try[Robot] = Try {
    def mv(r: Robot, n: Int): Robot = {
      if (n <= 0) this
      else if (battery > 0) {
          Thread.sleep(1000);
          mv(Robot(pos+1, battery- 1), n-1)
        } else throw new LowBatteryException }
    mv(this, n)
}   }

for { atCheckpoint ← robot.move(3)
      atGoal ← atCheckpoint.move(2) } yield atGoal
```

- Aber gibt es hier noch mehr unsichtbare Besonderheiten?
- Die Methode gibt das Ergebnis `n` Sekunden verzögert zurück!

Blockierende Methoden

- Was ist das Problem an Verzögerungen?

```
import scala.util.Random

val robotSwarm = List.fill(6)(Robot(0,Random.nextInt(10)))
val survivors = robotSwarm.map(_.move(5)).collect {
  case Success(survivor) ⇒ survivor }
```

- Wie lange dauert das?

Blockierende Methoden

- Was ist das Problem an Verzögerungen?

```
import scala.util.Random

val robotSwarm = List.fill(6)(Robot(0,Random.nextInt(10)))
val survivors = robotSwarm.map(_.move(5)).collect {
  case Success(survivor) ⇒ survivor }
```

- Wie lange dauert das?

- Bis zu 30*s*, weil die Methode `move` blockiert!

Typische Verzögerungen

Verzögerung	Zeit
execute typical instruction	1 ns
fetch from L1 cache memory	0.5 ns
branch misprediction	5 ns
fetch from L2 cache memory	7 ns
Mutex lock/unlock	25 ns
fetch from main memory	100 ns
send 2K bytes over 1Gbps network	20,000 ns
read 1MB sequentially from memory	250,000 ns
fetch from new disk location (seek)	8,000,000 ns
read 1MB sequentially from disk	20,000,000 ns
send packet US to Europe and back	150,000,000 ns

$1ns = 10^{-9}s$

`http://norvig.com/21-days.html#answers`

Nebenläufigkeit in Scala

- Scala hat kein sprachspezifisches Thread-Modell, sondern nutzt das Threadmodell der JVM.
- Daher sind Threads vergleichsweise teuer.
- Synchronisation auf unterster Ebene durch Monitore (`synchronized`)
- Bevorzugtes Abstraktionsmodell: Aktoren (dazu später mehr)

Futures

- Futures machen Fehler und Verzögerungen explizit!

```
case class Robot(pos: Int, battery: Int) {
  def move(n: Int): Future[Robot] = Future {
    def mv(r: Robot, n: Int): Robot = {
      if (n <= 0) this
      else if (battery > 0) {
          Thread.sleep(1000);
          mv(Robot(pos+1, battery- 1), n-1)
        } else throw new LowBatteryException }
    mv(this, n)
}   }
val robotSwarm = List.fill(6)(Robot(0,Random.nextInt(10)))
val moved = robotSwarm.map(_.move(5))
```

- Wie lange dauert das?

Futures

- Futures machen Fehler und Verzögerungen explizit!

```
case class Robot(pos: Int, battery: Int) {
  def move(n: Int): Future[Robot] = Future {
    def mv(r: Robot, n: Int): Robot = {
      if (n <= 0) this
      else if (battery > 0) {
          Thread.sleep(1000);
          mv(Robot(pos+1, battery- 1), n-1)
        } else throw new LowBatteryException }
    mv(this, n)
}   }
val robotSwarm = List.fill(6)(Robot(0,Random.nextInt(10)))
val moved = robotSwarm.map(_.move(5))
```

- Wie lange dauert das?
- 0 Sekunden! Nach spätestens 5 Sekunden sind alle Futures erfüllt:

```
moved.map(_.onComplete(println))
```

Wie funktioniert das?

- Futures haben ein einfaches Interface

```
trait Future[+T] {
  def isCompleted: Boolean
  def onComplete(f: Try[T] ⇒ Unit): Unit
  def value: Option[Try[T]]
  def map[U](f: T ⇒ U): Future[U]
  def flatMap[U](f: T ⇒ Future[U]): Future[U]
  def filter(p: T ⇒ Boolean): Future[T]
}
```

- Und können einfach erzeugt werden

```
object Future {
  def apply[T](f: ⇒ T): Future[T] = ???
}
val f = Future { someExpensiveTask }
f.onComplete(_.map(println))
```

- Siehe `Future.scala`

Promises

- Promises sind das Gegenstück zu Futures

```
trait Promise {
  def complete(result: Try[T])
  def future: Future[T]
}

object Promise {
  def apply[T]: Promise[T] = ???
}
```

- Das Future eines Promises wird durch die `complete` Methode erfüllt.
- Siehe `Promise.scala`

Execution Contexts

- Wir haben etwas verschwiegen:

```
trait Future[T] {
  def onComplete(cb: Try[T] ⇒ Unit)
    (implicit ec: ExecutionContext): Unit
}
```

- Die meisten Methoden auf Futures erwarten implizit einen `ExecutionContext`

```
trait ExecutionContext {
  def execute(runnable: Runnable): Unit
  def reportFailure(cause: Throwable): Unit
  def prepare(): ExecutionContext
}
```

- Darüber kann kontrolliert werden wo der Code ausgeführt wird.

Await und Duration

- Mit Await können Futures in den Klassischen Kontrollfluss eingebunden werden:

```
import scala.language.postfixOps
import scala.concurrent.util.duration._

val answer = for {
  _ ← sendRequest()
  a ← awaitResponse()
} yield a

Await.result(answer, atMost = 1 hour)
```

Blocking vs. Non-blocking IO

- Blockierende Futures verbrauchen einen ganzen Thread.

```
def nextRequest: Future[String] = Future {
    Stream.readLine() }
```

- Threads sind teuer! (Limit typischerweise $<$ 100000 Threads)
- Wenn möglich: nichtblockierende IO

```
def nextRequest: Future[String] = {
  val p = Promise[String]
  Stream.onNextLine(p.success)
  p.future
}
```

Nebenläufigkeit in anderen Sprachen

- Andere funktionale Sprachen (Haskell, Erlang) haben leicht-gewichtige Threads
  - Laufzeitsystem handelt Threads, Erzeugung und Synchronisation billig
- Haskell hat `MVar` (ähnlich `Futures`, aber ohne Callback)
- Erlang hat Aktoren

Zusammenfassung

- Klassifikation von Effekten:

	Einer	Viele
Synchron	`Try[T]`	`Iterable[T]`
Asynchron	`Future[T]`	`Observable[T]`

- `Try` macht Fehler explizit
- `Future` macht Verzögerung explizit
- Explizite Fehler bei Nebenläufigkeit unverzichtbar
- Nächste Vorlesung: `Observables`