Funktionale Zufallszahlen

Tom Warnke

13. Mai 2021

Übersicht

Viele Bibliotheken zur Erzeugung von (Pseudo)zufallszahlen sind im imperativen statt funktionalen Paradigma implementiert.

Wie können wir eine solche Bibliothek in ein funktionales Programm integrieren?

Imperative Zufallszahlen

Nicht pur, nicht deterministisch

```
val rng = new scala.util.Random(seed = 42)
println(rng.nextInt) // -1170105035
println(rng.nextInt) // 234785527
println(rng.nextInt) // -1360544799
```

Der Ausdruck `rng.nextInt` ist quasi vom Typ `() => Int`.

Jeder Aufruf verursacht Seiteneffekte und liefert ein anderes Ergebnis.

Funktionale Zufallszahlen

Pur, deterministisch

```
trait RNG {
  def nextInt: (RNG, Int)
}
val rng: RNG = mkRNG(seed = 42)
println(rng.nextInt) // (RNG(seed = 13), -1170105035)
println(rng.nextInt) // (RNG(seed = 13), -1170105035)
println(rng.nextInt) // (RNG(seed = 13), -1170105035)
```

Der Ausdruck `rng.nextInt` ist quasi vom Typ `RNG => (RNG, Int)`.

Der implizite Zustand des Zufallszahlengenerators ist jetzt explizit.

Problem

und Lösungsideen

(Imperative) Bibliotheken geben uns ein `() => Int`.

Für unser funktionales Programm brauchen wir ein `RNG => (RNG, Int)`.

Wie kommen wir vom einen zum anderen?

- RNG-Quellcode anpassen
- Zustand speichern und rekonstruieren
- Zufallszahlen in Liste sammeln
- Zufallszahlen in Stream sammeln

Lösungsidee 1

RNG-Quellcode anpassen/FP-RNG implementieren

Mathemagischen Code aus dem imperativem Stil in funktionalen Code übersetzen

```
case class Simple(seed: Long) extends RNG {
  def nextInt: (Int, RNG) = {
    val newSeed = (seed * 0x5DEECE66DL + 0xBL) &
      0xFFFFFFFFFFFFL
    val nextRNG = Simple(newSeed)
    val n = (newSeed >>> 16).toInt
    (n, nextRNG)
  }
}
```

Problem: (komplizierter) Code wird dupliziert, Lizenzrecht

Lösungsidee 2

Zustand speichern und rekonstruieren

Z.B. für Apache Commons:

```
case class MersenneTwister(
        private val state: RandomProviderState
    ) extends RNG {

  override def nextInt: (RNG, Int) = {
    val rng = RandomSource.create(RandomSource.MT_64)
    rng.restoreState(state)
    val i = rng.nextInt()
    val s = rng.saveState()
    (MersenneTwister(s), i)
  }
}
```

Problem: umständlich, ineffizient (?)

Lösungsidee 3

Zufallszahlen in Liste sammeln

```
def mkRNG(seed: Long, count: Int): ListRNG = {
  val random = new scala.util.Random(seed)
  val numbers = List.fill(count)(random.nextInt())
  new ListRNG(numbers)
}

class ListRNG(numbers: List[Int]) extends RNG {
  override def nextInt: (RNG, Int) =
    (new ListRNG(numbers.tail), numbers.head)
}
```

Problem: Anzahl benötigter Zufallszahlen muss vorher bekannt sein

Lösungsidee 4

Zufallszahlen in Stream sammeln

```
def mkRNG(seed: Long): StreamRNG = {
  val random = new scala.util.Random(seed)
  val numbers = LazyList.continually(random.nextInt)
  new StreamRNG(numbers)
}

class StreamRNG(numbers: LazyList[Int]) extends RNG {
  override def nextInt: (RNG, Int) =
    (new StreamRNG(numbers.tail), numbers.head)
}
```

Zusammenfassung

- Lazy Evaluation hilft, imperativen Code in einer funktionalen “on-Demand”-Datenstruktur zu verstecken.
- Für den Nutzer des Zufallszahlengenerators ist nicht ersichtlich, wie die Zufallszahlen erzeugt werden.
- Alte, nicht mehr benötigte Zufallszahlen werden vom garbage collector aufgeräumt.