Reactive Streams

Software-Architektur

Prof. Dr. Oliver Braun

Letzte Änderung: 27.01.2020 21:31

Streams

- große Datenmengen, Live-Daten, etc. können nicht als Ganzes verarbeitet werden
- Datenteile müssen während er “Anlieferung” sofort verarbeitet werden
- dies geschieht in sog. *Streams*
- Streams müssen in asynchronen Systemen verarbeitet werden
    - sonst ist die App nur damit beschäftigt den Streams zu verarbeiten
- Hauptproblem:
    - schnelle Datenquelle “überschwemmt” langsamer Datensenke

Reactive Streams

- Reactive Streams steuern den Austausch von Datenströmen und
- stellen sicher, dass die Empfangsseite keine große Menge an Daten puffern muss
- d.h. “Gegendruck” ist ein integraler Bestandteil dieses Modells um Queues die zwischen den Threads vermitteln zu begrenzen
- wichtig dabei ist, dass das gesamte System ein asynchrones und nicht blockierendes Verhalten vorweist

Reactive Streams Initiative

- http://www.reactive-streams.org/
- Ziel: Minimale Menge von Interfaces, Methoden und Protokolle für asynchrone Datenströme mit nicht blockierendem Gegendruck
- Arbeitsgruppen
  - Basic Semantics
  - JVM Interfaces
    - Version 1.0.0 am 30.04.2015 fertig spezifiziert
  - JavaScript Interfaces
  - Netzwerkprotokolle
- übrigens auch andere Ansätze
  - Microsofts Rx (Reactive Extensions)

Einschub: Publisher-Subscriber-Pattern

- ein Messaging Pattern für die Software-Architektur
- der Sender (*publisher*) sendet nicht direkt an die Empfänger (*subscriber*)
- Messages werden veröffentlicht und können in Klassen eingeteilt werden
- Subscriber können dann die Klassen abonnieren

Reactive Streams auf der JVM

- README.md
- API Komponenten
  - `Publisher`
  - `Subscriber`
  - `Subscription`
  - `Processor`

API

- ein `Publisher`
  - bietet eine potentiell unendliche Anzahl aufeinanderfolgender Elemente
  - veröffentlicht diese gem. der Nachfrage der Subscriber
- als Antwort auf `Publisher.subscribe(Subscriber)` stehen dem Subscriber die Methoden in der folgenden Sequenz zur Verfügung

  ```
  onSubscribe onNext* (onError | onComplete)?
  ```

Publisher

```java
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

- die *Signalisierungen* (`onSubscribe`, ...) müssen sequentiell erfolgen
- ein `Publisher` sendet höchstens soviele `onNext`-Signale wie vom `Subscriber` engefordert wurden

Subscriber

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

Subscription

```java
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

- `request(n)` fordert die nächsten `n` `onNext`-Signale an
  - kann aus `onNext` und `onSubscribe` synchron aufgerufen werden

Processor

```
public interface Processor<T, R>
    extends Subscriber<T>, Publisher<R> {
}
```

- repräsentiert eine (Zwischen-)Verarbeitungsphase
- sie ist beides, `Subscriber` und `Publisher`

Beispiele

https://github.com/reactive-streams/reactive-streams-jvm/tree/v1.0.0/examples

Akka Streams

- Akka Streams and HTTP Module (22.05.2015: 1.0-RC3)
- nutzen intern Reactive Streams
- typisches Beispiel: Konsumieren und filtern von Tweets

Beispiel: Datenmodell

```
final case class Author(handle: String)

final case class Hashtag(name: String)

final case class Tweet(
    author: Author,
    timestamp: Long,
    body: String) {
  def hashtags: Set[Hashtag] =
    body.split(" ").collect {
        case t if t.startsWith("#") ⇒
            Hashtag(t) }.toSet
}

val akka = Hashtag("#akka")
```

Einfache Streams

- der `materializer` ist für die Streams

```
implicit val system =
    ActorSystem("reactive-tweets")
implicit val materializer =
    ActorFlowMaterializer()
```

- eine Quelle für die Tweets

```
val tweets: Source[Tweet, Unit]
```

- eine Quelle für die Autoren

```
val authors: Source[Author, Unit] =
    tweets
        .filter(_.hashtags.contains(akka))
        .map(_.author)
```

Aus einem Element mehrere machen

```
val hashtags: Source[Hashtag, Unit] =
    tweets.mapConcat(_.hashtags.toList)
```

Einen Stream broadcasten

```
val writeAuthors: Sink[Author, Unit] = ???
val writeHashtags: Sink[Hashtag, Unit] = ???
val g = FlowGraph.closed() { implicit b ⇒
  import FlowGraph.Implicits._
  val bcast = b.add(Broadcast[Tweet](2))
  tweets ⇝ bcast.in
  bcast.out(0) ⇝
    Flow[Tweet].map(_.author) ⇝
    writeAuthors
  bcast.out(1) ⇝
    Flow[Tweet].mapConcat(_.hashtags.toList) ⇝
    writeHashtags
}
g.run()
```

Gegendruck

- die aktuellen Tweets mit einem Puffer von 10 Elementen

```
tweets
  .buffer(10, OverflowStrategy.dropHead)
  .map(slowComputation)
  .runWith(Sink.ignore)
```

Elemente zählen

- obwohl so ein Strom potentiell unendlich ist, kann es sinnvoll sein, für einen endlichen Teil die Elemente zu zählen oder ähnliches
- Beispiel

```
val sumSink: Sink[Int, Future[Int]] =
  Sink.fold[Int, Int](0)(_ + _)
val counter: RunnableFlow[Future[Int]] =
  tweets.map(t ⇒ 1).toMat(sumSink)(Keep.right)
val sum: Future[Int] = counter.run()
sum.foreach(c ⇒ println(s"Total tweets processed: $c"))
```

  - `sumSink` ist eine `FoldSink`
  - mit dem `counter` wird ein Element gezählt und an die `FoldSink` weitergegeben
    - `Keep.right` besagt nur den *materialisierten* Teil weiter geben (2.

Reactive Streams in Play

- in Play werden sog. Iteratees, Enumerators und Enumeratees verwendet
- wir sprechen von dem **iteratees API**
- Fokus auf:
  - Datenströme erzeugen, konsumieren und transformieren
  - verschiedene Datenquellen einheitlich behandeln (Dateien auf der Festplatte, Websockets, File Upload, ...)
  - Komponierbar
  - non-blocking, reactive und kontrollierbar

Iteratees

- ein Iteratee konsumiert einen Datenstrom und berechnet einen Wert, z.B.

```
Iteratee[String,Int]
```

  konsumiert `String`s und produziert einen `Int`

- hat drei mögliche Zustände: `Done`, `Cont` und `Error`
- und eine Methode

```
def fold[B](folder: Step[E, A] ⇒ Future[B]): Future[B]
```

- `Step` hat auch drei Zustände

```
object Step {
  case class Done[+A, E](a: A, remaining: Input[E])
    extends Step[E, A]
  case class Cont[E, +A](k: Input[E] ⇒ Iteratee[E, A])
```

Iteratees: Beispiel 1

- ein Iteratee im Zustand **Done** erzeugt eine **1** und gibt **Empty** als Rest des letzten Inputs zurück

```
val doneIteratee = new Iteratee[String,Int] {
  def fold[B](
    folder: Step[String,Int] ⇒ Future[B])(
    implicit ec: ExecutionContext) : Future[B] =
      folder(Step.Done(1, Input.Empty))
}
```

oder kürzer

```
val doneIteratee = Done[String,Int](1, Input.Empty)
```

- um den Iteratee zu nutzen, benötigen wir noch die **folder**-Funktion

```
def folder(step: Step[String,Int]):Future[Option[Int]] =
  step match {
```

Iteratees: Beispiel 2

- Iteratee der einen Input konsumiert

```
val consumeOneInputAndEventuallyReturnIt =
  new Iteratee[String,Int] {
    def fold[B](
        folder: Step[String,Int] ⇒ Future[B])(
        implicit ec: ExecutionContext): Future[B] = {
      folder(Step.Cont {
        //Assuming 0 for default value
        case Input.EOF ⇒ Done(0, Input.EOF)
        case Input.Empty ⇒ this
        case Input.El(e) ⇒ Done(e.toInt,Input.EOF)
      })
    }
  }
```

Iteratee.fold

- Funktion um einen Iteratee zu bauen

```
def fold[E, A](state: A)(f: (A, E) ⇒ A): Iteratee[E, A]
```

- Beispiele:

```
val inputLength: Iteratee[Array[Byte],Int] = {
  Iteratee.fold[Array[Byte],Int](0) {
    (length, bytes) ⇒ length + bytes.size
  }
}
```

```
val consume: Iteratee[String,String] = {
  Iteratee.fold[String,String]("") {
    (result, chunk) ⇒ result ++ chunk
  }
}
```

Enumerators

- dienen als Quelle und übergibt eine Eingabe an einen Iteratee

```
trait Enumerator[E] {
  /**
   * Apply this Enumerator to an Iteratee
   */
  def apply[A](i: Iteratee[E, A]):
    Future[Iteratee[E, A]]
}
```

Enumerators: Beispiel

- ein String-Enumerator

```
val enumerateUsers: Enumerator[String] = {
  Enumerator("Guillaume", "Sadek", "Peter", "Erwan")
}
```

- auf einen Iteratee anwenden:

```
val consume = Iteratee.consume[String]()
val newIteratee: Future[Iteratee[String,String]] =
  enumerateUsers(consume)
```

- und nutzen

```
// We use flatMap since newIteratee is a promise,
// and run itself return a promise
val eventuallyResult: Future[String] =
  newIteratee.flatMap(i ⇒ i.run)
```

Enumerators komponieren

- Enumerators können auch komponiert werden, z.B.

```
val colors = Enumerator("Red","Blue","Green")
val moreColors = Enumerator("Grey","Orange","Yellow")
val combinedEnumerator = colors.andThen(moreColors)
val eventuallyIteratee = combinedEnumerator(consume)
```

- oder per Operatoren:

```
val eventuallyIteratee = {
  Enumerator("Red","Blue","Green") >>>
  Enumerator("Grey","Orange","Yellow") |>>
  consume
}
```

Enumerators für Dateien etc.

- selbstverständlich gibt es bereits Funktionen um Enumerators aus vorhandenen Datenquellen zu erzeugen
- Beispiel für einen Enumerator der den Inhalt einer Datei repräsentiert:

```
val fileEnumerator: Enumerator[Array[Byte]] = {
  Enumerator.fromFile(new File("path/to/some/file"))
}
```

- mit `Enumerator.fromStream` kann aus `java.io.InputStream` ein Enumerator erzeugt werden
- oder mit `generateM` ein Enumerator der alle 100ms einen Datewert erzeugt:

```
Enumerator.generateM {
  Promise.timeout(Some(new Date), 100 milliseconds)
}
```

Enumeratees

- mit Enumeratees können Streams angepasst und transformiert werden, z.B. mit `Enumeratee.map`
- Beispiel: wir haben einen Iteratee und einen Enumerator:

```
val sum: Iteratee[Int,Int] =
    Iteratee.fold[Int,Int](0){ (s,e) ⇒ s + e }
val strings: Enumerator[String] =
    Enumerator("1","2","3","4")
```

- mit Hilfe des Enumeratees

```
val toInt: Enumeratee[String,Int] =
    Enumeratee.map[String]{ s ⇒ s.toInt }
```

- können wir die beiden dann zusammenschalten:

```
strings |>> toInt &>> sum
```

Und zum Schluß noch

... der Hinweis auf das Activator-Template Play Iteratees

How to use Play Iteratees to build a custom body parser, using as an example an mp3 file metadata parser.