Stateful operations and materialized values

Programming Reactive Systems

Konrad Malawski, Julien Richard-Foy

Stateful operations

While stream processing often can “get away with” being stateless, such that each operator only needs to operate on the current element.

Examples of such stateless operators include `map`, `filter` and other classics.

Sometimes we may need to implement an operator that depends on some state that it constructed from all previously processed elements.

statefulMapConcat

```
source
  .statefulMapConcat { () =>
    // safe to keep state here
    var state = ???
    // end of place to keep mutable state

    // invoked on every element:
    element => {
      val elementsToEmit = List[String]()

      // return a List of elements that shall be emitted
      elementsToEmit
    }

  }
```

Example 1: implement `zipWithIndex`

- `zipWithIndex` produces elements paired with their index
- It has to keep track of how many elements have been previously produced

```
def zipWithIndex[A]: Flow[A, (A, Int), _] =
  Flow[A].statefulMapConcat { () =>
    var i = -1
    element => {
      i += 1
      (element -> i) :: Nil
    }
  }
```

Example 2: filter above current average

Imagine you are implementing a ranking system, and from all incoming ratings maintain a “current average rating”.

Based on that value, you want to filter elements that are *above* the current average score.

Example 2: filter above current average (implementation)

```
val aboveAverage: Flow[Rating, Rating, _] =
  Flow[Rating].statefulMapConcat { () =>
    var sum = 0 // current sum of ratings
    var n = 0   // number of summed ratings
    rating => {
      sum += rating.value
      n += 1
      val average = sum / n
      if (rating.value >= average) rating :: Nil
      else Nil
    }
  }
```

Materialized values

So far we have mostly ignored materialized values, however they are crucial in building non-trivial streams.

Simplest example:

```
val sink: Sink[Int, Future[Int]] = Sink.head
```

`sink` describes a process that consumes at most one `Int` element. When this process is executed, it returns a `Future[Int]`, which is completed when the element is received:

```
val eventuallyResult: Future[Int] = someSource.runWith(sink)
```

Materialized values: 2nd example

Here is an example of use of a value materialized by a `Source`:

```
// First, describe a source
val source: Source[Int, Promise[Option[Int]]] = Source.maybe[Int]
// Second, materialize it
val promise: Promise[Option[Int]] = source.to(Sink.ignore).run()
// Third, emit exactly one element
promise.complete(Some(42))
```

Keeping materialized values

When we combine two stages together (for instance, a `Source` and a `Sink`), by default only one of their materialized values is kept:

```
// keeps the materialized value of the source
source.to(sink).run(): Promise[Option[Int]]
// keeps the materialized value of the sink
source.runWith(sink): Future[Int]
```

We can get more fine-grained control on what to do with materialized values by using `toMat` instead of `to`:

```
source
  .toMat(sink)((sourceMat, sinkMat) => (sourceMat, sinkMat))
  .run(): (Promise[Option[Int]], Future[Int])
```

Keeping materialized values (2)

```
trait Source[+O, +M] {
  def toMat[M2, M3](s: Sink[O, M2])(op: (M, M2) => M3): RunnableGraph[M3]
}
```

Convenient shorthands are provided by the `Keep` helper object:

```
source.toMat(sink)(Keep.both):  RunnableGraph[(Promise[Option[Int]], Future[Int])]
source.toMat(sink)(Keep.right): RunnableGraph[Future[Int]]
source.toMat(sink)(Keep.left):  RunnableGraph[Promise[Option[Int]]]
source.toMat(sink)(Keep.none):  RunnableGraph[NotUsed]
```

Materializing “control” objects

Materialized values can be used to *control* the execution of a stream.

Consider for instance the following stream that emits values each second when it is executed:

```
val ticks: Source[Int, Cancellable] = Source.tick(1.second, 1.second, 1)
val cancellable = ticks.to(Sink.ignore).run()

// the stream will run forever...
// ... unless we explicitly cancel it
cancellable.cancel()
```

Summary

In this video we learnt:

- how to build *stateful operations* using `statefulMapConcat`
- why *materialized values* are useful and powerful
- how to use the `*Mat` versions of APIs to "keep" only the materialized values we care about