グラフを知って理解するAkka Streams

2017-09-09 Scala Kansai Summit　@kamijin_fanta

自己紹介

About me

Twitter	@kamijin_fanta
業務範囲	フロントエンド・バックエンド・インフラ
言語	Scala, C#, JS, TS, Python, Golang

AkkaStreamsを使う上での基礎知識

Basic knowledge

AkkaStreamsとは

Introduction

Akka の 1 つのプロダクト。Akka-Actor とは考え方が違う。

特徴

- ストリーム処理（低遅延・無限のデータ処理）
- バックプレッシャーによるフロー制御
- 型安全
- グラフを用いたシンプルな処理記述

用語
Glossary

Element

AkkaStreams での処理の単位

Producer

生産者　データを送信する側

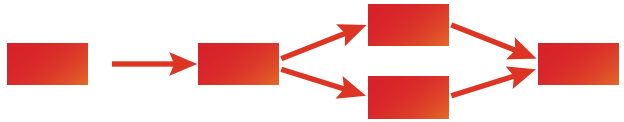
Consumer

消費者　データを受信する側

Back-pressure

フロー制御の一種　多くのエレメントを輻輳無く配信

Graph



ストリームの処理を行う部品、またそれを合わせたもの

主なグラフの構成要素

Components

下流からの要求に応じ、エレメントの出力を行う。

上流からのエレメントを受け取り処理を行い、下流に流す。

Flow は map/filter 等から簡単に実装できる。

上流からのエレメントを受け取り、何らかの処理を行う。

グラフの入口・出口

Outlet / Inlet

部品の入口を inlet・出口を outlet と呼び、合わせてポートと呼ぶ。

入出力を行うエレメントの型が指定され、型が一致した outlet/inlet が接続できる。

outlet が一つ

outlet,inlet 各 1 つ

inlet が 1 つ

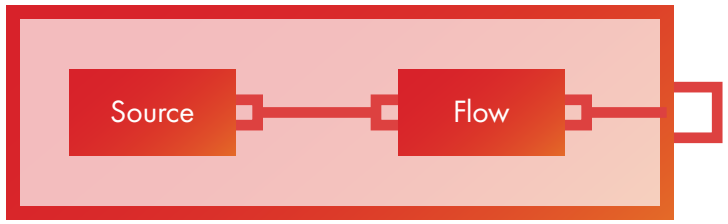
複数の入力 / 出力ポートを持つ Fan-in/Fan-out も存在する。

部品を組み合わせて作るグラフ

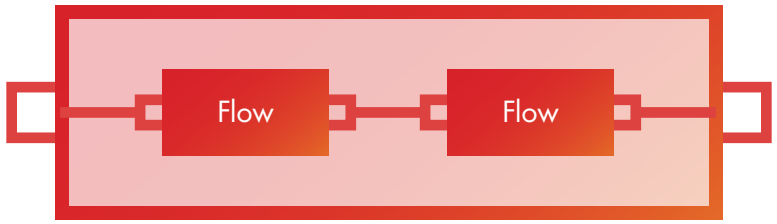
Combining parts

Source/Flow/Sink を組み合わせると、大きな Source/Flow/Sink になる。

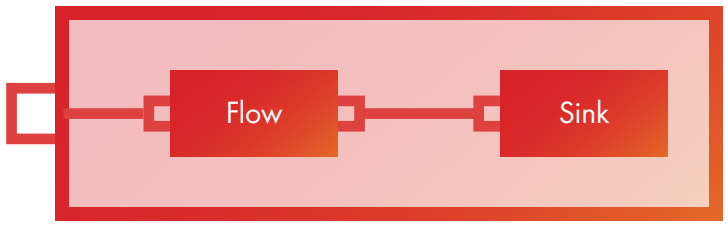
空きのポートがなくなれば、動作可能なグラフである RunnableGraph になる。



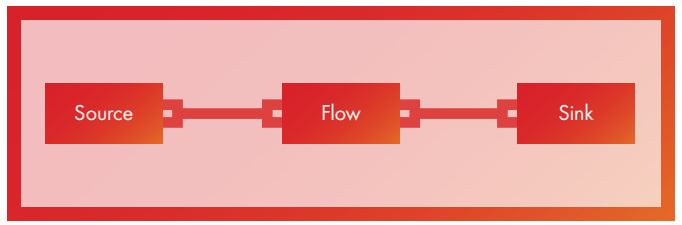
Nested Source



Nested Flow



Nested Sink



RunnableGraph

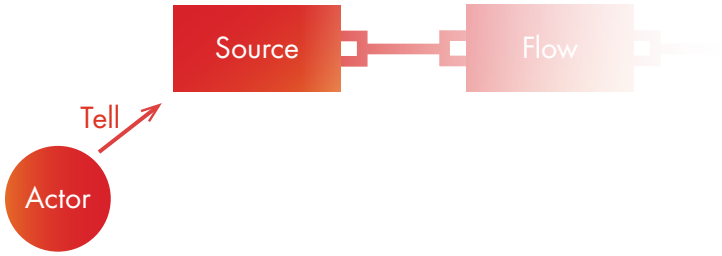
閉じたグラフへの通信手段

Materialized values

グラフの外へ変数の受渡しを行う仕組みを Materialized Values と呼ぶ。

代表的な Materialized Values の利用例

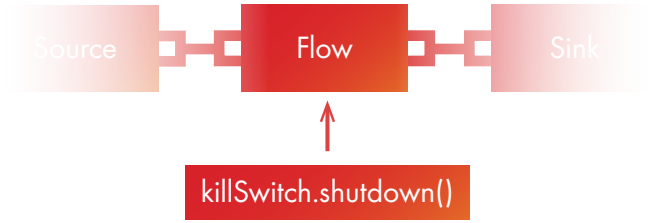
Source.actorRef



ActorRef を返す

Actor から送信した値が Source の値になる

KillSwitches.single



killSwitch オブジェクトを返す

外部からストリームを終了できる

Sink.fold

Future を返す

ストリーム終了後に、集計の結果を渡す

Materialized Valuesの取捨選択

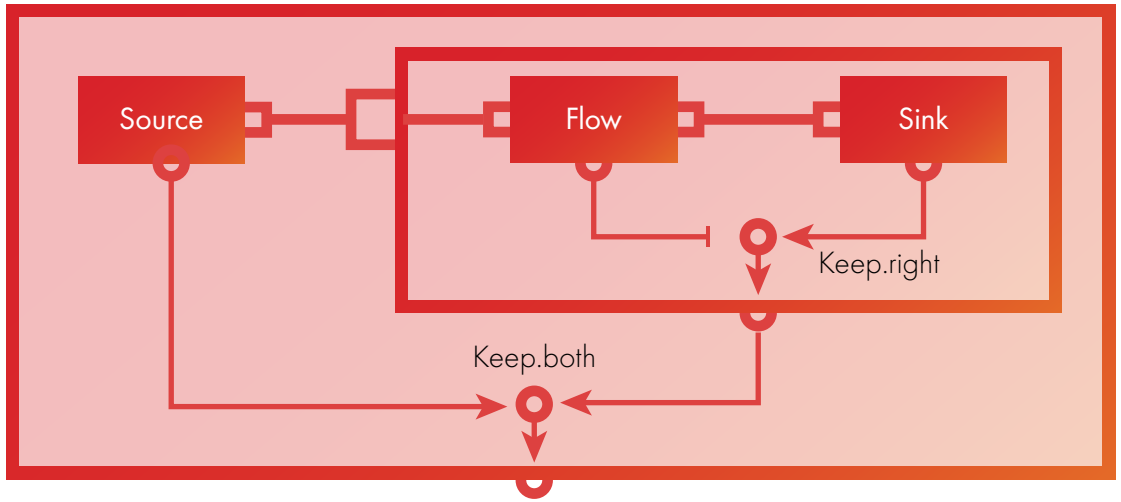
Combiner

すべてのグラフに Materialized Values は有るが、使用しないものも多い。

Combiner(混合器) を使用して、Materialized Values の取捨選択を行う。

Hint:

1. Mat を使用しない場合、NotUsed として型が指定されている
2. Right(上流側) ・ Left(下流) ・ Both(両方)
3. Both を選んだ際、タプルで両方の値が返される



AkkaStreamsのコード例

Examples

実装例1 - 基本

Basic Example

```scala
val src = Source(1 to 5)
val doubleFlow = Flow[Int].map(x => x * 2)
val printSink = Sink.foreach(println)

val runnableGraph = src via doubleFlow to printSink

runnableGraph.run()
```

examples/src/main/scala/Basic.scala

1~5 の Source を作り、Int を 2 倍する Flow を経由し、println する Sink に行く

処理の分離が出来ていて綺麗・再利用しやすい

実装例2 - Materialized values

Materialized values Examples

```
val src: Source[Int, NotUsed] = Source(1 to 20)                          ←1-20 の数列
val primeFilterFlow: Flow[Int, Int, NotUsed] = Flow[Int].filter {
  case i if i <= 1 => false
  case i if i == 2  => true                                              ←素数だけ通すフィルタ
  case i => !(2 until i).exists(x => i % x == 0)
}
val collectIntSink: Sink[Int, Future[Set[Int]]] =                        ←来た値を集計して Set に詰める
  Sink.fold(Set[Int]()){ case (a, b) => a + b }

val runnableGraph: RunnableGraph[Future[Set[Int]]] =
  (src via primeFilterFlow toMat collectIntSink)(Keep.right)             ←集計値を受け取るため Mat 使用

val future: Future[Set[Int]] = runnableGraph.run()
future.onComplete { x =>
    println(x)
    system.terminate()
}
```

examples/src/main/scala/Mat.scala

Graph の外と通信するために Materialized values を使用している

Flowのテスト例

Flow Tests

```
val primeFilterFlow: Flow[Int, Int, NotUsed] = Flow[Int].filter {
  case i if i <= 1 => false
  case i if i.==(2)  => true
  case i => !(2 until i).exists(x => i % x == 0)
}

val testSrc = Source(1 to 20)
val probe = testSrc
  .via(primeFilterFlow)
  .toMat(TestSink.probe[Int])(Keep.right)
  .run()

probe.request(8)
probe.expectNext(2, 3, 5, 7, 11, 13, 17, 19)
```

←Probe を受け取るために Mat を使用

examples/src/test/scala/FlowTestSpec.scala

各パーツの実装が単純なのでテストが容易　testkit も用意されている

提供されているSource/Sink

Libraries

Default

Source.tick / FileIO / TCP, TLS

デフォルトで多くの IO・コレクションから Source/Sink を作れる

alpakka

AMQP / MQTT / AWS / Azure / HBase

デフォルトで多くの IO・コレクションから Source/Sink を作れる

Akka Streams Kafka

Producer / Consumer

Akka Streams の分散を行いたい場合に使うことも多い

https://github.com/akka/reactive-kafka

Reactive Streams

Slick / elasticsearch

Java9 で導入予定のストリーム API

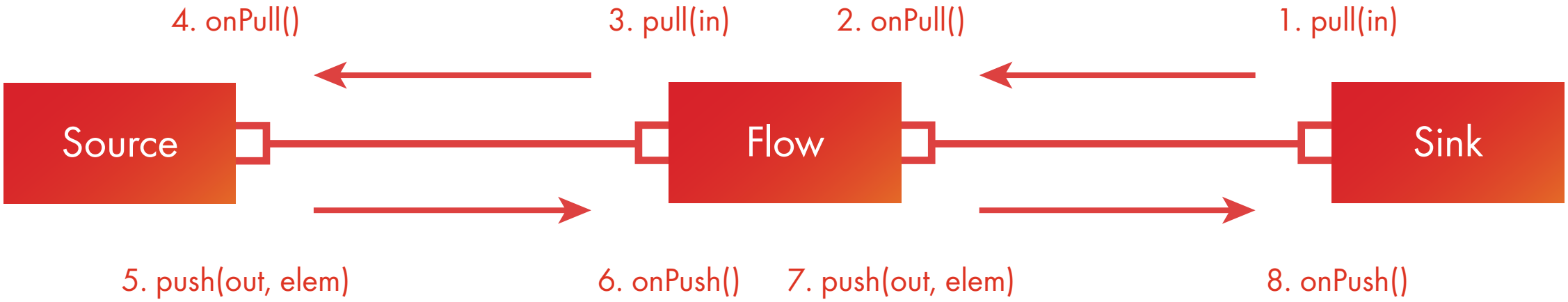
Database、検索エンジン等への接続を行いやすい

AkkaStreamsを深く知る

Details

基本的な動作の流れ

Element Flows



1 Sink が上流に Pull(エレメントの要求) する

2,3 Flow が Pull を受け取り上流に Pull を受け流す

4,5 Source が Pull を受け取り、エレメントを Push する

6,7 Flow がエレメントの Push を受け取り、必要な処理を行い下流に Push する

8 Sink がエレメントの Push を受け取り、必要な処理を行い 1 に戻る

下流から始まる要求でバックプレッシャーを実現

Mapをグラフとして実装

Implement Map

```
case class MapGraph[In, Out](val fn: In => Out) extends GraphStage[FlowShape[In, Out]] {
  private val in = Inlet[In]("in")
  private val out = Outlet[Out]("out")

  override val shape = FlowShape(in, out)

  override def createLogic(attr: Attributes) = new GraphStageLogic(shape) {
    setHandler(in, new InHandler {
      override def onPush(): Unit = push(out, fn(grab(in)))
    })
    setHandler(out, new OutHandler {
      override def onPull(): Unit = pull(in)
    })
  }
}
```

←inlet/outlet ポートを宣言

←Flow を宣言

←上流から値を grab し、
fn を適用し、下流に push する

←下流からの pull を上流に受け流す

examples/src/main/scala/MapGraph.scala

まとめ

Summary

バックプレッシャーによるフロー制御が行える

小さな部品を組み合わせてグラフを作る　→　見通しが良い , テストが容易

Source/Flow/Sink の部品を 1 から作るのも難しくない

勉強する上で役に立つ資料

公式ドキュメント　-　これ読むだけでマスターという事にはならないけど、大変良い資料

http://doc.akka.io/docs/akka/current/scala/stream/index.html

API リファレンス　-　DSL と実装 (impl) に分かれていて読みやすい

http://doc.akka.io/api/akka/current/akka/stream/

ご清聴ありがとうございました

Thank you

グラフを知って理解する Akka Streams
2017-09-09 Scala Kansai Summit　@kamijin_fanta