PARADIGMI DI PROGRAMMAZIONE

A.A. 2021/2022

Laurea triennale in Informatica

21: Attori

MESSAGGI

Il paradigma reattivo ci ha portato ad esaminare un modello di esecuzione particolare, con lo scopo di ottenere benefici specifici per un insieme di problemi con esigenze specifiche, anche se sempre più comuni.

L'implementazione che abbiamo esaminato, `RxJava`, non è tuttavia completamente conforme al *Reactive Manifesto* perché non è basata sullo scambio di *messaggi*.

Il modello di esecuzione che gli autori del *Manifesto* avevano in mente è in realtà un modello molto precedente, molto peculiare, nato per requisiti ancora più specifici, ma che si è rivelato estremamente utile negli ambiti in cui il paradigma reattivo si muoveva.

Quello che andiamo ad esaminare quindi è un'altro paradigma, un altro modello di esecuzione, con la caratteristica però di essere in grado di fornire un'ottima implementazione del modello *Reactive* e di soddisfare i requisiti del *Reactive Manifesto*.

ORIGINI

Il modello ad attori viene formalmente definito da Carl Hewitt nel 1973, in relazione ad esperienze provenienti da Lisp, SIMULA ed i primi sistemi orientati agli oggetti.

A partire dal quella prima definizione si sviluppa un proficuo filone di ricerca nella modellazione di sistemi concorrenti e distribuiti.

Oggi il termine
"Actor model"
richiama invece il
linguaggio Erlang ed la sua
piattaforma di runtime
OTP.

Sebbene il lessico sia differente (in Erlang si parla di "processi" e non di attori), la tecnologia inventata alla fine degli anni '80 da Däcker e Armstrong nel loro lavoro in Ericssonn, il modello risultante è praticamente identico ed è un riferimento per ogni implementazione in altri linguaggi.

Usando questo approccio per modellare la concorrenza e la distribuzione dei processi, Ericssonn può vantare all'inizio del 2000 una fetta di mercato dominante nell'ambito degli apparati telefonici, ed un prodotto come il AXD301 che vanta un'affidabilità di "9 nines"

Oggi quindi il modello degli attori significa concorrenza e distribuzione su larga scala, e affidabilità di lunga durata.

CARATTERISTICHE

Un Attore è una unità indipendente di elaborazione, dotata di uno stato privato, inacessibile dall'esterno.

L'attore comunica unicamente con messaggi diretti ad altri attori di cui conosce il nome.

Un Attore, in un determinato momento, ha un Comportamento che definisce la sua reazione ad un determinato messaggio.

Nel reagire ad un messaggio, un attore può:

- mutare il proprio stato interno
- creare nuovi attori
- inviare messaggi ad attori noti
- cambiare il suo comportamento

All'interno dell'attore non c'è concorrenza: l'esecuzione è strettamente sequenziale. Ovviamente, è preferibile che la risposta sia il più veloce possibile.

Al di fuori dell'attore tutto è concorrente e distribuito:

- un altro attore può trovarsi dovunque
- ogni attore è concorrente a tutti gli altri
- ogni messaggio è concorrente a tutti gli altri

Il fallimento di un attore è una eventualità assolutamente normale, che non ha alcuna conseguenza sulla stabilità del programma in sè.

Un Attore può supervisionare gli attori che ha creato, ed essere notificato del loro fallimento.

La notifica è un messaggio, per il quale il supervisore deve predisporre un comportamento.

Le conseguenze di questo approccio sul modello di esecuzione sono profonde;
il risultato è un sistema dove

- l'avvio di nuovi processi/attori è estremamente economico
- l'affidabilità è molto elevata

- le performance e l'efficienza sono molto alte
- la scalabilità è lineare o quasi, e molto stabile
- la distribuzione e la concorrenza sono caratteristiche primarie.

L'elaborazione di grandi moli di dati asincroni, spesso modellati come eventi, è il tipo di problema in cui questo paradigma si esprime al meglio, abilitando risultati molto difficili da raggiungere in modo differente.

PROBLEMATICHE

Ovviamente, le caratteristiche del modello ad Attori hanno un costo.

In questo caso, è un costo di natura concettuale.

Modellare un algoritmo come l'interazione di un un insieme di attori concorrenti non è sempre naturale.

Quando lo è, il risultato è estremamente efficace.

Molte delle euristiche e dei modi di pensare tipici della programmazione tradizionale sono in questo modello totalmente inutili, se non addirittura dannosi.

Il modello mentale in cui il problema va ricondotto è molto differente.

L'interazione non può fare assolutamente nessuna ipotesi nè sull'ordine della ricezione dei messaggi, nè sulla loro affidabilità.

Questo costringe a dover considerare attentamente il modello del risultato e le varie possibilità di fallimento.

IMPLEMENTAZIONE

Per iniziare con alcuni semplici esempi usiamo un modello di Attori minimale, descritto da Edoardo Vacchi in una serie di articoli alla fine del 2021.

Indirizzo di un attore:

```
public interface Address < T > {
  Address < T > tell(T msg);
}
```

Comportamento di un attore:

```
public interface Behavior < T >
  extends Function < T, Effect < T > > {}
```

Effetto della ricezione di un messaggio:

```
public interface Effect < T >
  extends Function<Behavior < T >, Behavior < T > >
```

Effetto: mantenimento del comportamento attuale

```
static < T > Effect < T > stay() {
  return old -> old;
}
```

Effetto: sostituzione del comportamento attuale

```
static < T > Effect < T > become(Behavior < T > next) {
  return current -> next;
}
```

Effetto: l'attore diventa inattivo

```
static < T > Effect < T > die() {
  return become(
    msg -> {
        return stay();
    });
}
```

Sistema: produce attori

```
public record System(Executor executor) {

  public < T > Address < T > actorOf(
    Function< Address < T >, Behavior < T > > initial) {
```

Costruzione dell'attore: gestione della concorrenza

```
abstract class AtomicRunnableAddress< T >
  implements Address < T >, Runnable {
    AtomicInteger on = new AtomicInteger(0);
}
```

Costruzione dell'attore: inizializzazione

```
var addr = new AtomicRunnableAddress< T >() {
  final ConcurrentLinkedQueue< T > mbox =
    new ConcurrentLinkedQueue<>();
  Behavior < T > behavior= initial.apply(this);
```

Costruzione dell'attore: ricezione di un messaggio

```
public Address<T> tell(T msg) {
  mbox.offer(msg);
  async();
  return this;
}
```

Costruzione dell'attore: elaborazione di un messaggio

```
void async() {
  if (!mbox.isEmpty() && on.compareAndSet(0, 1)) {
    try {
      executor.execute(this);
    } catch (Throwable t) {
      on.set(0);
      throw t;
    }
  }
}
```

Costruzione dell'attore: esecuzione di un messaggio

```
public void run() {
  try {
    if (on.get() == 1)
      behavior = behavior.apply(mbox.poll()).apply(behavior);
  } finally {
    // processing complete, the actor is inactive
    on.set(0);
    async();
  }
}
```

Costruzione dell'attore: conclusione

```
  return addr;
}
```

L'oggetto System, quindi, ci permette di costruire degli attori a cui possiamo associare un comportamento iniziale ed inviare dei messaggi.

Esempio 1: accetta un messaggio poi si spegne

```
Address < String > actor =
  actorSystem.actorOf(
    self -> msg -> {
      out.println("got msg: '" + msg);
      return Effect.die();
    });

actor.tell("foo").tell("bar");
```

Esempio 2: Ping Pong

```
record Ping(Address < Pong > sender) {};

sealed interface Pong {};

record SimplePong(Address < Ping > sender) implements Pong {};
record DeadlyPong(Address < Ping > sender) implements Pong {};
```

```
var actorSystem = new System(Executors.newCachedThreadPool());
Address < Ping > ponger = actorSystem.actorOf(
  self -> msg -> pongerBehavior(self, msg, 0));
Address < Pong > pinger = actorSystem.actorOf(
  self -> msg -> pingerBehavior(self, msg));
ponger.tell(new Ping(pinger));
```

```
static Effect< Ping > pongerBehavior(
  Address< Ping > self, Ping msg, int counter) {
  return switch (msg) {
    case Ping p && counter < 10 -> {
      p.sender().tell(new SimplePong(self));
      yield Effect.become(m ->
        pongerBehavior(self, m, counter + 1));
    }
    case Ping p -> {
      p.sender().tell(new DeadlyPong(self));
      yield Effect.die();
    }
  };
}
```

```
static Effect<Pong> pingerBehavior(Address<Pong> self, Pong msg) {
  return switch (msg) {
    case SimplePong p -> {
      p.sender().tell(new Ping(self));
      yield Effect.stay();
    }
    case DeadlyPong p -> {
      p.sender().tell(new Ping(self));
      yield Effect.die();
    }
  };
}
```

Nota 1: la sintassi di switch usata richiede l'abilitazione delle feature di preview, e potrebbe mettere in crisi il vostro IDE.

Nota 2: l'uso della closure come contenitore di stato deriva dal dualismo closure/oggetto:

a closure is a poor man's object;
an object is a poor man's closure

AKKA

Sulla JVM, la più diffusa implementazione del modello ad attori è il framework Akka

Inventato nel 2009 da Jonas Bonér e sviluppato da Viktor Klang e Roland Khun, prende a esplicito riferimento il modello di Erlang per implementare sulla JVM un sistema ad attori in grado di supportare applicazioni reattive e scalabili.

Le principali caratteristiche del framework sono:

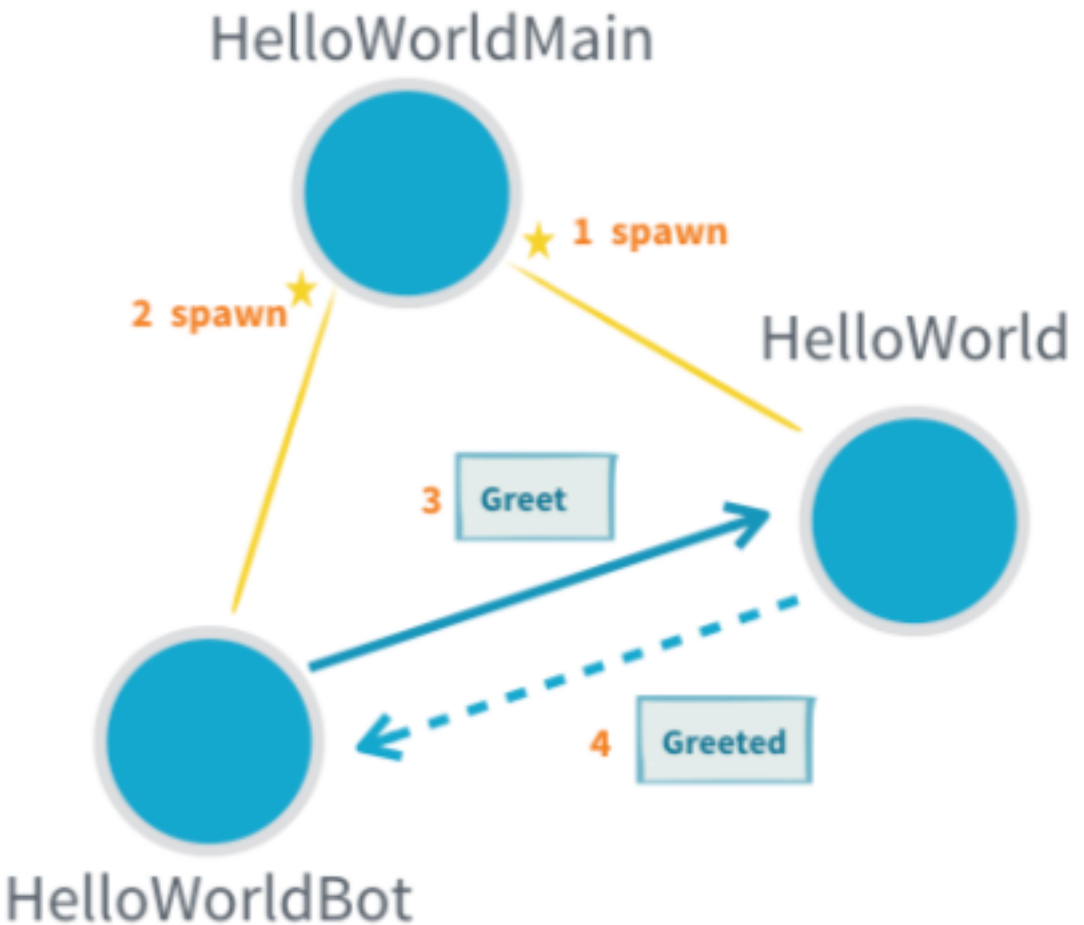
- compatibile con l'ecosistema della JVM, e quindi in grado di usare tutte librerie e le tecnologie disponibili per Java
- scritto in Scala, ma usabile anche attraverso una API Java
- Il modello di supervisione degli attori è obbligatoriamente da parte del "genitore"

Gli attori comunicano fra loro in modo pilotato dai tipi: questo permette di codificare l'interazione nelle caratteristiche delle classi usate per guidare l'interazione. E' il compilatore stesso ad impedirci di inviare messaggi che non possono essere ricevuti.

Akka supporta una implementazione di Reactive Streams che può sfruttare la scalabilità del sistema per raggiungere prestazioni considerevoli esponendo il modello di esecuzione degli stream.

ESEMPIO

Analizziamo un semplice esempio per capire le complessità del modello ad attori.


HelloWorldMain
1 spawn
2 spawn
HelloWorld
3 Greet
4 Greeted
HelloWorldBot

Definiamo per prima cosa i messaggi: saluto e risposta

```
public static final class Greet {
  public final String whom;
  public final ActorRef< Greeted > replyTo;
  public Greet(String whom, ActorRef< Greeted > replyTo) {
    this.whom = whom;
    this.replyTo = replyTo;
  }
}
```

```java
public static final class Greeted {
  public final String whom;
  public final ActorRef< Greet > from;

  public Greeted(String whom, ActorRef< Greet > from) {
    this.whom = whom;
    this.from = from;
  }
}
```

Dobbiamo definire come si comporta l'attore che risponde al saluto al momento della creazione:

```
public class HelloWorld extends
  AbstractBehavior< HelloWorld.Greet > {

  public static Behavior< Greet > create() {
    return Behaviors.setup(HelloWorld::new);
  }

  private HelloWorld(ActorContext< Greet > context) {
    super(context);
  }
```

Infine, dobbiamo definire come si comporta al ricevimento di un messaggio:

```
@Override
public Receive< Greet > createReceive() {
  return newReceiveBuilder().onMessage(Greet.class,
    this::onGreet).build();
}

private Behavior< Greet > onGreet(Greet command) {
  getContext().getLog().info("Hello {}!", command.whom);
  command.replyTo.tell(new Greeted(command.whom,
    getContext().getSelf()));
  return this;
}
```

Ora, l'attore che invia il saluto. Il comportamento alla creazione:

```
public class HelloWorldBot
  extends AbstractBehavior< HelloWorld.Greeted > {

  public static Behavior< HelloWorld.Greeted >
    create(int max) {
    return Behaviors.setup(c -> new HelloWorldBot(c, max));
  }
  private final int max;
  private int greetingCounter;

  private HelloWorldBot(ActorContext< HelloWorld.Greeted >
    ctx, int max) {
    super(ctx); this.max = max;
  }
```

Se abbiamo raggiunto il massimo, interrompiamo le risposte terminando l'esecuzione dell'attore. Altrimenti, rispondiamo a chi ci ha inviato il saluto.

```
@Override
public Receive< HelloWorld.Greeted > createReceive() {
  return newReceiveBuilder().onMessage(
    HelloWorld.Greeted.class, this::onGreeted).build();
}
```

```
private Behavior< HelloWorld.Greeted >
  onGreeted(HelloWorld.Greeted message) {
    greetingCounter++;
    getContext().getLog().info("Greeting {} for {}",
      greetingCounter, message.whom);
    if (greetingCounter == max) {
      return Behaviors.stopped();
    } else {
      message.from.tell(
        new HelloWorld.Greet(message.whom,
          getContext().getSelf()));
    return this;
  }
}
```

L'attore che coreografa l'interazione deve rispondere ad un messaggio diverso:

```
public class HelloWorldMain
  extends AbstractBehavior< HelloWorldMain.SayHello> {

  public static class SayHello {
    public final String name;

    public SayHello(String name) {
      this.name = name;
    }
  }
```

Il suo stato è il riferimento all'attore creato al momento della costruzione.

```
private final ActorRef< HelloWorld.Greet > greeter;

public static Behavior< SayHello > create() {
  return Behaviors.setup(HelloWorldMain::new);
}

private HelloWorldMain(ActorContext< SayHello > context) {
  super(context);
  greeter = context.spawn(HelloWorld.create(), "greeter");
}
```

Al ricevimento del messaggio d'avvio, crea l'attore di risposta, e invia il primo saluto per avviare la conversazione.

```
@Override
public Receive< SayHello > createReceive() {
  return newReceiveBuilder().onMessage(SayHello.class,
    this::onSayHello).build();
}

private Behavior< SayHello > onSayHello(SayHello command) {
  ActorRef< HelloWorld.Greeted > replyTo = getContext()
    .spawn(HelloWorldBot.create(3), command.name);
  greeter.tell(new HelloWorld.Greet(command.name, replyTo));
  return this;
}
```

La classe principale costruisce il sistema di attori, e avvia il coreografo.

```
public static void main(String[] args) {

  final ActorSystem< HelloWorldMain.SayHello > greeterMain =
    ActorSystem.create(HelloWorldMain.create(), "helloakka");
  greeterMain.tell(new HelloWorldMain.SayHello("Charles"));

  try {
    System.in.read();
  } catch (IOException ignored) {
  } finally {
      greeterMain.terminate();
  }
}
```