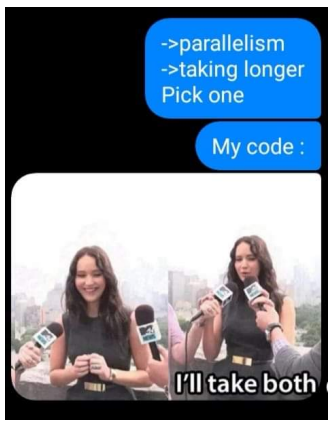
Aktorių modelis

Karolis Ryselis

Kauno Technologijos Universitetas



Paskaitos turinys

ktu 1922

Aktorių modelis

- Vienas iš paskirstytos atminties variantų.
- Kiekvienas procesas laikomas aktoriumi.
- Kiekvienas aktorius turi savo „pašto dėžutę“, į kurią gali priimti žinutes.
- Kiekvienas aktorius gali kitam aktoriui siųsti žinutes, iš kitų aktorių priimti žinutes, jas apdoroti.
- Žinučių apdorojimas vykdomas nuosekliai.

Aktorių modelis

- Procesų pašto dėžutėse laikomos visos dar neapdorotos žinutės.
- Procesas vykdomas kaip žinučių apdorojimo ciklas.
- Pašto dėžutė elgiasi kaip buferis, t. y., siunčiantis procesas žinutę įrašo ir toliau tęsia darbą, o gavėjas žinutę apdoros, kai ateis tos žinutės eilė.
- Jei pašto dėžutė tuščia, o procesas bando apdoroti sekančią žinutę, jis yra blokuojamas, kol ta žinutė ateis. Dėl šios priežasties gali susidaryti aklavietės situacija (*deadlock*).

Aktorių modelis

- Pašto dėžučių programuotojas tiesiogiai nepasiekia.
- Žinutės yra siunčiamos tiesiogiai procesui, t. y., egzistuoja kintamasis, kuris identifikuoja procesą (proceso ID, procesą atitinkantis objektas ar pan.), ir tas identifikatorius naudojamas žinutei siųsti.
- Žinutės gavėjas „mato“ tik pačias žinutes; jei reikia žinoti, kuris procesas siuntė, tai reikia tą realizuoti pačiam.

Aktorių modelis ir kiti modeliai

- Priešingai nei CSP modelyje, aktorių modelyje nereikalingi kanalai, žinutės siunčiamos tiesiai reikiamam procesui.
- Priešingai nei MPI modelyje, nereikalingas komunikatorius, kuriuo siunčiamos žinutės.
- Pašto dėžučių buferiai elgiasi panašiai, kaip buferizuoti kanalai, t. y., galima siųsti žinutes, kol jos nepriimamos.
- Modelis tinkamas skaičiavimams paskirstyti per keletą mašinų, nes nereikalingi kanalai, kurie turi būti pasiekiami keliems procesams.
- Aktorių modelyje pašto dėžutė paslėpta nuo programuotojo, negalima nustatyti jos dydžio, kaip sinchronizuotiems kanalams. Programuotojui reikia pačiam užtikrinti, kad nebus perpildoma pašto dėžutė ir jai užteks atminties.

ktu 1922

Aktorių modelio realizacijos

- Aktorių modelis realizuotas šiose programavimo kalbose:
  - Erlang
  - Dart
  - Elixir
  - kt.
- Bibliotekos, realizuojančios aktorių modelį:
  - Akka (Scala)
  - Akka.NET (C#)
  - MailboxProcessor (F#)
  - Rotor (C++)
  - Bastion (Rust)
  - kt.

Elixir

- Funkcinio programavimo kalba, veikianti su Erlang virtualia mašina, skirta lengvasvoriams procesams vykdyti.
- Tinka realaus laiko sistemoms, paskirstytoms sistemoms, klaidas toleruojančioms sistemoms.
- Taip pat naudojama saityno programoms, įterptinėms sistemoms ir kt.
- Kadangi vykdomi lengvasvoriai procesai, kurie paskirstomi OS gijoms, jų galima turėti didelius kiekius (taip pat, kaip Go kalboje).
- Turi pritaikytą sintaksę žinučių siuntimui ir priėmimui.

Elixir

- Kalba sukurta 2011 m.
- Kalba įkvėpta Erlang, Ruby ir Closure programavimo kalbų, sintaksė primena Ruby sintaksę.

Elixir procesai

`spawn(module, function, args)`

Funkcija, paleidžianti Elixir lengvasvorį procesą.

- `module` Modulis, kurio funkciją reikia kviesti
- `function` Atomas, nurodantis kviečiamos funkcijos pavadinimą
- `args` Kviečiamos funkcijos parametrai (sąrašas)

`spawn(function)`

- `function` Funkcija, kurią reikia paleisti naujame procese.

Funkcija `spawn` grąžina sukurto proceso ID — specialų kintamąjį, skirtą procesui identifikuoti, pvz., `#PID<0.132.0>`

Elixir žinutės siuntimas

`send(pid, message)`

Funkcija, siunčianti žinutę kitam procesui,

`pid` Proceso ID

`message` Žinutė

Žinutė gali būti bet kokio tipo, taip pat ir kelių reikšmių rinkinys.

Elixir žinutės priėmimas

- Žinutės priėmimui naudojamas `receive` sakinys, kuris leidžia pagal žinutės turinį parinkti reikiamą atlikti veiksmą.
- Žinutės priėmimo sintaksė:

```
receive do
  msg1 -> action1
  msg2 -> action2
end
```

- Vienas `receive` bloko iškvietimas yra vienas žinutės priėmimas. Jei norime priimti daugiau žinučių, tame procese reikia vėl kviesti `receive`.
- Kadangi Elixir yra funkcinio programavimo kalba, veiksmų kartojimas dažniausiai realizuojamas kaip rekursinis tos pačios funkcijos kvietimas.

Elixir paprastas skaitiklis

```
def count(initial) do
  receive do
    {:increase} -> count(initial + 1)
    {:decrease} -> count(initial - 1)
    {:stop} -> IO.puts initial
  end
end
```

Elixir paprastas skaitiklis

```
def increase(counter, stopper, remaining_increases) do
  if remaining_increases > 0 do
    send counter, {:increase}
    increase(counter, stopper, remaining_increases - 1)
  else
    send stopper, {:stop}
  end
end

def decrease(counter, stopper, remaining_decreases) do
  if remaining_decreases > 0 do
    send counter, {:decrease}
    decrease(counter, stopper, remaining_decreases - 1)
  else
    send stopper, {:stop}
  end
end
```

Elixir paprastas skaitiklis

```
def stopper(stopped_process_count, counter) do
  if stopped_process_count == 2 do
    send counter, {:stop}
  else
    receive do
      {:stop} -> stopper(stopped_process_count + 1,
                         counter)
    end
  end
end
```

Elixir paprastas skaitiklis

```
def start(_type, _args) do
  counter = spawn(ActorTest, :count, [0])
  stopper = spawn(ActorTest, :stopper, [0, counter])
  spawn(ActorTest, :increase, [counter, stopper, 50])
  spawn(ActorTest, :decrease, [counter, stopper, 50])

  Task.start(fn -> 0 end)
end
```

Elixir žinutės priėmimas

- `receive` bloke galima nurodyti `after` šaką:

```
receive do
  msg1 -> action1
  msg2 -> action2
  after 1000 -> action3
end
```

- `after` šaka bus vykdoma, jei tam tikrą laiką (duotu atveju 1000 ms) neateis jokia žinutė.

Elixir agentai

- Agentas — specialus aktorius, skirtas tam tikrai būsenai saugoti ir valdyti.
- Agentas jau turi realizuotas būsenos valdymo funkcijas.
- Naudojantis agentais patogu kurti komponentą, kuris viduje turi būseną, o kiti procesai jam gali siųsti žinutes tos būsenos atnaujinimui.
- Būseną keičiantys aktoriai patys gali nurodyti, kaip ta būsena turi būti keičiama, t. y., agentas užsiima tik būsenos valdymu, o kaip ji keičiama, sprendžia kiti aktoriai.

Agento sukūrimas

`Agent.start_link(fun)`

Funkcija sukuria ir paleidžia procesą — agentą.

`fun` Funkcija, kuri nepriima parametrų ir grąžina reikšmę, kuri bus pradinė agento saugoma reikšmė.

Funkcija grąžina dvi reikšmes — paleidimo būseną ir proceso ID: `{:ok, pid}`.

Agento būsenos valdymas

`Agent.update(agent, fun)`

Funkcija, atnaujinanti agento būseną.

`agent` Agento proceso ID

`fun` Funkcija, parametru priimanti esamą agento būseną ir grąžinanti naują norimą būseną.

`Agent.get(agent, fun)`

Funkcija, vykdanti naudotojo pasirinktą funkciją su esama būsena ir grąžinanti jos rezultatą.

`agent` Agento proceso ID

`fun` Funkcija, parametru priimanti esamą agento būseną. Iš `Agent.get` grąžinamas šios funkcijos rezultatas.

Elixir paprastas skaitiklis su agentu

```
def increase(counter, stopper, remaining_increases) do
  if remaining_increases > 0 do
    Agent.update(counter, fn (count) -> count + 1 end)
    increase(counter, stopper, remaining_increases - 1)
  else
    send stopper, {:stop}
  end
end

def decrease(counter, stopper, remaining_decreases) do
  if remaining_decreases > 0 do
    Agent.update(counter, fn (count) -> count - 1 end)
    decrease(counter, stopper, remaining_decreases - 1)
  else
    send stopper, {:stop}
  end
end
```

Elixir paprastas skaitiklis su agentu

```
def stopper(stopped_process_count, counter) do
  if stopped_process_count == 2 do
    IO.puts Agent.get(counter, fn count -> count end)
  else
    receive do
      {:stop} -> stopper(stopped_process_count + 1, counter)
    end
  end
end
```

Elixir paprastas skaitiklis su agentu

```
def start(_args, _type) do
  {:ok, counter} = Agent.start_link(fn -> 0 end)
  stopper = spawn(Agents, :stopper, [0, counter])
  spawn(Agents, :increase, [counter, stopper, 50])
  spawn(Agents, :decrease, [counter, stopper, 50])

  Task.start(fn -> 0 end)
end
```

GenServer

- Elixir turi dar vieną aktoriaus tipą - `GenServer`.
- Šis aktorius palaiko keletą žinučių siuntimo tipų:
  - kvietimas - priima žinutę ir siunčia atsakymą, dėl to palaiko tik sinchroninį siuntimą;
  - perdavimas - priima žinutę ir nesiunčia atsakymo, šis variantas yra asinchroninis;
  - informacinis - panašus į perdavimą, bet skirtas žinutėms, kurias aktorius siunčia sau.

Serverio kūrimas

- Serveris kuriamas sukuriant modulį, kuris naudoja `GenServer` modulį.
- Serveris paleidžiamas kviečiant sukurto modulio `start_link` funkciją.
- Serveris viduje saugo būseną, pradinė būsena grąžinama modulio `init` funkcijoje.

Kvietimas

- Žinutė siunčiama naudojant aktoriaus funkciją `GenServer.call(pid, message)`
  - `pid` - aktoriaus ID;
  - `message` - siunčiama žinutė.
- Žinutė apdorojama aktoriui realizuojant funkciją `handle_call(message, sender, current_state)`
  - `message` - siunčiama žinutė;
  - `sender` - siuntėjo ID;
  - `current_state` - dabartinė būsena.
- `handle_call` grąžina struktūrą `{:reply, status, new_state}`. Čia `:reply` yra atomas, kuris nurodo, kad grąžinamas atsakymas. `status` grąžinamas tam, kas kvietė, `new_state` - nauja serverio būsena.

Perdavimas

- Žinutė siunčiama naudojant aktoriaus funkciją `GenServer.cast(pid, message)`
  - `pid` - aktoriaus ID;
  - `message` - siunčiama žinutė.
- Žinutė apdorojama aktoriui realizuojant funkciją `handle_cast(message, current_state)`
  - `message` - siunčiama žinutė;
  - `current_state` - dabartinė būsena.
- `handle_cast` grąžina struktūrą `{:noreply, new_state}`. Čia `:noreply` yra atomas, kuris nurodo, kad atsakymas negrąžinamas.

Informaciniai pranešimai

- Žinutė apdorojama aktoriui realizuojant funkciją `handle_info(message, current_state)`
  - `message` - siunčiama žinutė;
  - `current_state` - dabartinė būsena.
- `handle_cast` grąžina struktūrą `{:noreply, new_state}`. Čia `:noreply` yra atomas, kuris nurodo, kad atsakymas negrąžinamas.

Sumuojantis aktorius

- Sekančio pavyzdžio esmė - realizuoti aktorių, kuris priima skaičius ir saugo visų gautų skaičių vidurkį.
- Vidurkis skaičiuojamas ne tada, kai gautas skaičius, o vienu iš dviejų atvejų:
  - Kai praeina tam tikras laiko tarpas nuo paskutinio vidurkio perskaičiavimo;
  - Kai susikaupia pilnas buferis elementų, kuriems dar neskaičiuotas vidurkis.

Sumuojantis aktorius

```elixir
defmodule Aggregator do
  use GenServer

  @flush_interval_ms 500
  @max_capacity 100

  def start_link() do
    GenServer.start_link(__MODULE__, :ok)
  end

  def init(_opts) do
    timer = Process.send_after(self(), :tick, @flush_interval_ms)
    {:ok, %{value_count: 0, average: 0, buffered_values: [], timer: timer}}
  end

  def add_value(pid, value) do
    GenServer.cast(pid, {:add, value})
  end
```

Sumuojantis aktorius

```
def handle_cast(
      {:add, value},
      %{value_count: value_count, average: average,
        buffered_values: buffered_values, timer: timer}
    ) do
  updated_values = [value | buffered_values]
  if length(updated_values) > @max_capacity do
    Process.cancel_timer(timer)
    %{value_count: total_new_values,
      average: new_average
    } = transfer_average(updated_values, value_count, average)
    new_timer = Process.send_after(self(), :tick, @flush_interval_ms)
    {:noreply, %{value_count: total_new_values, average: new_average,
      buffered_values: [], timer: new_timer}}
  else
    {:noreply, %{value_count: value_count, average: average,
      buffered_values: updated_values, timer: timer}}
  end
end
```

Sumuojantis aktorius

```
defp transfer_average(updated_values, value_count, average) do
  item_count = length(updated_values)
  if item_count == 0 do
    %{value_count: value_count, average: average}
  else
    total_new_values = value_count + item_count
    new_average = (value_count * average +
                     Enum.sum(updated_values)) / total_new_values
    %{value_count: total_new_values, average: new_average}
  end
end

def handle_info(
      :tick,
      %{value_count: value_count, average: average,
        buffered_values: buffered_values} = _state
    ) do
  %{value_count: total_new_values,
    average: new_average} = transfer_average(buffered_values, value_count,
                                             average)
  new_timer = Process.send_after(self(), :tick, @flush_interval_ms)
  {:noreply, %{value_count: total_new_values, average: new_average,
    buffered_values: [], timer: new_timer}}
end
```

Elixir užduotys

- Elixir palaiko užduočių programavimo modulį, panašų į C# `async` / `await`, bet paremtą aktoriais.
- Užduotis paleidžiama naudojant funkciją `task = Task.async(fn -> compute() end)`.
- Užduoties rezultato laukiama kviečiant `res = Task.await(task)`.
- `Task.async` grąžins užduotį, kuri iš karto pradedama vykdyti, o po vykdymo siųs žinutę (funkcijos suskaičiuotą rezultatą), kurią galima perskaityti su `Task.await`.

Kelių užduočių vykdymas lygiagrečiai

```
defmodule AsyncExample do
  use Application
  defp factorial(n) when n >= 0, do: Enum.reduce(1..n, 1, &(&1*&2))

  def start(_args, _type) do
    task1 = Task.async(fn -> factorial(50) end)
    task2 = Task.async(fn -> factorial(70) end)
    task3 = Task.async(fn -> factorial(100) end)

    result1 = Task.await(task1)
    result2 = Task.await(task2)
    result3 = Task.await(task3)

    IO.puts "Factorials: #{result1}, #{result2}, #{result3}"
    Task.start(fn -> 0 end)
  end
end
```