Karnaj

Geste de savoir

Les acteurs à la rescousse des philosophes

21 janvier 2021

Table des matières

J'ai déjà plus ou moins parlé du modèle d'acteurs de Ruby dans ce billet . L'idée est d'avoir une abstraction pour faciliter la tâche d'écriture de programmes concurrents.

Le modèle d'acteurs permet d'avoir plusieurs tâches s'exécutant en parallèle dans leurs threads, **sans partager leurs données de manière directe**. Ainsi, il y a beaucoup moins de problème d'accès concurrent à une ressource vu que les acteurs sont isolés dans leur bulle. Pour communiquer, ils peuvent cependant s'envoyer des messages.

Dans ce billet, nous allons utiliser ce modèle pour résoudre le fameux problème du dîner de philosophes . L'idée est la suivante: un groupe de `n` philosophes se retrouve autour d'une table (ronde) pour manger des spaghetti. Chacun a son plat devant lui et à gauche de chaque plat se trouve une fourchette.

Un philosophe fait deux choses: manger et penser. Il commence par penser pendant un certain temps, ce qui lui creuse fortement l'estomac. Il décide alors de manger. Pour cela, il doit prendre la fourchette à gauche de son assiette, et celle à gauche de l'assiette de son voisin (de droite).

On se rend tout de suite compte que si un philosophe est en train de manger, alors ses voisins ne peuvent pas manger! Il pourrait essayer d'arracher les fourchettes des mains de ses voisins, mais ce ne serait pas très courtois… Donc si les fourchettes ne sont pas disponibles, on va dire qu'il se remet à penser avant de réessayer. S'il réussit à avoir les fourchettes, il mange pendant un certain temps, puis nettoie les fourchettes et les redépose.

Votre mission, si vous l'acceptez, est de faire en sorte que chaque philosophe mange. Pour plus d'information sur les modalités du problème, ne pas hésiter à lire la page Wikipedia

> *i* Cette situation est celle de différents tâches (les philosophes) qui ont besoin de ressources (les fourchettes) pour s'exécuter et c'est donc un problème classique de programmation concurrente et de partage de ressources.

En particulier, on peut facilement se retrouver dans une situation d'interblocage : chaque philosophe prend la fourchette a sa gauche puis attend que la deuxième fourchette dont il a besoin soit libre.

Dans notre résolution du problème, chaque philosophe sera un acteur. Et nous aurons un acteur supplémentaire, un **serveur** en costume trois pièces (avec cravate et pochette assortie). Ce serveur sera le seul à gérer les ressources (les fourchettes).

— Quand un philosophe a fini de penser, il n'essaye pas de prendre les fourchettes, il appelle le serveur et lui demande s'il peut manger. Le serveur regarde alors si les fourchettes sont libres et les lui donne si c'est le cas.
— Quand un philosophe a fini de manger, il appelle le serveur, lui dit qu'il a fini et lui remet les fourchettes.

Ici, on a code qui correspond à peu près à cette idée. Il manque juste le passage de ressources du serveur aux philosophes; mon serveur a juste un tableau indiquant si les fourchettes sont libres.

```ruby
class Waiter
  def initialize(n)
    @forks = Array.new(n, false)
  end

  def call?(i, j)
    if @forks[i] || @forks[j]
      print "Les couverts #{i} et #{j} ne sont pas libres.\n"
      false
    else
      print "Les couverts #{i} et #{j} sont libres.\n"
      @forks[i] = true
      @forks[j] = true
      true
    end
  end

  def clean(i, j)
    @forks[i] = false
    @forks[j] = false
  end
end

class Philosopher
  def initialize(name, waiter, left_fork, right_fork)
    @name = name
    @waiter = waiter
    @left_fork = left_fork
    @right_fork = right_fork
  end

  def think
    print "#{@name} pense\n"
    sleep(rand(3..8))
  end

  def eat
    print "#{@name} mange.\n"
    sleep(3)
```

```
40      @waiter.send([Ractor.current, :end, @left_fork, @right_fork])
41    end
42
43    def run
44      loop do
45        think
46        @waiter << [Ractor.current, :hungry, @left_fork, @right_fork]
47        eat if Ractor.recv
48      end
49    end
50  end
51
52  waiter = Waiter.new(4)
53  philosophers = ["Aristote", "Platon", "Nietzsche", "Descartes"]
54  n = philosophers.size
55  ractors = philosophers.each_with_index.map do |name, i|
56    r = Ractor.new(name, i, (i + 1) % n, Ractor.current) do |p_name,
          j, k, waiter|
57      p = Philosopher.new(p_name, waiter, j, k)
58      p.run
59    end
60  end
61
62  loop do
63    p, m, i, j = Ractor.recv
64    p << waiter.call?(i, j) if m == :hungry
65    waiter.clean(i, j) if m == :end
66  end
```

Je vous laisse comprendre le code qui n'est pas vraiment compliqué et retranscrit assez bien l'idée décrite. L'acteur principal est dédié au serveur; il reçoit les messages des philosophes et les traite. Chaque philosophe est dans son acteur, demande au serveur s'il peut manger et le prévient quand il a terminé.