Karnaj

Zeste de savoir

Les Ractors de Ruby 3

1er octobre 2020

Table des matières

Comme nous le savons tous plus ou moins, la version 3 de Ruby 3 est enfin disponible, et elle vient avec plusieurs nouveautés qui permettront d'enthousiasmer petits et grands. On notera en particulier l'apparition de RBS et d'un type-checker et de Ractor qui nous fournit une abstraction pour la concurrence en se basant sur le modèle d'acteurs.

> **Sortie de Ruby 3**
>
> Pour plus d'informations sur les nouveautés de Ruby 3, voir l'article du site officiel de Ruby .

La nouvelle abstraction pour la concurrence était prévue depuis un moment sous le nom de *Guild*. Le but était notamment de fournir un outil permettant d'écrire des programmes en parallèles plus simplement et avec plus de sûreté qu'en utilisant juste des threads, et de supprimer le GIL de Ruby (qui ne permettait pas d'exécuter plusieurs threads en même temps).

Cette abstraction se basant sur le modèle d'acteur, elle a été renommée *Ractor* pour *Ruby actor* (nom d'atant plus cool qu'il fait penser à *Raptor* et à *Réactor*). Dans ce billet, nous allons nous amuser un peu avec cette nouvelle abstraction pour la concurrence, voir un peu comment elle fonctionne et ce qu'elle permet de faire.

1. Fonctionnement de base

1.1. La base

L'idée du modèle d'acteurs est d'avoir un système où des entités appelées «acteurs» fonctionnent en parallèle et communiquent à l'aide de message et ne partagent pas la majorité de leurs données. Ainsi, le travail à faire est réparti entre des acteurs isolés qui lorsqu'ils ont besoin de partager de l'information, s'envoient des messages.

Ce système a alors plusieurs avantages. En particulier, vu que les acteurs sont isolés (ils ne partagent pas de variables), ils n'ont pas à se demander si un autre acteur est en train d'utiliser une variable lorsqu'ils souhaitent l'utiliser (là où des mutex seraient utilisés avec des threads).

Avec Ruby, nous créons un acteur avec la classe `Ractor`. Chaque acteur a un ou plusieurs threads et le code principal est lui aussi exécuté dans un acteur (c'est vraiment un système où tout est acteur). Voici un premier exemple d'utilisation.

```ruby
hello = Ractor.new do
  puts 'Hello word dans Ractor.'
end
puts 'Hello word.'
```

Ici, nous avons créé un acteur qui se contente d'afficher un message et nous affichons également un message en dehors de cet acteur. Après plusieurs exécutions, nous nous rendons compte que l'affichage peut changer d'une exécution à l'autre. En effet, le code de l'acteur et le code «principal» s'exécutent en parallèle.

Parfois le message de l'acteur n'est pas affiché. En fait, le programme se termine une fois que le code principal a fini de s'exécuter et n'attend pas que tous les threads aient fini de s'exécuter.

Avec `Ractor#take`, nous attendons un résultat de la part d'un acteur. Ainsi, en rajoutant `hello.take` à la fin de notre code, nous attendrons la fin de l'exécution de `hello`.

```ruby
hello = Ractor.new do
  puts 'Hello word dans Ractor.'
end
puts 'Hello word.'
hello.take
```

Ça y est, nous avons complété ce premier exemple!

1.2. Envoi de message

Comme nous l'avons dit, les acteurs sont isolés. En particulier, un acteur ne peut pas accéder aux variables créées en dehors de son bloc.

```ruby
name = 'Clem'
hello = Ractor.new do
  puts "Hello #{name} dans Ractor."
end
puts "Hello #{name}."
hello.take
```

Avec ce code, nous obtenons une `ArgumentError` avec le message *can not isolate a Proc because it can accesses outer variables.* Pour dire quelque chose à un acteur, il nous faut lui transmettre un message. Cela se fait avec la méthode `Ractor#send` (d'alias `Ractor#<`). La réception du message, elle, se fait à l'aide de `Ractor::recv` qui demande à l'acteur courant de récupérer un message.

Il faut vraiment comprendre ces deux méthodes comme un système de messagerie. Avec `<`, on envoie un message à un acteur; le message est envoyé immédiatement et est mis dans la liste des messages de l'acteur (il n'y a pas d'attente). Avec `recv`, on récupère le premier message non lu pour le traiter (là par contre, on n'est obligé d'en attendre un pour le traiter).

Faisons donc quelques petits tests avec ces nouvelles méthodes. Commençons par corriger notre code précédent.

```
1 name = 'Clem'
2 hello = Ractor.new do
3   msg = Ractor.recv
4   puts "Hello #{msg} dans Ractor."
5 end
6 puts "Hello #{name}."
7 hello << name
8 hello.take
```

Ici, puisque nous envoyons le message à `hello` après l'affichage de l'acteur principal, il est certain que le message de `hello` sera affiché en dernier. En échangeant la ligne d'envoi et celle d'affichage, ce n'est plus forcément le cas.

Notons que les objets envoyés sont copiés (de manière profonde) pour éviter qu'ils ne soient partagées entre plusieurs acteurs et garder ces derniers isolés. En pratique, c'est un peu plus compliqué et certains objets (dont les objets immuables) peuvent être partagés directement (on parle de *shareable objects* et d'*unshareable objects* dans l'autre cas). La plupart des objets ne sont pas partageable.

Maintenant, ayons plusieurs envois et plusieurs réceptions.

```
1 ractor = Ractor.new do
2   puts 'Je fais une addition'
3   a = Ractor.recv
4   puts 'Le premier nombre est donné.'
5   b = Ractor.recv
6   puts 'Le deuxième nombre est donné.'
7   puts a + b
8 end
9 puts 'On va faire une addition dans un acteur.'
```

```
puts 'On lui transmet le premier nombre.'
ractor << 10
puts 'On a fini'
ractor.take
```

On donne le premier nombre à `ractor`, mais pas le second et donc `ractor` se bloque à la réception du premier message, tandis que l'acteur principal est bloqué parce qu'il attend la fin de `ractor`. Comme quoi les acteurs ne sont pas exempts de tout défaut… Mais ils simplifient quand même beaucoup la vie!

2. Des conversations endiablées

Une métaphore de la vie réelle serait de voir les acteurs comme le personnel d'une entreprise. Chacun travaille dans son propre bureau et ils discutent en s'envoyant des messages par mail. Un employé peut avoir besoin d'une information pour faire son travail donc il attend de la recevoir (et fait des pauses café en attendant) et on peut attendre qu'un employé finisse une tâche pour poursuivre sa propre tâche.

Avec cette métaphore en tête, nous voyons comment construire nos acteurs.

2.1. Le retour de bâton

Pour commencer, il nous faut voir comment un employé peut s'exprimer. Pour qu'il envoie un message, mais pour cela, il lui faut connaître la boîte mail de celui qu'il veut contacter (avoir un acteur à qui envoyer le message). Par exemple, dans le code qui suit, nous lui envoyons un acteur en message à qui il peut envoyer un message quand il a fini.

`Ractor::current`

La méthode `Ractor::current` permet d'obtenir l'acteur courant. Dans le code qui suit, nous le transmettons à `employee` pour qu'il sache à qui répondre une fois son travail terminé.

```
employee = Ractor.new do
  actor = Ractor.recv
  puts 'Je commence le travail, je préviens quand il est fini.'
  sleep(3)
  actor << 42
end

employee << Ractor.current
result = employee.recv
puts result
```

Avec `<`, on envoie un message à un acteur particulier et avec `recv`, on reçoit des messages de n'importe qui. Il existe une autre méthode de communication basée sur les méthodes `Ractor::yield` et `Ractor#take` (que nous avons déjà utilisée).

Avec `yield`, un acteur indique qu'il a un résultat à transmettre et attend qu'un autre acteur récupère ce résultat avec `take`. Ainsi, l'acteur qui `yield` ne sait pas qui récupérera son résultat, mais celui qui récupère sait tout à fait de qui provient le résultat. Les deux méthodes sont bloquantes.

```ruby
employee = Ractor.new do
  puts 'Employé : je travaille.'
  Ractor.yield 1
  sleep(3)
  Ractor.yield 2
  puts 'Employé : hop, un dernier résultat et je rentre.'
  Ractor.yield 3
  puts
      'Employé : il a pris tout son temps pour prendre le dernier !'
end

puts "Je récupère un résultat de employee : #{employee.take}"
puts "J'attends un autre résultat de employee."
puts "Enfin : #{employee.take} !"
puts "À lui d'attendre, je vais pas passer le voir tout de suite !"
sleep(3)
puts "Je vais voir son résultat : #{employee.take}."
```

i

Si l'on utilise `take` pour attendre la fin d'un acteur, c'est parce qu'un acteur yield la valeur de la dernière expression de son bloc de code (un peu comme la valeur de la dernière expression est implicitement retournée dans une fonction).

```ruby
r = Ractor.new { 2 }
puts r.take # => 2
```

Si on reprend la métaphore de l'entreprise ça donne à peu près ça.

— Avec `Ractor#<`, on envoie un mail.
— Avec `Ractor::recv`, on regarde le premier mail non lu (notons que l'adresse de l'expéditeur n'est pas disponible, même si elle peut bien sûr être envoyée par mail).
— Avec `Ractor::yield`, on écrit un résultat au tableau et on attend que quelqu'un vienne le lire. Tant que personne ne l'a lu, on ne peut pas continuer à travailler (le tableau est rempli).
— Avec `Ractor#take`, on va voir s'il y a quelque chose d'écrit au tableau de quelqu'un. Tant qu'il n'y a rien d'écrit, on attend qu'il écrive (on a besoin de ce résultat pour continuer notre travail), et quand on a enfin notre résultat, on efface le tableau pour que le collègue puisse poursuivre son travail.

Le dernier point en particulier est intéressant. Lorsque qu'un acteur `yield`, un seul acteur peut lire ce qui s'y passe, et si deux essaient, seul le premier l'aura et le second attendra qu'il réécrive quelque chose.

```
r1 = Ractor.new do
  r = Ractor.recv
  print "r1 récupère #{r.take}.\n"
end
r2 = Ractor.new do
  r = Ractor.yield 1
  sleep(1)
  r = Ractor.yield 2
end
r1 << r2
print "Mainteant r1 et main vont tous deux attendre r2.\n"
print "On récupère #{r2.take}.\n"
r1.take # On attend r1
```

L'acteur principal ou `r1` récupère le `1` et l'acteur restant récupère le `2` (après une seconde d'attente dans `r2`).

2.2. Un peu plus ?

Faisons un petit programme où deux acteurs vont jouer à s'envoyer des messages pour un ping-pong (le premier affiche «Ping» et dit à l'autre que c'est à son tour de jouer). L'acteur principal va envoyer l'acteur `pong` à l'acteur `ping` pour qu'il sache à qui envoyer les messages et de qui recevoir les messages.

```
ping = Ractor.new do
  pong = Ractor.recv
  nil while Ractor.recv != :Start
  loop do
    puts 'Ping'
    sleep(0.5)
    pong.send :Ping
    break if pong.take != :Pong
  end
end

pong = Ractor.new do
  loop do
    msg = Ractor.recv
    if msg == :Ping
      puts 'Pong'
      sleep(0.5)
      Ractor.yield :Pong
```

```
    end
  end
end

ping.send pong
ping.send :Start
gets
```

Nous obtenons un code plutôt simple (en tout cas bien plus simple qu'avec de simples threads). Nos acteurs `ping` et `pong` ne sont pas symétriques. L'acteur `ping` connaît `pong` alors que `pong` ne connaît pas celui qui lui envoie le message. En particulier, nous pouvons rajouter un acteur `ping_bis` et utiliser `pong` avec `ping` et `ping_bis` à la fois! Cela signifie également que n'importe quel acteur peut arriver au milieu de notre jeu et intercepter le message `:Pong` qui est envoyé par `pong`.

Une solution pour régler ce petit souci est de ne pas juste envoyer à `pong` le message `:Ping` mais également l'acteur courant pour qu'il sache à qui répondre.

Dans le code précédent, nous avons une boucle dans notre acteur. C'est en fait quelque chose qui a l'air assez raisonnable même dans d'autres cas. Plutôt que d'avoir un employé qui fait une action quand il reçoit un certain message puis s'arrête, pourquoi ne pas le faire boucler et faire cette action dès qu'il reçoit le message.

```
def a
  puts 'Je fais une action'
end

r = Ractor.new do
  a if Ractor.recv == :DoIt
end

r_loop = Ractor.new do
  loop do
    a if Ractor.recv == :DoIt
  end
end
```

Et on pourrait même faire cet employé particulier s'arrêter lorsqu'il reçoit un certain message (par exemple `:Break`). Faisons alors un employé qui nous permet, lorsqu'on lui donne un entier `n`, de savoir si `n` est premier.

```
require 'prime'

mathematician = Ractor.new do
  loop do
    msg = pipe.take
```

```
    break if msg == :Break
    Ractor.yield [msg, msg.prime?]
  end
end
```

On peut alors utiliser l'acteur plusieurs fois. Cela veut également dire que tant que l'acteur n'est pas arrêté, il y a un thread pour cet acteur. Pour notre exemple c'est un peu bête, mais par exemple, si on crée un serveur, on peut imaginer un acteur `listener` dont le travail est d'attendre les connexions entrantes (en gros, un employé dont le travail est d'accueillir et de rediriger vers les services compétents).

Nous aurions pu aller encore plus loin et à l'origine j'avais prévu de parler de l'utilisation de pipe comme queue qu'on peut voir dans plusieurs exemples de `Ractor`. Mais nous allons nous arrêter là, chacun pourra aller se renseigner pour en savoir plus. Je laisse cependant ce petit code.

```
require 'prime'

MAX_WORKERS = 5

pipe = Ractor.new do
  loop do
    Ractor.yield Ractor.recv
  end
end

workers = (1..MAX_WORKERS).each do
  Ractor.new(pipe) do |pipe|
    loop do
      n = pipe.take
      print "#{n} is #{n.prime? ? '' : 'not'} prime\n"
    end
  end
end


(10000...10020).each do |i|
  pipe << i
end
gets
```

Voici également quelques liens.

— L'article sur la sortie de Ruby 3.0.0 .
— La documentation de `Ractor` sur le dépôt Github de Ruby.
— Un article et un autre article , ça en fait deux pour le prix d'un.

Fonctionnalité expérimentale

S'il est sûr que la fonctionnalité restera, la spécification (et aussi l'implémentation) de `Ractor` a des chances de changer. Par exemple, il est probable que l'envoi de message avec `Ractor#send` soit supprimé puisque la méthode `send` correspond déjà à l'envoi de message au sens appel de méthode pour les autres objets.