Karnaj

Geste de savoir

Les Fibers de Ruby

10 mars 2021

Table des matières

1. Au début était la tâche 1
2. Puis il y eut la machine 4
3. Et enfin vinrent les arguments 6

Aujourd'hui, je me suis dit que j'allais m'amuser avec la classe `Fiber` et recréer l'exemple de restauration rapide dans cet article de @nohar. Et puisque plus il y a de fous, plus y a de riz, je me suis dit que j'allais en profiter pour essayer d'écrire un article pendant que je m'amuse.

Je vous recommande de lire cet article avant de lire ce billet pour savoir ce qu'est une coroutine et pour voir quel exemple nous allons recréer.

> **La *gem* `async`**
>
> Le but de ce billet est de s'amuser avec la classe `Fiber`, on n'utilisera pas la *gem* `async` ou d'autres *gems* utilisés pour l'asynchrone.

1. Au début était la tâche

Du côté de Ruby, il nous faut savoir ça.

- Les coroutines se créent avec la classe `Fiber` qui prend en paramètre un bloc avec le code de la coroutine (ce bloc peut prendre des paramètres à passer à la coroutine).
- On démarre une coroutine `f` avec `f.resume(args)`.
- Dans une coroutine, on redonne la main avec `Fiber.yield(ret)` qui retourne `ret`.
- Une exception `FiberError` est lancée si le code d'une coroutine est finie.
- On peut savoir si une coroutine est terminée avec `Fiber#alive?` (qui requiert `'fiber'`).

```ruby
require 'fiber'

f = Fiber.new do |arg|
  (0...5).each do |x|
    puts x
    y = Fiber.yield(x)
    puts "On redémarre avec #{y}"
  end
end

y = 0
```

```ruby
while f.alive?
  y = f.resume(2*y)
  puts "La coroutine nous donne #{y}."
end
```

Avec ça, on peut déjà faire un premier test de restauration rapide.

```ruby
steak = Fiber.new do
  puts 'On demande un steak en cuisine.'
  Fiber.yield
  puts 'On récupère le steak.'
end

soda = Fiber.new do
  puts 'On lance une machine à soda.'
  Fiber.yield
  puts 'On récupère le soda.'
end

while steak.alive? || soda.alive?
  soda.resume if soda.alive?
  steak.resume if steak.alive?
end
```

On va rajouter une fonction `asleep` pour faire du sommeil asynchrone. Une des premières choses à la quelle on peut penser pour ça, c'est récupérer le temps actuel, puis redonner la main quand on la coroutine est appelée tant que le temps n'est pas écoulé.

```ruby
def asleep(n)
  t = Time.now
  Fiber.yield while Time.now < t + n
  yield if block_given?
  Fiber.yield
end

steak = Fiber.new do
  puts 'On demande un steak en cuisine.'
  asleep(3)
  puts 'On récupère le steak.'
end

soda = Fiber.new do
  puts 'On lance une machine à soda.'
  asleep(1)
  puts 'On récupère le soda.'
end
```

```ruby

while steak.alive? || soda.alive?
  soda.resume if soda.alive?
  steak.resume if steak.alive?
end
```

Ici il nous faut vérifier si nos coroutines sont vivantes et tout, encapsulons tout ça dans une classe `Task` (on s'inspire vraiment à fond de l'article de @nohar).

```ruby
class Task
  RUNNING = 1
  FINISHED = 0
  ERROR = -1
  NEW = 2

  attr_reader :status, :result

  def done?
    !@fiber.alive?
  end

  def initialize(&block)
    @fiber = Fiber.new(&block)
    @status = NEW
  end

  def run
    @result = @fiber.resume unless done?
    @status = (@fiber.alive? ? RUNNING : FINISHED)
  rescue StandardError => e
    @result = e
    @status = ERROR
  end
end

steak = Task.new do
  puts 'On demande un steak en cuisine.'
  asleep(3)
  puts 'On récupère le steak.'
end

soda = Task.new do
  puts 'On lance une machine à soda.'
  asleep(1)
  puts 'On récupère le soda.'
end
```

2. Puis il y eut la machine

Maintenant, on aimerait pouvoir avoir plusieurs steaks et plusieurs sodas et limiter leur fabrication. Pour ça, créons des machines. Une machine est associée à un bloc. Quand nous l'utilisons, cela crée une nouvelle tâche ajoutée à la liste des tâches en cours (s'il y a une place libre) ou à la liste des tâches en attente (une file d'attente que nous gérons avec la classe 'Queue').

```ruby
class Machine
  def initialize(max = Float::INFINITY, &block)
    @waiting = Queue.new
    @block = block
    @max = max
    @running = []
  end

   def free?
    @running.size < @max
  end

  def todo?
    !@waiting.empty?
  end

  def launch
    t = Task.new(&@block)
    free? ? @running << t : @waiting << t
  end

  def running?
    @running.size > 0
  end

  def run
    @running.each(&:run)
    @running.reject!(&:done?)
    @running << @waiting.pop while free? && todo?
  end
end
```

On peut alors créer une machine à soda et une machine à steak. Ce sont eux qui se chargent de lancer tout ce qui leur est lié.

```ruby
steak_machine = Machine.new(3) do
  puts 'On demande un steak en cuisine.'
  asleep(3)
  puts 'On récupère le steak.'
```

```ruby
end

soda_machine = Machine.new(2) do
  puts 'On lance une machine à soda.'
  asleep(1)
  puts 'On récupère le soda.'
end

10.times do
  steak_machine.launch
  soda_machine.launch
end

while soda_machine.running? || steak_machine.running?
  soda_machine.run
  steak_machine.run
end
```

On a un résultat assez satisfaisant. Mais on ne sait pas même pas quelle commande est finie, on sait juste qu'un soda ou un steak est prêt. Ce qu'il nous faut, ce sont des tâches pour les clients. Une tâche de service attendra que la commande soit prête. On commence par écrire une commande `await` qui permet d'attendre des tâches.

```ruby
def await(*tasks)
  Fiber.yield while tasks.any? { |t| !t.done? }
  Fiber.yield
end
```

Et maintenant on peut créer notre service client.

```ruby
service = Machine.new do
  puts 'On lance une commande.'
  steak = steak_machine.launch
  soda = soda_machine.launch
  await(soda, steak)
  puts 'Une commande est prête.'
end
```

Ça a nécessité la modification de la commande `launch` pour renvoyer la tâche. On se rend alors compte que si on a accès à la tâche, on peut la lancer en dehors de la machine correspondante (et donc sans vérifier qu'elle est libre). Par exemple, rien ne nous empêche d'appeler la méthode `run` de la tâche qu'on a récupéré dans `soda`. Voici deux pistes pour régler cela.

— Ne pas renvoyer une tâche, mais un objet par exemple un `Machine::Task` qui n'a pas de méthode `run` et qui encapsule une `Task` qui n'est pas visible.
— Ajouter un attribut `launchable` à une tâche qui est vérifié avant de lancer une tâche.

Mais nous n'allons pas nous en occuper et allons juste renvoyer `t` dans la méthode `launch` de `Machine`.

```
class Machine
  def launch
    t = Task.new(&@block)
    free? ? @running << t : @waiting << t
    t
  end
end
```

Ici, on considère une machine de services qu'il nous faut également la `run` dans notre boucle. Par contre, il nous suffit d'attendre que `service` soit terminé, pas besoin d'attendre les autres machines.

```
4.times { service.launch }
while service.running?
  soda_machine.run
  steak_machine.run
  service.run
end
```

3. Et enfin vinrent les arguments

On est proche d'un résultat complet. On aimerait bien savoir quelle commande est prête, etc. Pour ça, on va rajouter des arguments à nos tâches. On va gérer ceci avec les arguments du bloc donné à `Fiber`. On stocke donc les arguments dans la tâche, et quand on lui redonne la main, on les utilise.

```
class Task
  def initialize(*args, &block)
    @fiber = Fiber.new(&block)
    @status = NEW
    @args = args
    print @args
  end

  def run
    @result = @fiber.resume(*@args) unless done?
    @status = (@fiber.alive? ? RUNNING : FINISHED)
  rescue StandardError => e
    @result = e
    @status = ERROR
```

```ruby
  end
end

class Machine
  def launch(*args)
    t = Task.new(*args, &@block)
    free? ? @running << t : @waiting << t
    t
  end
end

steak_machine = Machine.new(3) do |name|
  puts 'On demande un steak en cuisine.'
  asleep(3)
  puts "On récupère le steak de #{name}."
end

soda_machine = Machine.new(2) do |name|
  puts 'On lance une machine à soda.'
  asleep(1)
  puts "On récupère le soda de #{name}."
end

service = Machine.new do |name|
  puts "On lance la commande de #{name}. "
  steak = steak_machine.launch(name)
  soda = soda_machine.launch(name)
  await(soda, steak)
  puts "La commande de #{name} est prête."
end

('A'..'E').each { |client| service.launch(client) }
while service.running?
  soda_machine.run
  steak_machine.run
  service.run
end
```

Finalement, voici le code final. On a rajouté à `await` et à `asleep` l'exécution potentielle d'un bloc quand on a fini d'attendre.

```ruby
def asleep(n)
  t = Time.now
  Fiber.yield while Time.now < t + n
  yield if block_given?
  Fiber.yield
end

```

```ruby
def await(*tasks)
  Fiber.yield while tasks.any? { |t| !t.done? }
  yield if block_given?
  Fiber.yield
end
```

```ruby
class Task
  RUNNING = 1
  FINISHED = 0
  ERROR = -1
  NEW = 2

  attr_reader :status, :result

  def initialize(*args, &block)
    @fiber = Fiber.new(&block)
    @status = NEW
    @args = args
  end

  def run
    @result = @fiber.resume(*@args) unless done?
    @status = (@fiber.alive? ? RUNNING : FINISHED)
  rescue StandardError => e
    @result = e
    @status = ERROR
  end

  def done?
    !@fiber.alive?
  end
end
```

Listing 1 – La classe `Task`.

```ruby
class Machine
  def initialize(max = Float::INFINITY, &block)
    @waiting = Queue.new
    @block = block
    @max = max
    @running = []
  end

   def free?
    @running.size < @max
  end
```

```ruby

  def todo?
    !@waiting.empty?
  end

  def launch(*args)
    t = Task.new(*args, &@block)
    free? ? @running << t : @waiting << t
    t
  end

  def running?
    @running.size > 0
  end

  def run
    @running.each(&:run)
    @running.reject!(&:done?)
    @running << @waiting.pop while free? && todo?
  end
end
```

Listing 2 – La classe `Machine`.

```ruby
steak_machine = Machine.new(3) do |name|
  puts 'On demande un steak en cuisine.'
  asleep(3) { puts "Le steak de #{name} est prêt !"}
  puts "On récupère le steak de #{name}."
end

soda_machine = Machine.new(2) do |name|
  puts 'On lance une machine à soda.'
  asleep(1)
  puts "On récupère le soda de #{name}."
end

service = Machine.new do |name|
  puts "On lance la commande de #{name}. "
  steak = steak_machine.launch(name)
  soda = soda_machine.launch(name)
  await(soda, steak)
  puts "La commande de #{name} est prête."
end

('A'..'Z').each { |client| service.launch(client) }
while service.running?
  soda_machine.run
  steak_machine.run
```

```
25    service.run
26  end
```

Listing 3 – Le code principal.

On a utilisé le bloc de `asleep` dans la `steak_machine` pour afficher que le steak est prêt (on peut y penser comme à un minuteur ou à un cuisinier qui dit ce qui est prêt).

Maintenant voici quelques éléments pour aller plus loin.

- Gérer différents menus. Pour ça, on peut donner en argument à la `Machine` de service les différentes machines à appeler pour le menu à créer. Ainsi, `service` prendrait en argument le nom du client et ce qu'il veut commander (les machines à appeler pour avoir ce qu'il veut).
- Pour le moment, on rajoute tous nos clients et ensuite on gère leur commande. Pourquoi ne pas accueillir les clients petits à petit dans la boucle principale. On pourrait avoir une méthode basée sur un peu de hasard et accueillir un nouveau client quand elle renvoie `true` (et plus le dernier client est arrivé depuis longtemps, plus y a de chances qu'il arrive ou plus c'est l'heure du repas, plus y a de chances qu'il arrive).
- En se basant un peu sur le point précédent, on pourrait avoir des bornes pour prendre les commandes. Il y aurait alors un nombre limité de bornes (un service avec un `max` non illimité). On dira alors par exemple qu'une tâche de commande prend un certain temps (un petit appel à `await`). Attention, dans ce cas il nous faudra une machine pour les commandes différentes de la machine `Service`: un client qui n'est pas encore servi ne bloque pas une borne en empêchant un autre client de commander.
- Et finalement, il nous faudrait nous débarrasser de notre boucle `while`. Un ordonnanceur à qui on donne les tâches à exécuter en parallèle (`soda_machine`, `steak_machine`, `service` et potentiellement `order_machine`) et qui les exécute. Et il aurait des commandes pour le lancer avec des conditions d'arrêts; toutes les tâches sont finies, ou encore une tâche est finie.

Voilà, c'est fini; on a tant ressassé les mêmes théories , mais c'est un très bel exemple procuré par @nohar. C'était amusant à écrire ~~et ça a pris mon après-midi~~.