Écrire un moteur de tâches en Ruby

Julien Kirch

2020-10-12

Table des matières

Partie 1 : introduction

Un moteur de tâches est une brique logicielle dont le rôle est d'exécuter de manière asynchrone des morceaux de code qu'on lui fournit qu'on appelle tâches. On l'alimente en tâches (par exemple via une file de messages) et il les consomme, en général aussi vite que possible.

Vous avez par exemple Quartz en Java ou Sidekiq en Ruby.

Ils sont généralement très simple à utiliser, surtout si on s'en tient aux fonctionnalités standard, mais ce n'est pas le cas de leur fonctionnement interne.

En effet un moteur de tâches se doit (en principe) d'être efficace via du parallélisme, de ne pas perdre de données, de fournir une gestion d'erreur qui permette d'investiguer les problèmes et un tas d'autres choses.

Ces caractéristiques ne sont pas propres aux moteurs de tâches, au contraire on les retrouve, ou on est sensé les retrouver, dans la majorité des logiciels serveurs.

Aucune d'entre elle ne requiert de code particulièrement complexe, du moins tant qu'on choisi de s'appuyer sur des outils existants comme des moteurs de bases de données.

Il s'agit plutôt de s'appuyer sur des patterns bien choisis et éprouvés. Cet article illustre cette situation dans le domaine de la gestion d'erreur.

Cela signifie que bien comprendre le fonctionnement d'un moteur de tâches permet de comprendre le fonctionnement d'autres logiciels qui ont les mêmes caractéristiques.

Pour cela je vous propose donc de décrire pas à pas la construction de ce type d'outil. À chaque étape, je vais expliquer le besoin auquel répond chaque nouvel élément, et la manière d'y répondre.

Si l'outil résultant ne sera pas aussi complet qu'un "vrai" moteur de tâches, il en comportera tous les éléments essentiels.

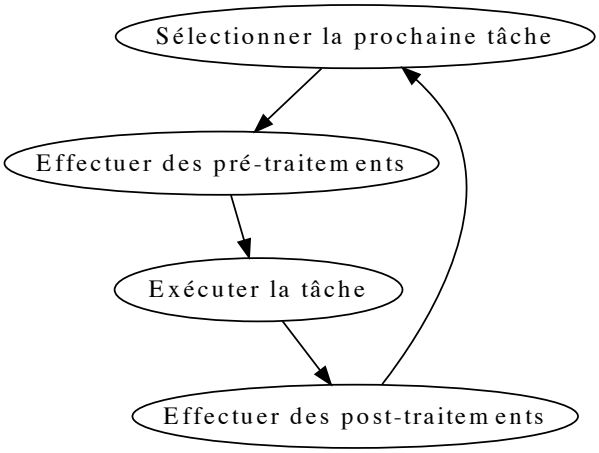
Il est temps de passer aux choses sérieuses, et je vais commencer par construire la boucle de traitement qui forme le centre de l'outil.

Le code source complet de l'outil est disponible en annexe, et il est également disponible en ligne ici.

Partie 2 : la boucle de traitement

La boucle de traitement

Le centre d'un moteur de tâches c'est une boucle qui fait quatre choses :



1. Sélectionner la prochaine tâche à exécuter
2. Effectuer des pré-traitements (par exemple loguer)
3. Exécuter la tâche
4. Effectuer des post-traitements (par exemple loguer et faire des traitements spécifiques en cas d'erreur)

Les composants du système qui effectuent cette boucle sont généralement appelés des "workers".

Augmenter la capacité de traitement

Un moteur de tâches passe la très grande majorité de son temps à exécuter les tâches.

Une fois que les tâches ont été correctement optimisées, augmenter la capacité de traitement nécessite de pouvoir lancer des tâches supplémentaires, et donc ajouter des workers.

Cela signifie que les phases en dehors de celle d'exécution de la tâche doivent pouvoir être parallélisées le plus possible. Cela passe principalement par éviter ou limiter les moment où les différents workers peuvent interférer les uns avec les autres.

Si les tâches à exécuter ont elles-même des problèmes de contention, cela va limiter la parallélisation de tout le système, mais c'est est en dehors du périmètre du moteur de tâches en lui-même.

Tout ce qu'il à faire pendant l'exécution c'est de ne pas être une gène.

Ajouter des workers peut se faire de deux manières : en lançant plusieurs workers dans un même processus et en lançant plusieurs processus.

La première approche s'appuie souvent sur des threads, qui est un mécanisme fournit par la plupart des systèmes d'exploitations. Elle a l'avantage de limiter la consommation mémoire car les différents threads peuvent partager des ressources, par exemple la mémoire dans laquelle le code qui s'exécute est chargé.

Utiliser plusieurs processus évite d'être limité par la puissance d'une machine, en effets les différents processus peuvent être lancés sur des serveurs différents. Par contre elle transforme le moteur de tâches en système distribué, et subit par conséquents les contraintes de ce type de programmes.

Les deux options sont complémentaires : on peut avoir plusieurs processus hébergeant chacun plusieurs threads.

L'idéal est d'avoir un système qui sache tirer partie des threads pour bénéficier de l'économie de ressources, tout en fonctionnant bien en multi-process quand ça devient nécessaire.

Threading et Ruby

Les thread en Ruby ont une spécificité : l'implémentation Ruby de référence (probablement celle que vous utilisez si vous n'avez pas choisi spécifiquement d'en utiliser une autre) ne permet pas plusieurs thread de s'exécuter en même temps, à cause de limitations dans la manière dont elle est implémentée. Si vous utilisez un moteur de tâches en Ruby et que vous voulez pleinement profiter d'avoir plusieurs threads, il faut ainsi utiliser des implémentation alternatives comme JRuby.

Cela ne signifie pas qu'utiliser plusieurs threads en Ruby soit inutile dans un moteur de tâches avec l'implémentation de référence. Car même si un seul thread s'exécute à la fois, plusieurs tâches peuvent tout de même être en train de progresser. C'est le cas parce que lorsqu'un appel extérieur est fait, par exemple une requête en base de données ou un appel réseau, le

thread qui fait l'appel est mis en pause et un autre thread peut alors s'exécuter, pendant que l'appel est en train de se faire. Lorsqu'il se termine, le premier thread va alors attendre son tour et continuer son traitement.

Dans un système ou la majorité du temps est passé à attendre le reste du monde, avoir plusieurs threads peut donc être tout de même assez avantageux. À l'inverse si les traitements demandent des traitements longs dans le moteur de tâches, les threads sont pratiquement inutiles et faudra alors s'appuyer uniquement sur des processus ou utiliser une autre machine virtuelle.

Démarrer le projet

Je vais mettre l'ensemble du code (framework et code d'exemple) dans un projet unique et je ne le packagerai pas sous forme d'une gem.

Le faire ajouterait de la configuration et du code et donc un peu de complexité sans que cela apporte quelque chose aux sujets dont je veux discuter ici.

Je commence par définir la version de Ruby à utiliser

.ruby-version.

```
2.7.1
```

et ma dépendance de départ

Gemfile.

```
source 'https://rubygems.org'

gem 'rake', '~> 13.0'
```

Pour montrer comment fonctionnent le threading, la première implémentation des tâches se contentera d'attendre quelques secondes via la méthode `Kernel#sleep` ce qui permet aux autres threads de s'exécuter. Et pour le moment elle n'iront pas chercher quelles tâches sont disponibles, mais s'exécuteront de manière continue dans une boucle infinie.

La version simplifiée d'un worker se résume alors à :

task_engine.rb.

```
Thread.new do
  while true
    execute
  end
```

```ruby
def execute
  p 'Task is starting'
  sleep(5)
  p 'Task is stopping'
end
```

Le thread principal du moteur de tâches va démarrer les différent workers puis se mettre en pause pour leur permettre s'exécuter.

Pour vérifier que cela fonctionne, il est nécessaire de lancer plusieurs workers, et d'ajouter des logs.

Je vais placer le code des workers dans un classe spécifique `TaskEngine::Worker`, et une autre classe `TaskEngine::Engine` sera chargée de les lancer, c'est elle qui sera exécutée au démarrage du moteur de tâches.

Chaque instance de worker reçoit un numéro de façon à pouvoir suivre ce qu'elle fait dans les logs.

task_engine.rb.

```ruby
require 'date'
require 'logger'

module TaskEngine

  WORKERS_NUMBER = 5
  MILLISECONDS_IN_A_DAY = 24 * 60 * 60 * 1000
  LOGGER = Logger.new(STDOUT)

  class Engine
    def initialize
      LOGGER.info('Starting engine')
      0.upto(WORKERS_NUMBER - 1) do |worker_index|
        Worker.new(worker_index)
      end
      sleep
    end
  end

  class Worker

    # @param [Integer] worker_index
    def initialize(worker_index)
      @worker_index = worker_index
      LOGGER.info("Starting worker #{worker_index}")
      Thread.new do
        while true
          execute
        end
```

```
      end
    end

    private

    def execute
      starting_time = DateTime.now
      LOGGER.info("Worker #{@worker_index} is starting")
      sleep(5)
      stopping_time = DateTime.now
      # The difference between two DateTimes is a Rational
      # representing the value as a number of days
      elapsed_time = ((stopping_time - starting_time) * MILLISECONDS_IN_A_DAY).to_f
      LOGGER.info("Worker #{@worker_index} is stopping, took #{elapsed_time}ms")
    end
  end

end
```

Le nombre de workers est défini dans une constante, passer par un fichier de configuration ajouterait du code sans changer l'idée.

Il me reste à écrire une tâche Rake pour pouvoir démarrer le moteur :

Rakefile.

```
desc 'Start the engine'
task :start_engine do
  require_relative 'task_engine'
  TaskEngine::Engine.new
end

task default: :start_engine
```

Et je peux vérifier que tout fonctionne comme prévu :

```
$ rake
Starting engine
Starting worker 0
Starting worker 1
Starting worker 2
Starting worker 3
Starting worker 4
Worker 0 is starting
Worker 2 is starting
Worker 1 is starting
Worker 3 is starting
Worker 4 is starting
Worker 0 is stopping, took 5006.025ms
Worker 0 is starting
Worker 1 is stopping, took 5007.419ms
```

```
Worker 1 is starting
Worker 4 is stopping, took 5006.673ms
Worker 3 is stopping, took 5006.849ms
Worker 3 is starting
Worker 4 is starting
Worker 2 is stopping, took 5006.9259999999995ms
Worker 2 is starting
Worker 4 is stopping, took 5002.389ms
Worker 4 is starting
Worker 1 is stopping, took 5003.09ms
Worker 1 is starting
Worker 0 is stopping, took 5003.359ms
Worker 0 is starting
Worker 2 is stopping, took 5002.508ms
Worker 2 is starting
Worker 3 is stopping, took 5002.456ms
Worker 3 is starting
```

Les tâches se bien lancent les unes après les autres, et se passent effectivement la main après avoir démarré une fois qu'elles atteignent le `sleep`. La boucle de traitement fonctionne donc bien, ou du moins sa version simplifiée.

Arrêter le programme stoppe immédiatement toutes les tâches en cours d'exécution.

Dans la partie suivante je vais m'intéresser au stockage des tâches pour que les workers aillent chercher ce qu'ils ont à faire.

Partie 3 : le début de la persistance

Quoi et comment stocker ?

Un moteur de tâches a besoin de stocker deux types de données : les tâches qui ne sont pas encore lancées et les logs de tâches exécutées.

Pour les tâches qui ne sont pas encore lancées, l'idéal est d'avoir un système de stockage :

1. fiable (pour ne pas perdre de tâches même en cas de crash),
2. qui gère la parallélisation (pour qu'il soit possible d'avoir un nombre importants de workers, et pouvoir faire en sorte qu'une tâche ne soit prise en compte que par *un seul* worker),
3. qui gère des notification pour être prévenu quand des tâches sont disponibles pour être exécutées (ce besoin peut être couvert en exécutant régulièrement du code qui vérifie si des tâches sont en attente, mais un système de notification est en général plus rapide et plus économe en ressources).

Pour les logs de tâches exécutées, les contraintes sont plus faibles (même si on préfère tout de même éviter de perdre des données et que le système ne soit pas trop lent), on a surtout besoin d'un système qui permette de la recherche pour faciliter les investigations.

Il est possible d'utiliser deux systèmes différents pour les deux usages, par exemple pour les tâches pas encore lancées il est courant d'utiliser un bus de messages, car en plus du stockage il peut servir de médiateur entre un système qui créé des tâches et le moteur de tâches en fournissant une API facile à utiliser.

Il est également possible d'utiliser un seul système, par exemple une base de données SQL ou même une base de données comme Redis.

Les technologies du projet

J'ai choisi d'utiliser PostgreSQL pour l'ensemble du stockage : le fonctionnement des transactions des bases de données SQL sera utile pour éviter la perte de données et gérer la paralléli-

sation, et PostgreSQL fournit un système de notification permettant de répondre au troisième point. Pour la recherche, les capacités d'indexation SQL seront bien suffisantes.

Pour l'accès aux données je vais utiliser un ORM. Plutôt qu'ActiveRecord qui amène avec lui tout l'écosystème Rails, je vais utiliser Sequel.

Sequel n'est pas aussi complet qu'ActiveRecord mais il a trois avantages :

- les API se ressemble pas mal
- il embarque beaucoup moins de code
- il permet de maîtriser le SQL généré de façon assez simple, alors qu'avec ActiveRecord c'est un peu moins naturel.

J'ajoute donc les dépendances au Gemfile :

Gemfile.

```
source 'https://rubygems.org'

gem 'rake', '~> 13.0'
gem 'sequel', '~> 5.34'
gem 'pg', '~> 1.2.3'
```

Et je créé la base de donnée avec un accès pour le compte de l'application :

```
$ createuser task_engine
$ createdb --owner=task_engine task_engine
```

Modéliser et créer les tâches

Pour commencer je vais m'intéresser aux tâches qui ne sont pas encore lancées.

Les tâches seront créées dans un état `waiting`, et passeront dans un état `running` lorsqu'un des worker du moteur de tâches les lancera. Les tâches les plus anciennes seront choisies d'abord. Après exécution, les tâches seront supprimées de la base (la gestion de l'historique sera faite plus tard).

Pour le moment je vais continuer d'utiliser les mêmes tâches qui se contentent de mettre le worker en attente, ce qui signifie que je n'ai pas à gérer de paramètres pour le moment.

Pour stocker l'état des tâches je vais utiliser un type énuméré (on dit aussi une énumération). Un peu comme dans les langages comme C ou Java, cela signifie que la base de données pourra vérifier que les valeurs qu'on lui fournit seront valides (en cas de valeur inconnue la

base renvoie une erreur), et que si dans le code Ruby et dans les requêtes on utilisera les valeur textuelles, en interne la base le stockera de manière efficace comme un nombre.

Voici la migration permettant de créer la table `tasks` (si vous avez l'habitude des migrations ActiveRecord, vous ne devriez pas être dépaysé·e) :

migrations/01_create_tasks.rb.

```
Sequel.migration do
  change do
    # Load enum-related methods
    extension :pg_enum

    # Declare the enum with the list of allowed values
    create_enum(:task_status, ['waiting', 'running'])

    create_table(:tasks) do
      primary_key :id
      # The status column is using the task_status enum type
      task_status :status, null: false, default: 'waiting'

      DateTime :created_at, null: false
      DateTime :updated_at, null: false
    end
  end
end
```

Je peux ensuite la lancer (le `-m` indique à Sequel où se trouve le répertoire qui contient les migrations) :

```
$ sequel -m migrations postgres://task_engine@localhost/task_engine
```

Et vérifier que la table a bien été créée :

```
$ psql --user=task_engine --dbname=task_engine --command="\d tasks"
                                      Table "public.tasks"
Column    |Type                         |Nullable|Default
----------+-----------------------------+--------+----------------------------
id        |integer                      |not null|generated by default as id
status    |task_status                  |not null|'waiting'::task_status
created_at|timestamp without time zone|not null|
updated_at|timestamp without time zone|not null|

Indexes:
    "tasks_pkey" PRIMARY KEY, btree (id)
```

Pour pouvoir manipuler les tâches il me reste à déclarer la connexion à la base et le modèle qui représente une tâche.

Lors de la configuration de l'accès à la base de donnée, j'indiquerai d'utiliser au maximum

une connexion par worker plus une pour le moteur. Cela permet que chaque worker ait une connexion à lui plutôt que de risquer que les workers ne s'attendent pour pouvoir faire des requêtes.

task_engine.rb.

```ruby
require 'sequel'

module TaskEngine

  # Database connexion path
  DATABASE_URL = 'postgres://task_engine@localhost/task_engine'
  # Open the connexions to the database
  # Use at most one connexion per worker
  DB = ::Sequel.connect(DATABASE_URL, max_connexions: WORKERS_NUMBER, logger: LOGGER)

  # The `created_at` and `updated_at` attributes should be managed by Sequel
  Sequel::Model.plugin :timestamps, force: true, update_on_create: true

  class Task < Sequel::Model
    STATUS_WAITING = 'waiting'
    STATUS_RUNNING = 'running'
  end
```

Je peux alors définir une tâche Rake pour créer 100 tâches à exécuter :

Rakefile.

```ruby
desc 'Create tasks'
task :create_tasks do
  require_relative 'task_engine'
  100.times do
    # As tasks has no parameters and the default status is `waiting`
    # we don't need to pass any parameter
    TaskEngine::Task.create
  end
end
```

Et la lancer :

```
$ rake create_tasks
(0.000223s) BEGIN
(0.003092s) INSERT INTO "tasks" ("created_at", "updated_at") VALUES ('2020-07-05
    22:07:24.194778+0200', '2020-07-05 22:07:24.194778+0200') RETURNING *
(0.001687s) COMMIT
(0.000355s) BEGIN
(0.000634s) INSERT INTO "tasks" ("created_at", "updated_at") VALUES ('2020-07-05
    22:07:24.203529+0200', '2020-07-05 22:07:24.203529+0200') RETURNING *
(0.000531s) COMMIT
(0.000250s) BEGIN
```

```
(0.000543s) INSERT INTO "tasks" ("created_at", "updated_at") VALUES ('2020-07-05
    22:07:24.206371+0200', '2020-07-05 22:07:24.206371+0200') RETURNING *
(0.000468s) COMMIT
(0.000171s) BEGIN
(0.000351s) INSERT INTO "tasks" ("created_at", "updated_at") VALUES ('2020-07-05
    22:07:24.209080+0200', '2020-07-05 22:07:24.209080+0200') RETURNING *
(0.000523s) COMMIT
(0.000152s) BEGIN
(0.000324s) INSERT INTO "tasks" ("created_at", "updated_at") VALUES ('2020-07-05
    22:07:24.211112+0200', '2020-07-05 22:07:24.211112+0200') RETURNING *
```

Ne pas perdre de tâche

Il reste la part du lion qui est de mettre à jour le moteur pour qu'il s'appuie sur la base de données.

La partie la plus complexe est l'acquisition de tâches à exécuter. Pour cela il faut sélectionner la tâche la plus ancienne, et changer son statut.

Mais comme plusieurs workers travaillent en parallèle, il faut faire en sorte que la même tâche ne soit pas prise en compte deux fois par deux workers.

Une chose importante dans le design des tâches : il est souhaitable — tant que cela est possible — qu'une même tâche puisse être exécutée plusieurs fois de suite sans que cela soit gênant.

Par exemple si une tâche est chargée d'envoyer un mail, si on l'exécute de nouveau il faudrait que le même mail ne soit pas envoyé une deuxième fois.

Cela vient du fait qu'il est très difficile de réunir deux conditions à la fois :

- faire en sorte que le moteur de tâches ne perde pas de tâche, pour faire en sorte que chaque tâche soit exécutée au moins une fois.
- faire en sorte que le moteur de tâches n'exécute pas la même tâche deux fois, même en cas d'erreur ou de problème réseau où on se fait pas forcément si une exécution qu'on a demandé a vraiment en lieu.

Dans les outils de messages, on appelle cela "*at least once*" (au moins une fois), "*at most once*" (au plus une fois) et "*exactly once*".

Le moteur de tâches se contentera de faire en sorte que la même tâche ne soit pas exécutée deux fois.

Plutôt que d'essayer de faire en sorte d'avoir un système qui garantit du "*exactly once*", on préfère que le moteur de tâches se concentre sur le fait de ne pas perdre de tâche, et si une

même tâche est exécutée une deuxième fois c'est la tâche elle-même qui est chargée que cela n'ait pas d'effet gênant.

Cela permet d'obtenir le même objectif (que le traitement final se comporte comme s'il avait été exécuté une fois), tout en modifiant les contraintes : au lieu d'avoir toutes les contraintes au niveau du moteur de tâches, elles sont en partie au niveau du moteur de tâches et en partie au niveau des tâches en elles-mêmes.

L'autre avantage de cette approche est qu'en cas de gros problème, il devient possible de réinjecter dans le moteur de tâches des tâches passées et de les laisser les traiter, sans avoir à se préoccuper de celles qui avaient déjà été exécutées.

Verrou et transaction

Revenons à nos moutons : sélectionner la tâche la plus ancienne et changer son statut, et empêcher qu'un autre thread ne fasse pareil.

En SQL pour cela on se sert de verrous (ou *locks*). Lorsqu'un thread va sélectionner la tâche T1 à exécuter il va en même temps verrouiller cette tâche dans la base, de sorte que les autres threads qui cherchent aussi la tâche la plus ancienne et qui pourraient vouloir utiliser T1 ne puissent le faire. Cela signifie que les autres workers seront bloqués en attendant que le verrou sur T1 soit relâché. Après avoir mis à jour le statut de T1, on la déverrouillera dès que possible pour débloquer les autres threads. À leur tour ils pourront chercher la tâche la plus ancienne à exécuter, qui sera donc la tâche suivante car le statut de T1 aura changé et elle ne sera donc plus sélectionnable pour eux.

Pour cela on va utiliser la commande `SELECT …FOR UPDATE`, qui verrouille les lignes sélectionnée en vue de pouvoir faire une mise à jour.

Le relâchement du verrou se fait à la fin de la transaction en cours, qui doit donc être terminée dès que possible pour limiter la période pendant laquelle les autres workers risquent d'être en attente.

L'acquisition de la tâche (sélection et mise à jour du statut) doivent de toutes façon se faire dans une transaction séparée de celle de l'exécution de la tâche :

- pour l'acquisition soit immédiatement visible dans la base ;
- pour qu'en cas d'erreur dans le code de la tâche qui déclencherait un `rollback`, l'acquisition ne soit pas automatiquement annulée, ce qui rendrait plus difficile la reprise.

Voici donc à quoi ressemble le code :

task_engine.rb.

```
# @return [TaskEngine::Task, nil]
def try_acquire_task
  # The transaction start here and will end after the end of the block
  DB.transaction do
    task = Task
      .where(status: Task::STATUS_WAITING)
      .order(:created_at)
      .for_update
      .first
    unless task.nil?
      Task
        .where(id: task.id)
        .update(status: Task::STATUS_RUNNING)
      task
    end
  end
end
```

La méthode `try_acquire_task` retournera la tâche à exécuter ou `nil` si aucune tâche n'est disponible.

Lorsque l'exécution se termine, pas besoin de poser de verrou car il n'y a pas de risque qu'un autre worker veuille terminer la même tâche. Par contre il est toujours nécessaire d'isoler le code dans une transaction à part, notamment pour qu'une erreur dans le moteur de tâches ne risque pas d'avoir de conséquence sur la tâche qui vient d'être exécutée.

task_engine.rb.

```
# @param [TaskEngine::Task] task
def end_task(task)
  DB.transaction do
    Task.where(id: task.id).delete
  end
end
end
```

La nouvelle boucle

Reste ensuite à modifier `TaskEngine::Worker#exécute` pour utiliser ces deux méthodes :

task_engine.rb.

```
def execute
  while (task = try_acquire_task)
    starting_time = DateTime.now
```

```
    LOGGER.info("Worker #{@worker_index} starting task #{task.id}")
    sleep(5)
    stopping_time = DateTime.now
    elapsed_time = (stopping_time - starting_time).to_f * MILLISECONDS_IN_A_DAY
    LOGGER.info("Worker #{@worker_index} finished task #{task.id}, took #{
        elapsed_time}ms")
    end_task(task)
  end
  LOGGER.info("Worker #{@worker_index} is stopping")
end
```

Tant qu'il reste des tâches, chaque worker les traitera les unes après les autres :

```
$ rake
Starting engine
Starting worker 0
Starting worker 1
Starting worker 2
Starting worker 3
Starting worker 4
(0.000731s) BEGIN
(0.001842s) SELECT "id" FROM "tasks" WHERE ("status" = 'waiting') ORDER BY "
    created_at" LIMIT 1 FOR UPDATE
(0.000392s) UPDATE "tasks" SET "status" = 'running' WHERE ("id" = 701)
(0.000818s) COMMIT
Worker 0 starting task 701
(0.000286s) BEGIN
(0.000582s) BEGIN
(0.000642s) BEGIN
(0.004352s) SELECT "id" FROM "tasks" WHERE ("status" = 'waiting') ORDER BY "
    created_at" LIMIT 1 FOR UPDATE
(0.001839s) UPDATE "tasks" SET "status" = 'running' WHERE ("id" = 702)
(0.000183s) BEGIN
(0.001389s) COMMIT
Worker 2 starting task 702
(0.008252s) SELECT "id" FROM "tasks" WHERE ("status" = 'waiting') ORDER BY "
    created_at" LIMIT 1 FOR UPDATE
(0.000682s) UPDATE "tasks" SET "status" = 'running' WHERE ("id" = 703)
(0.000644s) COMMIT
Worker 4 starting task 703
(0.008829s) SELECT "id" FROM "tasks" WHERE ("status" = 'waiting') ORDER BY "
    created_at" LIMIT 1 FOR UPDATE
(0.000422s) UPDATE "tasks" SET "status" = 'running' WHERE ("id" = 704)
(0.000721s) COMMIT
Worker 3 starting task 704
(0.005337s) SELECT "id" FROM "tasks" WHERE ("status" = 'waiting') ORDER BY "
    created_at" LIMIT 1 FOR UPDATE
(0.000946s) UPDATE "tasks" SET "status" = 'running' WHERE ("id" = 705)
(0.000791s) COMMIT
Worker 1 starting task 705
Worker 0 finished task 701, took 5002.389ms
```

```
(0.000386s) BEGIN
(0.000475s) DELETE FROM "tasks" WHERE ("id" = 701)
(0.000540s) COMMIT
```

À cette étape le noyau du système est là : on peut créer des tâches et quand on le lance le moteur va les exécuter jusqu'à épuisement.

Dans la partie suivante je vais m'intéresser à l'arrêt du moteur.

Partie 4 : l'arrêt

Pourquoi l'extinction est importante

Après l'avoir lancé, voici ce qui se passe quand on éteint le moteur de tâches en appuyant sur `CONTROL+C` :

```
Worker 2 starting task 844
(0.006916s) SELECT * FROM "tasks" WHERE ("status" = 'waiting') ORDER BY "created_at"
    LIMIT 1 FOR UPDATE
(0.000322s) UPDATE "tasks" SET "status" = 'running' WHERE ("id" = 845)
(0.000481s) COMMIT
Worker 1 starting task 845
^Crake aborted!
Interrupt:
/task-engine-ruby/task_engine.rb:35:in `sleep'
/task-engine-ruby/task_engine.rb:35:in `initialize'
/task-engine-ruby/Rakefile:4:in `new'
/task-engine-ruby/Rakefile:4:in `block in <top (required)>'
Tasks: TOP => default => start_engine
(See full trace by running task with --trace)
```

Le moteur de tâches s'arrête immédiatement et toutes les tâches s'arrêtent net.

Si on voulait utiliser ce type de fonctionnement dans un moteur de tâches en production, cela signifie que toutes les tâches doivent être développées de manière à ce qu'elles puissent être arrêté net n'importe quand.

Si les tâches n'utilisent que la base de données, cela peut être envisageable car alors la transaction est annulée, et quand on les relance elles recommencent de zéro.

Mais dès lors qu'elles communiquent avec l'extérieur les choses deviennent plus compliquées.

Par exemple si une tâches chargée d'envoyer un mail puis de mettre à jour la base de donnée pour indiquer que le mail est parti est arrêtée juste après l'envoi du mail.

Au redémarrage du moteur de tâches si la tâche est relancée elle enverrait un autre mail identique.

C'est pour cela qu'en général, lorsqu'un moteur de tâches reçoit une commande d'arrêt, au lieu de stopper les tâches en cours, il ne démarre plus de nouvelles tâches et attend pour s'éteindre que les tâches déjà lancées se terminent.

Cela signifie que dans les conditions normales d'opérations d'un arrêt/relance, les tâches peuvent s'appuyer sur l'hypothèse qu'elles ne seront pas stoppées en cours de route.

Je parle de conditions normale car un cas exceptionnel est toujours possible, de la même manière que n'importe quel programme, par exemple une machine virtuelle qui crashe. Cela signifie que le risque existe toujours, même s'il est faible.

Pour une tâche qui envoie un mail, il est peut-être acceptable qu'en cas de crash le mail soit envoyé deux fois, par contre pour une tâche qui fait un virement entre deux comptes bancaires, il vaut peut-être mieux se prémunir contre le risque que le virement soit effectué deux fois même si le risque de crash est faible.

Les signaux

Pour exécuter du code lorsqu'un programme s'arrête, il faut généralement indiquer au système qu'en cas de demande d'arrêt il doit exécuter un bloc de code ou une méthode spécifique plutôt que d'utiliser le comportement par défaut qui est de tout stopper d'un coup.

Sur les systèmes d'exploitations comme Linux ou macOS, la demande d'arrêt (par exemple lorsque vous appuyez sur `CONTROL+C`) est communiquée au programme sous forme d'un signal. Un programme peut informer le système d'exploitation que tel ou tel signal l'intéresse, et qu'il prend la responsabilité de répondre au signal, en remplacement du comportement par défaut.

Il existe différents signaux identifiés chacun par un nom et un numéro, certains sont normalisés et correspondent aux messages que le système avait besoin d'envoyer dans les systèmes UNIX classiques et d'autres sont dépendants du système d'exploitation.

Pour gérer une fin d'exécution, il faut en principe écouter deux signaux qui font partie de ceux qui sont normalisés :

- `SIGTERM` (pour *termination*) qui porte le numéro 15 et qui indique que le programme à qui on envoie le signal doit s'arrêter.
- `SIGINT` (pour *interrupt*) qui porte le numéro 2 et qui est envoyé lorsqu'on appuie sur `CONTROL+C` dans un terminal.

Le comportement est souvent le même pour les deux signaux.

En Ruby, pour définir un bloc de code à appeler quand un signal est reçu il faut appeler la méthode `Signal::trap` en lui passant en paramètre le nom ou le numéro du signal :

```
Signal.trap('SIGTERM') do
  # TODO: implement signal handling
end
```

À noter que le code appelé lors de la réception d'un signal n'est pas du code comme les autres car le système d'exploitation utilise des routines spécifiques pour appelle ce code et il s'exécute dans un contexte un peu spécial, et ce faisant certaines fonctionnalités qu'on tient normalement pour acquises sont indisponibles.

Cela signifie qu'il doit respecter certaines règles pour ne pas poser de problème au système d'exploitation et faite crasher l'application. Ainsi en théorie les allocation mémoire ou les entrées/sorties sont interdites.

Par exemple si le système d'exploitation décide d'arrêter un processus car sa mémoire est devenue insuffisante, si le code gérant l'arrêt essaie d'allouer de la mémoire supplémentaire cela ne fonctionnera pas.

En pratique il est possible que cela ne pose pas problème, même dans la très grande majorité des cas, mais vous n'avez aucune garantie.

Par exemple si vous essayer d'utiliser un log dans un code de gestion de signal :

```
require 'logger'

LOGGER = Logger.new(STDOUT)

Signal.trap('SIGTERM') do
  LOGGER.info('Received a SIGTERM signal')
end
```

Ruby vous indiquera que cela ne fonctionnera pas par un "log writing failed. can't be called from trap context", car pour éviter le risque que cet appel fasse crasher le programme, la machine virtuelle Ruby le désactive dans ce contexte.

La bonne pratique est donc de limiter au maximum ce que vous mettez dans le bloc de gestion de signal, par exemple d'uniquement changer la valeur d'une variable, et de faire en sorte que le vrai traitement se fasse en dehors du bloc.

Le signal d'arrêt dans le moteur de tâches

Dans le moteur de tâches, chaque worker exécute une boucle où il essaie de trouver une tâche à exécuter puis la lance :

task_engine.rb.

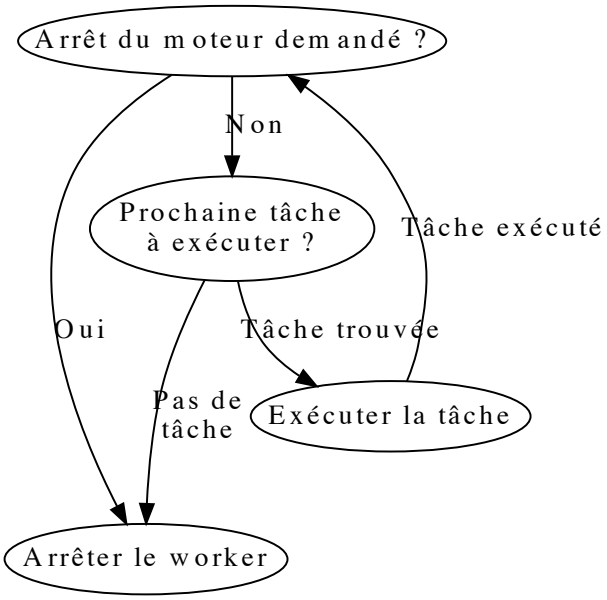
```ruby
def execute
  while (task = try_acquire_task)
    # …
  end
end
```

Pour gérer l'extinction, le bloc de gestion de signal va donc changer la valeur d'une variable, et à chaque tour de boucle le worker va vérifier s'il faut s'arrêter.

task_engine.rb.

```ruby
def execute
  while (status == RUNNING) && (task = try_acquire_task)
    # …
  end
end
```

Ce qui peut se traduire par :



Ce statut va être stocké dans l'instance de `TaskEngine::Engine`, auquel les workers vont accéder :

task_engine.rb.

```ruby
class Engine
  ENGINE_STATUS_RUNNING = 'running'
  ENGINE_STATUS_STOPPING = 'stopping'

  attr_reader :status
```

```
  def initialize
    @workers = []
    LOGGER.info('Starting engine')
    @status = ENGINE_STATUS_RUNNING
    Signal.trap('SIGTERM') do
      signal_trap
    end
    Signal.trap('SIGINT') do
      signal_trap
    end
    0.upto(WORKERS_NUMBER - 1) do |worker_index|
      Worker.new(self, worker_index)
    end
    sleep
  end

  def signal_trap
    @status = ENGINE_STATUS_STOPPING
  end
end

class Worker
  # @param [TaskEngine::Engine] engine
  # @param [Integer] worker_index
  def initialize(engine, worker_index)
    @engine = engine
    @worker_index = worker_index
    LOGGER.info("Starting worker #{worker_index}")
    @thread = Thread.new do
      execute
    end
  end

  def execute
    while (@engine.status == TaskEngine::Engine::ENGINE_STATUS_RUNNING) &&
        (task = try_acquire_task)
      starting_time = DateTime.now
      LOGGER.info("Worker #{@worker_index} starting task #{task.id}")
      sleep(5)
      stopping_time = DateTime.now
      elapsed_time = (stopping_time - starting_time).to_f * MILLISECONDS_IN_A_DAY
      LOGGER.info("Worker #{@worker_index} finished task #{task.id}, took #{
          elapsed_time}ms")
      end_task(task)
    end
    LOGGER.info("Worker #{@worker_index} is stopping")
  end
# …
```

Lorsque le thread du worker n'a plus rien à faire, il s'arrête :

```
Worker 2 starting task 850
^CWorker 0 finished task 846, took 5005.947ms
(0.000379s) BEGIN
(0.000607s) DELETE FROM "tasks" WHERE ("id" = 846)
(0.001182s) COMMIT
Worker 0 is stopping
Worker 3 finished task 847, took 5005.294ms
Worker 1 finished task 848, took 5002.687ms
(0.000278s) BEGIN
(0.000632s) BEGIN
(0.000560s) DELETE FROM "tasks" WHERE ("id" = 848)
(0.000462s) DELETE FROM "tasks" WHERE ("id" = 847)
(0.000455s) COMMIT
Worker 1 is stopping
(0.000555s) COMMIT
Worker 3 is stopping
Worker 4 finished task 849, took 5004.507ms
(0.000166s) BEGIN
(0.000324s) DELETE FROM "tasks" WHERE ("id" = 849)
Worker 2 finished task 850, took 5003.889ms
(0.000172s) BEGIN
(0.000671s) COMMIT
Worker 4 is stopping
(0.000354s) DELETE FROM "tasks" WHERE ("id" = 850)
(0.000449s) COMMIT
Worker 2 is stopping
```

S'arrêter vraiment

Après avoir appuyé sur sur `CONTROL+C`, les différents workers s'arrêtent après avoir terminé l'exécution de leurs tâches, mais le moteur de tâches lui ne s'arrête pas.

D'autres appuis sur `CONTROL+C` ne servent à rien.

Le problème c'est le thread principal qui est celui qui démarre du moteur.

Après avoir créé les différents worker, le thread principal se met en attente pour une durée indéfinie et ne s'arrête donc jamais, et il est de même pour le moteur de tâches.

On pourrait vouloir arrêter ce thread dans le code de traitement du signal, mais cela aurait pour effet d'arrêter immédiatement le programme, sans laisser aux workers la possibilité de terminer leurs traitements.

Il faut donc que le thread principal se termine après la fin de l'exécution de l'ensemble des tâches en cours, c'est à dire quand les threads correspondant aux différents workers se sont terminés.

Dans l'API des thread `Thread#join` permet justement de suspendre l'activité d'un thread en attendant qu'un autre thread se termine. L'idée est d'attendre les threads de tous les workers l'un après l'autre. Si un thread est déjà arrêté lorsqu'on appelle `Thread#join` sur lui, la méthode retourne immédiatement. Cela signifie qu'il n'y a pas à se préoccuper de savoir si un worker a déjà terminé ou pas avant de faire l'appel à `join`.

Lorsqu'on démarre les worker, il faut donc les stocker les différentes instances pour pouvoir ensuite accéder à leurs threads, et plutôt que d'appeler `Kernel#sleep`, le moteur attendra la fin du premier thread.

task_engine.rb.

```ruby
class Engine
  def initialize
    @workers = []
    LOGGER.info('Starting engine')
    @status = ENGINE_STATUS_RUNNING
    Signal.trap('SIGTERM') do
      @status = ENGINE_STATUS_STOPPING
    end
    Signal.trap('SIGINT') do
      @status = ENGINE_STATUS_STOPPING
    end
    0.upto(WORKERS_NUMBER - 1) do |worker_index|
      @workers << Worker.new(self, worker_index)
    end
    @workers.each do |worker|
      LOGGER.info("Joining worker #{worker.worker_index}")
      worker.thread.join
    end
  end
end

class Worker
  attr_reader :thread, :worker_index

  # @param [TaskEngine::Engine] engine
  # @param [Integer] worker_index
  def initialize(engine, worker_index)
    @engine = engine
    @worker_index = worker_index
    LOGGER.info("Starting worker #{worker_index}")
    @thread = Thread.new do
      execute
    end
  end
  # …
end
```

```
Starting engine
Starting worker 0
Starting worker 1
Starting worker 2
Starting worker 3
Starting worker 4
Joining worker 0

^CWorker 0 finished task 861, took 5005.397ms
(0.000375s) BEGIN
(0.000547s) DELETE FROM "tasks" WHERE ("id" = 861)
(0.001102s) COMMIT
Worker 0 is stopping
Joining worker 1
Worker 4 finished task 862, took 5002.765ms
Worker 1 finished task 863, took 5001.1179999999995ms
(0.000435s) BEGIN
(0.000862s) BEGIN
(0.000574s) DELETE FROM "tasks" WHERE ("id" = 862)
(0.000673s) DELETE FROM "tasks" WHERE ("id" = 863)
(0.000756s) COMMIT
Worker 4 is stopping
(0.000675s) COMMIT
Worker 1 is stopping
Joining worker 2
Worker 2 finished task 864, took 5004.611ms
(0.000432s) BEGIN
(0.000701s) DELETE FROM "tasks" WHERE ("id" = 864)
(0.000766s) COMMIT
Worker 2 is stopping
Joining worker 3
Worker 3 finished task 865, took 5005.013ms
(0.000240s) BEGIN
(0.000531s) DELETE FROM "tasks" WHERE ("id" = 865)
(0.000592s) COMMIT
Worker 3 is stopping
Joining worker 4
```

Après réception du signal, on voit que le thread de moteur attend successivement les différents worker pour ensuite arriver au bout de la méthode `TaskEngine::Engine#initialize`, ce qui stoppe le programme.

Cela règle le cas où le moteur de tâches s'arrête bien, je vais maintenant m'occuper de ce qui se passe après un arrêt d'urgence.

Partie 5 : le redémarrage

La théorie

Le cas de l'arrêt dans une situation normale est donc couvert. Reste celui de l'arrêt exceptionnel.

Cela peut correspondre à une machine virtuelle qui crashe, un bug qui fait planter la machine virtuelle Ruby. Cela peut aussi être une conséquence d'un arrêt normal qui ne s'est pas bien terminé.

En effet si une tâche reste bloquée, par exemple parce qu'un appel réseau n'a pas de timeout en cas de non réponse, une tâche peut rester bloquée indéfiniment, et donc empêcher le moteur de tâches de s'arrêter.

Pour le stopper, on peut utiliser un autre signal, typiquement `SIGKILL` (qui lui ne peut pas être intercepté par du code applicatif : c'est le système d'exploitation qui se charge de faire le ménage lui-même). Ce signal portant le numéro 9, c'est lui qui est envoyé quand on appelle `kill -9`.

Dans ce cas le programme s'arrête net et avec lui les tâches en cours d'exécution.

Avec l'implémentation actuelle du moteur de tâches, ces tâches restent définies comme étant dans l'état `running`. Par exemple voici le résultat obtenu en stoppant brutalement le code précédent où le thread principal ne s'arrêtait pas :

```
$ psql --user=task_engine --dbname=task_engine  --expanded --command="select * from
    tasks"

-[ RECORD 1 ]---+-------------------------------
id              | 211
status          | running
created_at      | 2020-08-10 16:52:50.219772
updated_at      | 2020-08-10 16:52:50.219772
-[ RECORD 2 ]---+-------------------------------
id              | 212
status          | waiting
created_at      | 2020-08-10 16:52:50.221219
updated_at      | 2020-08-10 16:52:50.221219
```

```
instance        | running
```

Cela signifie qu'il est difficile de différencier les tâches mal arrêtées de celles qui tournent réellement.

Il est possible d'aller chercher dans les logs pour déterminer les tâches qui ont été lancées et qui n'ont pas été terminées avant l'arrêt, mais c'est une approche artisanale alors que pour la gestion d'erreur je préfère des solutions les plus fiables possibles.

Pour identifier ces tâches, il faut pourvoir déterminer qu'elles appartiennent à l'instance de moteur de tâches qui a été arrêté de manière brutale. Une manière d'y parvenir est de stocker dans les tâches quelle instance du moteur de tâches est en train de les exécuter en ajoutant un identifiant.

Lorsque l'instance est plantée, on peut alors facilement identifier ces tâches. Au redémarrage, l'instance du moteur de tâches peut détecter qu'il existe des tâches marquée `running` qui portent son identifiant.

Cette approche aide aussi aux investigations pendant l'exécution d'une tâche : si une tâche est lente on peut aller consulter le monitoring de la machine sur laquelle elle est lancée pour voir si on voit quelque chose d'anormal.

Le plus simple, si un moteur de tâches démarre et trouve des tâches en statut `running` est alors de stopper immédiatement le moteur de tâches en indiquant explicitement quel est le problème.

Cette approche retarde la relance de du moteur de tâches, mais elle a deux avantages :

- elle force à traiter le problème (même si c'est en supprimant les tâches de la base de données, mais au moins une action volontaire a été faite)
- elle évite les mauvaises surprises dans le cas où deux instances de moteur de tâches sont lancées avec le même identifiant.

Par exemple, il serait aussi possible qu'au démarrage d'une instance I1 de moteur de tâches, toutes les tâches marquées comme `running` soit mise à jour dans un statut spécial, par exemple `crashed`. Mais imaginez que suite à une erreur de configuration une autre instance I2 de moteur de tâches utilise le même identifiant. Lors du démarrage de I1, les tâches qu'I2 est en train d'exécuter passeraient en statut `crashed`. Cela n'aurait peut-être aucune conséquence sur l'exécution de ces tâches, mais cela pourrait porter à confusion les personnes qui consulteraient l'état des tâches.

La solution de s'arrêter immédiatement évite donc ce risque.

La pratique

La mise en œuvre est moins longue que l'explication.

Tout d'abord il faut ajouter la colonne d' l'instance à la table `tasks` :

migrations/02_add_instance_name.rb.

```
Sequel.migration do
  change do
    alter_table(:tasks) do
      add_column :instance, String, null: true, text: true
    end
  end
end
```

Pour rappel, pour lancer les migrations, la commande est

```
sequel -m migrations postgres://task_engine@localhost/task_engine
```

Ensuite il faut récupérer le nom de l'instance dans la tâche Rake de lancement et la passer au moteur. Pour ce faire je vais utiliser une variable d'environnement :

Rakefile.

```
TASK_ENGINE_INSTANCE = 'TASK_ENGINE_INSTANCE'

task :start_engine do
  unless ENV.key?(TASK_ENGINE_INSTANCE)
    raise "Missing env variable #{TASK_ENGINE_INSTANCE}"
  end
  instance = ENV.fetch(TASK_ENGINE_INSTANCE)
  require_relative 'task_engine'
  TaskEngine::Engine.new(instance)
end
```

Je peux ensuite l'utiliser lorsqu'un worker récupère une tâche :

task_engine.rb.

```
class Engine
  ENGINE_STATUS_RUNNING = 'running'
  ENGINE_STATUS_STOPPING = 'stopping'

  attr_reader :instance, :status

  def initialize(instance)
    @instance = instance
    # …
  end
end
```

```
class Worker
  # @return [TaskEngine::Task, nil]
  def try_acquire_task
    DB.transaction do
      task = Task.where(status: Task::STATUS_WAITING).order(:created_at).for_update.
          first
      unless task.nil?
        Task.where(id: task.id).update(
            status: Task::STATUS_RUNNING,
            instance: @engine.instance
            )
        task
      end
    end
  end
end
```

Si je lance le moteur de tâches (en passant le nom de l'instance en variable d'environnement) et que je regarde ce qui se passe dans la base de données, on retrouve bien le nom de l'instance :

```
$ TASK_ENGINE_INSTANCE=instance_01 rake start_engine
```

```
$ psql --user=task_engine --dbname=task_engine  --expanded --command="select * from
    tasks"

-[ RECORD 1 ]---+-------------------------------
id              | 211
status          | running
created_at      | 2020-08-10 16:52:50.219772
updated_at      | 2020-08-10 16:52:50.219772
instance        | instance_01
-[ RECORD 2 ]---+-------------------------------
id              | 212
status          | running
created_at      | 2020-08-10 16:52:50.221219
updated_at      | 2020-08-10 16:52:50.221219
instance        | instance_01
```

Et pour terminer, au lancement du moteur de tâches, il faut vérifier si des tâches n'existent pas déjà avec le même nom d'instance :

task_engine.rb.

```
class Engine
  def initialize(instance)
    unless Task.where(
        status: ENGINE_STATUS_RUNNING,
        instance: instance).empty?
```

```
      raise "Found running tasks with same instance name in the database [#{instance
          }]"
    end
  # …
  end
end
```

On peut le tester en stoppant brutalement l'instance en train de se tourner et en la relançant :

```
$ TASK_ENGINE_INSTANCE=instance_01 rake start_engine
rake aborted!
Found running tasks with same instance name in the database [instance_01]
/task-engine-ruby/task_engine.rb:32:in `initialize'
/task-engine-ruby/Rakefile:11:in `new'
/task-engine-ruby/Rakefile:11:in `block in <top (required)>'
```

On peut ensuite affiner les choses, par exemple afficher la liste des tâches qui sont en cours d'exécution, ou fournir des méthodes pour faciliter la reprise, mais la partie importante est là.

Dans la partie suivante je vais m'intéresser au paramétrage des tâches.

Partie 6 : le paramétrage des tâches

Pour le moment le moteur de tâches ne sait exécuter qu'un seul type de tâches et n'accepte pas de paramètres, c'était suffisant jusque là mais pour la suite il va devenir nécessaire de le changer.

Définition de l'API des tâches

Le type de tâche sera défini par le nom de la classe de la tâche qu'il faut lancer, c'est ce que font tous les outils que je connais et je ne vois pas d'autre approche intéressante.

Les paramètres d'exécution des tâches sont sous forme d'une `Hash` qui est à la main des tâches.

Il y a plusieurs choix possibles pour le fonctionnement des tâches, par exemple :

Solution 1 : singletons

Les tâches peuvent être des singletons qu'on instancie une seule fois par moteur de tâches et sur lesquelles on appelle une méthode `execute`

```
class MyTask
  # Instance is a singleton managed by the engine
  def initialize()
  end

  # @param [Hash] parameters
  def execute(parameters)
  end
end
```

Solution 2 : **`execute`** avec paramètres

Les tâches sont instanciées à chaque exécution, pour cela on appelle un constructeur sans paramètre puis une méthode `execute` qui prend en paramètre les paramètres d'exécution

```ruby
class MyTask
  def initialize()
  end

  # @param [Hash] parameters
  # @return [Hash, nil]
  def execute(parameters)
  end
end
```

Solution 3 : **`execute`** sans paramètres

Les tâches sont instanciées à chaque exécution, pour cela on appelle un constructeur qui prend en paramètre les paramètres d'exécution puis une méthode `execute` sans paramètre

```ruby
class MyTask
  # @param [Hash] parameters
  def initialize(parameters)
  end

  # @return [Hash, nil]
  def execute()
  end
end
```

Solution 4 : tout dans le constructeur

Les tâches sont instanciées à chaque exécution, pour cela on appelle un constructeur qui prend en paramètre les paramètres d'exécution qui fait tout le traitement, il n'y a pas de méthode `execute`

```ruby
class MyTask
  # @param [Hash] parameters
  def initialize(parameters)
  end
end
```

Le choix

La solution 1 a une forme d'élégance mais elle a deux inconvénients :

- si le code de la tâche est un peu complexe et que des sous-méthodes deviennent nécessaires, le fait de ne pas pouvoir utiliser de membre d'instance oblige à passer plus de

choses en paramètres, ou de créer des classes ou des structures spécifiques

- elle empêche d'utiliser certains modules qui ajoutent des membres d'instances ou à nouveau elle oblige à créer des classes supplémentaires qui de fait vont remplacer la classe de la tâche

Le résultat c'est que le code va souvent ressembler à :

```
class MyTask
  def initialize
  end

  # @param [Hash] parameters
  def execute(parameters)
    MyRealTask.new(parameters)
  end

  class MyRealTask
    # @param [Hash] parameters
    def initialize(parameters)
    end
  end
end
```

Au final il vaut mieux laisser aux tâches la responsabilité d'utiliser elles-mêmes des singletons en interne si c'est nécessaire, et de permettre d'économiser une indirection dans le cas le plus fréquent.

Les autres solutions sont assez ressemblantes, et entre toutes je préfère la solution de "`execute` avec paramètres" qui a le numéro 2.

Le fait d'utiliser une méthode `execute` séparée du constructeur permet de récupérer des valeurs de retours qui peuvent être utiles pour le monitoring.

Et quitte à avoir une méthode `execute` séparée, autant qu'elle reçoive les paramètres, c'est un peu plus logique du point de vue objet, et cela permet de se passer du constructeur dans pas mal de cas.

Modification de schéma

Je vais ajouter les deux colonnes qui manquent à la table `tasks`. Je vais les définir comme non nullable pour éviter toute fausse manipulation. Si une tâche n'a aucun paramètre d'exécution, on utilisera une `Hash` vide plutôt d'un `nil`, cela évite l'ambiguïté entre aucun paramètre d'exécution et une valeur qui manque.

Cela signifie que la table doit être vide avant de lancer la migration, sinon elle échouera car la base n'acceptera pas d'ajouter des colonnes non nullable sans valeur par défaut.

Pour stocker les paramètres d'exécution en utilisant une colonne de type `jsonb` qui permet de stocker des données sous forme de JSON.

Par rapport à stocker le JSON dans un champ de texte, cela coûte un peu plus cher en temps d'insertion (car la base de données doit parser et valider le format des données à l'insertion), par contre cela peut simplifier les investigation car il est possible de d'accéder au contenu des valeurs depuis des requêtes.

migrations/03_add_parameters.rb.

```
Sequel.migration do
  change do
    run 'truncate tasks'
    alter_table(:tasks) do
      add_column :task_class, String, null: false, text: true
      add_column :task_parameters, 'JSONB', null: false
    end
  end
end
```

Définition d'une classe de tâche

Pour avoir une première définition de tâche je vais modifier le code la tâche existante pour utiliser la nouvelle API, et en profiter pour la déplacer dans un fichier séparé.

Pour mémoire, cette tâche se contentait d'attendre 5 secondes :

```
sleep(5)
```

Puisqu'on a maintenant des paramètres, je vais rendre la durée d'attente paramétrable. Pour cela je vais récupérer la valeur dans la `Hash` de paramètres :

tasks.rb.

```
module TaskEngine
  module Tasks
    class WaitingTask
      PARAMETER_WAITING_TIME = 'waiting_time'

      # @param [Hash] parameters
      # @return [Hash, nil]
      def execute(parameters)
        sleep(parameters[PARAMETER_WAITING_TIME])
      end
```

```
    end
  end
end
```

Création de tâches

Il est possible d'insérer directement des tâches dans la base de données depuis le code métier, mais il est plus pratique de passer par une API Ruby qui isole un peu le stockage, notamment le fait de stocker les paramètres au format JSON. Je vais donc ajouter une méthode dans le module `TaskEngine`.

Dans une version packagée du code, on pourrait vouloir l'isoler dans une gem à part pour éviter d'avoir à embarquer le code du moteur de tâches dans une application qui ne ferait qu'insérer des tâches.

task_engine.rb.

```
module TaskEngine
  # Make json-related methods available
  DB.extension :pg_json

  # @param [String] task_class
  # @param [Hash] task_parameters
  def self.create_task(task_class, task_parameters)
    Task.create(
        task_class: task_class,
        # Sequel.pg_json_wrap serialize a Hash
        # into something that can be inserted into a jsonb column
        task_parameters: Sequel.pg_jsonb_wrap(task_parameters)
    )
  end
end
```

Et je peux m'en servir pour modifier la tâche Rake qui peuple la table :

Rakefile.

```
desc 'Create tasks'
task :create_tasks do
  require_relative 'task_engine'
  100.times do
    TaskEngine.create_task(
        'TaskEngine::Tasks::WaitingTask',
        {
            waiting_time: 5
        })
  end
```

```
end
```

Et après exécution on peut vérifier que l'insertion se passe bien :

```
$ psql --user=task_engine --dbname=task_engine  --expanded --command="select * from
    tasks"

-[ RECORD 1 ]---+------------------------------
id              | 211
status          | waiting
created_at      | 2020-08-10 16:52:50.219772
updated_at      | 2020-08-10 16:52:50.219772
instance        |
task_class      | TaskEngine::Tasks::WaitingTask
task_parameters | {"waiting_time": 5}
-[ RECORD 2 ]---+------------------------------
id              | 212
status          | waiting
created_at      | 2020-08-10 16:52:50.221219
updated_at      | 2020-08-10 16:52:50.221219
instance        |
task_class      | TaskEngine::Tasks::WaitingTask
task_parameters | {"waiting_time": 5}
```

L'exécution

Il manque la dernière pièce du puzzle : utiliser les valeurs pour créer la tâche et l'exécuter.

Pour récupérer une classe à partir d'une chaîne contenant son nom, Ruby fournit une méthode `Module#const_get`, il faudra ensuite en instancier un objet de la classe ainsi obtenue, puis appeler `execute` sur l'instance en lui passant les paramètres :

task_engine.rb.

```
module TaskEngine
  class Engine
    def execute
      while (@engine.status == TaskEngine::Engine::ENGINE_STATUS_RUNNING) &&
          (task = try_acquire_task)
        starting_time = DateTime.now
        LOGGER.info("Worker #{@worker_index} starting task #{task.id}")

        task_class_name = task.task_class
        task_class = Object.const_get(task_class_name)
        task_instance = task_class.new
        task_instance.execute(task.task_parameters)

        stopping_time = DateTime.now
```

```
        elapsed_time = (stopping_time - starting_time).to_f * MILLISECONDS_IN_A_DAY
        LOGGER.info("Worker #{@worker_index} finished task #{task.id}, took #{
            elapsed_time}ms")
        end_task(task)
      end
      LOGGER.info("Worker #{@worker_index} is stopping")
    end
  end
end
```

Ce qui donne :

```
$ TASK_ENGINE_INSTANCE=instance_01 rake start_engine
Starting engine
Starting worker 0
Starting worker 1
Starting worker 2
Starting worker 3
Starting worker 4
Joining worker 0
(0.000937s) BEGIN
(0.001048s) SELECT * FROM "tasks" WHERE ("status" = 'waiting') ORDER BY "created_at"
    LIMIT 1 FOR UPDATE
(0.000534s) UPDATE "tasks" SET "status" = 'running', "instance" = 'my_instance' WHERE
     ("id" = 1501)
(0.000797s) COMMIT
Worker 0 starting task 1501
```

Et voila : il est maintenant possible de sélectionner des tâches et de leur passer des paramètres.

Pour le moment c'est assez rustique, par exemple il n'y a pas encore de gestion d'erreur, mais la structure est là.

Dans la partie suivante je vais faire en sorte que le moteur réagisse quand on ajoute des tâches en cours de route.

Partie 7 : l'ajout de tâches

Pour le moment, quand le moteur de tâches est lancé il commence à traiter des tâches, et s'arrête quand il n'y en a plus. Si j'ajoute des tâches après-coup, il ne se passe rien. En effet une fois les workers en pause, aucun mécanisme ne les réveille quand d'autres tâches sont insérées dans la base de données.

L'un des avantage de l'utilisation de bus de messages comme ActiveMQ ou Kafka pour stocker les tâches d'une moteur de tâches plutôt qu'une base de données SQL traditionnelle vient de leur fonctionnalités de notification, où le code du moteur peut écouter des évènements (ici les tâches qui sont ajoutées) envoyés par le bus, pour déclencher le traitement des tâches.

Redis fournit également ce type de mécanisme, mais c'est aussi le cas pour PostgreSQL à travers les fonctions `LISTEN` et `NOTIFY`.

Le principe des notifications

Lorsque les workers sont en train de tourner, après avoir terminé une tâche ils vont en chercher une autre, même si cette dernière a été ajoutée après le démarrage du moteur de tâches. Dans ce cas il n'y a rien de spécial à faire.

Par contre, lorsqu'au moins un des workers s'est arrêté car il n'a pas trouvé de tâche disponible, il faut le solliciter pour qu'il se remette au travail quand une nouvelle tâche est créée.

Cela permet que la nouvelle tâche soit traitée au plus vite, plutôt que d'attendre qu'un worker qui est déjà en train d'exécuter une autre tâche ne la termine.

Lorsqu'on insère une nouvelle tâche, il faudra donc envoyer une notification pour réveiller les workers qui pourraient être disponibles.

Pour cela j'utiliserai un thread supplémentaire qui sera chargé d'écouter les notifications envoyées pour réveiller les workers.

En Ruby, la classe Queue est destinée à ce type d'usage où des thread doivent s'attendre et communiquer entre eux. Il s'agit d'une file (une sorte de liste dont les éléments sortent dans le même ordre que celui où ils sont insérés), à laquelle plusieurs threads peuvent accéder en

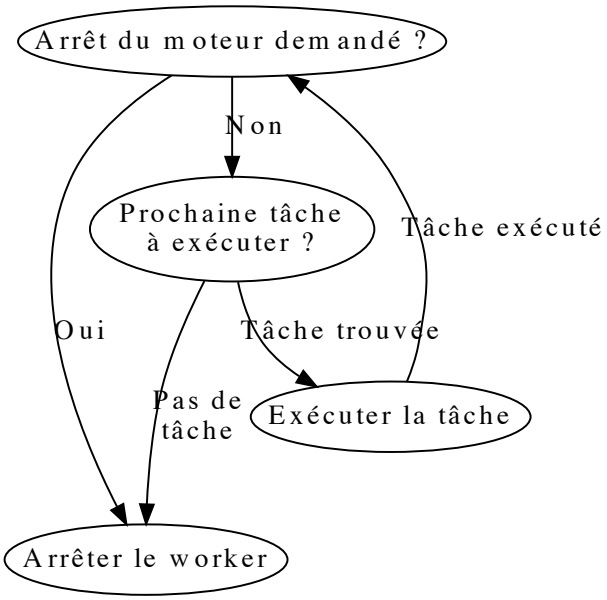
parallèle. Le comportement le plus intéressant pour un moteur de tâches est que si du code demande à récupérer un élément de la queue alors qu'elle est vide, le thread correspondant se mettra en attente jusqu'à ce qu'un élément soit disponible.

Il serait possible de manipuler directement les thread des workers comme pour l'arrêt du moteur de tâches, en passant les thread des workers en le statut `sleep` puis en les réveillant avec `Thread#wakeup`, mais `Queue` fournit une API prête à l'emploi pour cela, et cela évite d'avoir le genre de bugs pénibles qui apparaît quand on décide de réinventer la roue sur un sujet qui touche au threading.

Il faut garder en tête que le fonctionnement final doit être compatible avec le fait d'avoir plusieurs threads et plusieurs instances de moteur de tâches, et qu'il faut toujours pouvoir arrêter les différentes instances.

La nouvelle boucle de traitement des workers

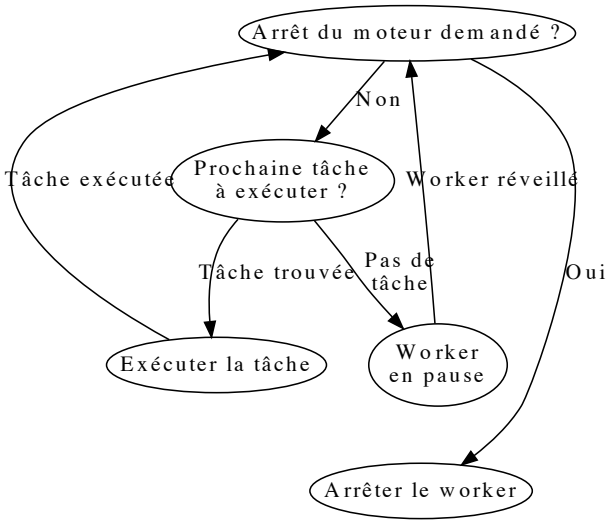
Le fonctionnement actuel des worker est celui-ci :



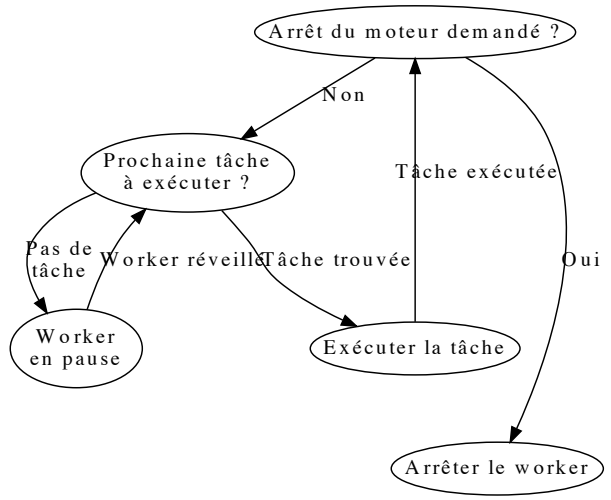
La demande d'arrêt du moteur et l'absence de tâche à exécuter mènent tous les deux à l'arrêt du worker.

Il faut ajouter une nouvelle boucle dans le schéma : en l'absence de tâche, au lieu de s'arrêter les workers doivent se mettre en pause en attendant une notification.

Lorsqu'on les réveille, ils vérifient que le moteur est toujours en fonctionnement, et dans ce cas se remettent à chercher une tâche.



On pourrait vouloir directement passer de la pause au fait de chercher des tâches à exécuter :



Mais dans ce cas, si le worker ne trouve pas de tâche à exécuter après avoir réveillé il ne peut pas s'arrêter mais seulement se remettre en sommeil, à moins d'ajouter du code spécifique pour cela.

La proposition précédente est donc un peu plus générique.

Pour implémenter le workflow, on peut utiliser une double boucle :

- la boucle extérieur vérifie l'extinction du moteur, et est déclenchée au réveil
- la boucle intérieur vérifier l'extinction et cherche la prochaine tâche, et est exécutée à chaque tâche

task_engine.rb.

```ruby
module TaskEngine
  class Worker
    def execute
      while @engine.status == Engine::ENGINE_STATUS_RUNNING
        while (@engine.status == Engine::ENGINE_STATUS_RUNNING) && (task =
            try_acquire_task)
          # Execute the task
        end
        # Wait until the notification awakes the worker
        LOGGER.info("Worker #{@worker_index} is sleeping")
        # We don't care about the message we got from the queue
        # since we just want to be awaken
        @engine.queue.pop
        LOGGER.info("Worker #{@worker_index} is awaken")
      end
      LOGGER.info("Worker #{@worker_index} is stopping")
    end
  end
end
```

Pour recevoir la notification, je dois créer un nouveau thread car l'écoute bloque l'exécution du thread qui attend d'être notifié.

Je vais créer ce thread dans le constructeur de l'`Engine`.

À la réception d'un message, il utilise la `Queue` pour réveiller un worker si l'un d'eux est disponible.

task_engine.rb.

```ruby
module TaskEngine
  class Engine
    def initialize(instance)
    # …
    Thread.new do
      DB.listen(NOTIFICATION_CHANNEL, loop: true) do |_channel, _pid, task_class|
        LOGGER.info("Received notification for a task [#{task_class}]")
        unless @queue.num_waiting == 0
          LOGGER.info('Notifying worker')
          @queue.push(task_class)
        end
      end
    end
    # …
  end
```

Le nom de la classe de la tâche est envoyé dans la notification pour être logué, mais ne sert pas dans le reste du code, en tous cas pour le moment.

Il faut penser à ajouter une connexion à la configuration de la base de donnée plus haut dans

le fichier :

task_engine.rb.

```
module TaskEngine
  # Use at most one connexion per worker + the connexion for notifications
  DB = ::Sequel.connect(DATABASE_URL, max_connections: WORKERS_NUMBER + 1, logger:
      LOGGER)
end
```

Envoyer une notification ou le contenu d'une tâche

Dans le modèle que je décris, le moteur de tâches reçoit des notification l'informant qu'une tâche devient disponible et il doit ensuite acquérir la tâche dans la base de données, et en passant récupérer les paramètres d'exécution.

Cela vient du fait que dans le système de notification de PostgreSQL les notifications sont envoyées à toutes les instances qui sont à l'écoute.

D'autres système de messages (par exemple ActiveMQ) permettent que chaque message ne soit reçu que par une instance au plus. Dans ce cas l'acquisition de la tâche et la récupération des paramètres sont concentrés en une seule action : un moteur de tâches qui reçoit un message sait qu'il est le seul dans ce cas et peut donc directement exécuter la tâche.

Pour reprendre le vocabulaire de la partie 3, les systèmes de message comme ActiveMQ peuvent fournir plus ou moins du "*exactly once*". Comme indiqué dans la partie 3, ce type de mécanisme est très difficile à mettre en œuvre de manière fiable, et en cas de crash du bus de message il y a des chances (même si elles sont faibles) que vous receviez le même message deux fois, il faut donc être préparé à cette éventualité.

Pourquoi ne pas utiliser une `id` pour récupérer la prochaine tâche ?

Lors de la création d'une tâche, le code connaît la tâche à créer et ses paramètres.

On pourrait vouloir utiliser ces informations pour simplifier l'acquisition de tâches{nsbp} : quand réveille un worker, au lieu de chercher la prochaine tâche à exécuter avec la requête existante, il pourrais plutôt chercher la tâche par son `id` une requête qui cherche si la tâche avec un certain `id` est toujours disponible, ce qui devrait rendre la requête plus rapide.

Malheureusement cette solution n'est pas adaptée au fait d'avoir plusieurs {mdts} qui s'exécutent simultanément.

Par exemple si du code créé deux tâches T1 et T2, et que deux instances du moteur de tâches disposent de workers en sommeil. En utilisant la requête par défaut, le premier worker à se réveiller va récupérer T1, et le deuxième worker T2. Si les deux workers cherchent T1 par son `id`, le premier worker va la trouver mais pas le deuxième.

Il lui faudra alors utiliser la requête par défaut pour déterminer si une autre tâche est disponible. En échange d'une requête plus rapide, on aura donc exécuté une requête inutile.

Utiliser la requête générique évite donc cela, tout en gardant le code un peu plus simple.

La partie notification

Pour déclencher la notification, j'ajoute l'appel à la méthode `notify` lors de la création d'une tâche :

task_engine.rb.

```
module TaskEngine
  # @param [String] task_class
  # @param [Hash] task_parameters
  def self.create_task(task_class, task_parameters)
    Task.create(
        task_class: task_class,
        task_parameters: Sequel.pg_json_wrap(task_parameters)
    )
    DB.notify(NOTIFICATION_CHANNEL, payload: task_class)
  end
end
```

On peut tester le fonctionnement en lançant le moteur de tâches dans une console, et en lançant la création de tâche dans une autre :

```
$ TASK_ENGINE_INSTANCE=instance_01 rake start_engine

Worker 2 is sleeping
Worker 3 is sleeping
Worker 0 is sleeping
Worker 1 is sleeping
Worker 4 is sleeping
Received notification for a task [TaskEngine::Tasks::WaitingTask]
Notifying worker
Worker 2 is awaken
Received notification for a task [TaskEngine::Tasks::WaitingTask]
Notifying worker
Worker 3 is awaken
Received notification for a task [TaskEngine::Tasks::WaitingTask]
Worker 2 starting task 1601
```

Notifications et transations

Une des avantages d'utiliser le même outil pour stocker les données et gérer les messages et que les mêmes transactions s'appliquent aux deux.

Ainsi dans le code de `TaskEngine.create_task`, l'insertion de la tâche et l'envoi de la notification se font dans la même transaction que le code appelant.

Cela signifie que qu'en cas de `rollback` la tâche n'est pas crée et la notification envoyée.

Par exemple si une tâche a besoin d'un objet métier créé juste avant, si la tâche est créée puis que la création de l'objet est rollbackée, la tâche ne pourra pas le trouver.

Si vos traitements ont un risque significatif de `rollbak`, cela peut être assez pratique car cela évite d'avoir à prendre en compte cette situation dans le code de la tâche.

En effet un `rollback` de transaction ne signifie pas forcément qu'une erreur s'est produite, mais peut être le signe d'une tentative de modification concurrente dans la base de données, et dans ce cas arriver régulièrement.

Si on ne souhaite pas ce comportement, la création de tâche peut être isolée dans une transaction séparée dans le code applicatif, mais le comportement par défaut fournit une garantie qui peut être bien pratique.

Maintenant que le moteur sait traiter les ajouts de tâche, je vais m'intéresser au monitoring pour pouvoir commencer à suivre ce qui se passe.

Partie 8 : le monitoring unitaire

Pour le moment quand une tâche a été exécutée elle est effacée et on n'en garde aucune trace, à part dans les logs.

Parser les logs est faisable, mais extraire certaines lignes parmi une myriades d'autres est toujours risqué et je préfère donc les stocker dans un endroit spécifique.

Comme expliqué dans la troisième partie, je vais les stocker dans PostgreSQL comme les données des tâches à exécuter.

Cela me permettra de faire des liens entre les informations de monitoring et les tâches.

Les informations de monitoring

Pour l'instant lors de l'exécution d'une tâche, trois informations sont disponibles :

- l'heure de début d'exécution
- l'heure de fin d'exécution
- le résultat de l'exécution des tâches

Dans les trois, la première est présente dès le début de l'exécution de la tâche, et elle peut donc être définie comme non-nullable. Les autres seront renseignées à la fin de la tâche. Le résultat de l'exécution de tâches sera stocké dans une colonne de type `jsonb`, pour les même raisons que les paramètres des tâches.

Il faut aussi ajouter un nouveau statut `stopped` aux tâches qui indique qu'elles ont été exécutées, puisqu'elles ne seront plus supprimées.

migrations/04_create_task_executions.rb.

```
Sequel.migration do
  change do
    add_enum_value(:task_status, 'stopped')

    create_table(:task_executions) do
      primary_key :id
```

```
      foreign_key :task_id, :tasks, null: false
      DateTime :started_at, null: false
      DateTime :stopped_at, null: true
      column :result, 'JSONB', null: true

      DateTime :created_at, null: false
      DateTime :updated_at, null: false
    end
  end
end
```

Ensuite j'ajoute la déclaration du modèle, en modifiant la définition de `Task` pour ajouter le nouveau statut la relation entre les deux :

task_engine.rb.

```
module TaskEngine
  class Task < Sequel::Model
    STATUS_WAITING = 'waiting'
    STATUS_RUNNING = 'running'
    STATUS_STOPPED = 'stopped'
    one_to_many :task_executions
  end
  class TaskExecution < Sequel::Model
    many_to_one :task
  end
end
```

L'ajout des informations

Reste à ajouter l'insertion et la mise à jour des informations d'exécution :

task_engine.rb.

```
module TaskEngine
  class Worker
    def execute
      while # …
        starting_time = DateTime.now
        LOGGER.info("Worker #{@worker_index} starting task #{task.id}")

        task_execution = nil
        DB.transaction do
          task_execution = TaskExecution.create(
              task: task,
              started_at: starting_time
          )
        end
```

```
        task_class_name = task.task_class
        task_class = Object.const_get(task_class_name)
        task_instance = task_class.new
        task_result = task_instance.execute(task.task_parameters)

        stopping_time = DateTime.now
        elapsed_time = (stopping_time - starting_time).to_f * MILLISECONDS_IN_A_DAY
        LOGGER.info("Worker #{@worker_index} finished task #{task.id}, took #{
            elapsed_time}ms")

        DB.transaction do
          task_execution.update(
              stopped_at: stopping_time,
              result: Sequel.pg_json_wrap(task_result),
              status: TaskExecution::STATUS_SUCCESS
          )
          task.update(
              status: Task::STATUS_STOPPED
          )
        end
      end
      # …
    end
end
```

Le test

On peut ensuite tester le résultat :

```
$ TASK_ENGINE_INSTANCE=instance_01 rake start_engine

Starting engine
Starting worker 0
Joining worker 0
(0.000138s) BEGIN
(0.000793s) SELECT * FROM "tasks" WHERE ("status" = 'waiting') ORDER BY "created_at"
    LIMIT 1 FOR UPDATE
(0.001295s) UPDATE "tasks" SET "status" = 'running', "instance" = 'engine_1' WHERE ("
    id" = 201)
(0.000911s) COMMIT
Worker 0 starting task 201
(0.000130s) BEGIN
(0.000946s) INSERT INTO "task_executions" ("task_id", "started_at", "created_at", "
    updated_at") VALUES (201, '2020-08-10 16:52:54.054783+0200', '2020-08-10
    16:52:54.055373+0200', '2020-08-10 16:52:54.055373+0200') RETURNING *
(0.000474s) COMMIT
Worker 0 finished task 201, took 5007.844ms
```

```
(0.000365s) BEGIN
(0.000829s) UPDATE "task_executions" SET "stopped_at" = '2020-08-10
    16:52:59.062627+0200', "result" = 'null'::jsonb, "updated_at" = '2020-08-10
    16:52:59.063566+0200' WHERE ("id" = 6)
(0.000355s) BEGIN
(0.000551s) UPDATE "tasks" SET "status" = 'stopped' WHERE ("id" = 201)
(0.001536s) COMMIT
```

La table `task_executions` contient bien les informations attendues :

```
$ psql --user=task_engine --dbname=task_engine  --expanded --command="select * from
    task_executions"

-[ RECORD 1 ]--------------------------
id          | 6
task_id     | 201
started_at  | 2020-08-10 16:52:54.054783
stopped_at  | 2020-08-10 16:52:59.062627
result      | null
created_at  | 2020-08-10 16:52:54.055373
updated_at  | 2020-08-10 16:52:59.063566
-[ RECORD 2 ]--------------------------
id          | 7
task_id     | 202
started_at  | 2020-08-10 16:52:54.080271
stopped_at  | 2020-08-10 16:52:59.091997
result      | null
created_at  | 2020-08-10 16:52:54.080519
updated_at  | 2020-08-10 16:52:59.09332
```

La structure fonctionne donc et les données sont bien là, même si pour le moment on ne sait suivre que les tâches qui se terminent bien car la gestion d'erreur n'a pas encore été implémentée.

Et justement le dernier élément qui manque pour mettre en place cette gestion d'erreur est la capacité à planifier des tâches, qui sera traitée dans la partie suivante.

Et un index

Avant de terminer, il faut s'intéresser un peu à la performance.

Avec le temps la table `tasks` devrait contenir de plus en plus d'occurrences de tâches terminées (avec l'hypothèse que le nombre de tâches à exécuter devrait rester raisonnable).

Cela risque de ralentir et fur et à mesure la sélection des tâches à exécuter, car la requête a besoin pour le moment de parcourir tous les enregistrements de la table.

Pour éviter cela, je vais ajouter un index à la table `tasks`.

La requête est la suivante :

```
SELECT * FROM "tasks" WHERE ("status" = 'waiting') ORDER BY "created_at" LIMIT 1
```

Je pourrais simplement créer un index sur `statut`, ce qui ferait que la requête n'aurait qu'à parcourir les enregistrements qui ont le bon statut.

Mais pour que la recherche soit encore plus rapide, il est possible de créer un index sur les colonnes `statut` et `created_at`, de cette manière la recherche utilisera seulement l'index.

J'ajoute pour cela une nouvelle migration :

migrations/05_create_task_index.rb.

```
Sequel.migration do
  change do
    alter_table(:tasks) do
      add_index [:status, :created_at]
    end
  end
end
```

Après exécution de la migration, l'exécution de la requête utilise bien l'index pour faire la recherche :

```
Limit  (cost=0.14..0.30 rows=1 width=120)
  ->  Index Scan using tasks_status_created_at_index on tasks  (cost=0.14..16.20 rows
      =100 width=120)
        Index Cond: (status = 'waiting'::task_status)
```

Partie 9 : la planification

Jusqu'à présent il est seulement possible de créer des tâches à exécution immédiate.

Les tâches dont l'exécution n'est pas immédiate correspondent à deux cas :

- les tâches dont on veut programmer des exécutions régulières, par exemple toutes les quelques minutes ou toutes les nuits à 03h12 ;
- les tâches qu'on veut exécuter une seule fois mais dans le futur à une heure déterminée, elles sont peu utilisées directement par les applications, mais permette de faire de réessais plus tard en cas d'erreur.

Les tâches à exécution régulière

Les tâches à exécution régulière correspondent à ce qu'il est possible de faire avec l'utilitaire cron dans les système Unix et Linux.

Cela peut par exemple servir pour programmer des traitement d'archivage ou des rappels toutes les nuits.

La solution la plus simple est d'utiliser l'outil cron directement. Même s'il est un peu passé de mode, il est fiable et bien documenté.

Lors d'un déploiement il est facile de mettre à jour la liste des tâches programmées si le fichier contenant la configuration cron fait partie du code.

Ses deux inconvénients majeurs pour cet usage sont :

- l'impossibilité de modifier la configuration cron facilement, par exemple à travers une interface graphique
- le fait que les commandes cron doivent être lancées sur une seule machine (sinon les tâches seront créés autant de fois qu'il y a de machines), ce qui peut poser des contraintes de disponibilité.

Si vous n'êtes pas dans une de ces situations, cron peut donc parfaitement faire l'affaire.

Pour créer les tâches depuis cron, il y a deux manières de faire :

1. Lancer une commande Ruby qui va faire l'insertion en utilisant la méthode `TaskEngine.create_task`.
2. Faire directement l'insertion en base de donnée en utilisant du SQL et psql.

La première a l'avantage de la simplicité de mise en œuvre, par contre elle a l'inconvénient de lancer une machine virtuelle Ruby pour exécuter deux commandes SQL très simple (l'insertion de la tâche et la notification).

L'inconvénient principal de la seconde manière est que le format des données dans la base devient une API. En effet s'il change il faudra modifier le code Ruby mais peut-être aussi les scripts qui accèdent directement à la base.

Si c'est une approche qui est plutôt à éviter lorsqu'il s'agit d'une API exposée à d'autres systèmes, elle peut être acceptable ici vu que la programmation des tâches fait partie du même ensemble applicatif.

Je ne vais détailler l'utilisation de cron, il y a déjà bien assez de documentation à ce sujet, et je vais donc passer aux tâches à exécution unique.

Les tâches à exécution unique

Ces tâches ressemblent beaucoup aux tâches existantes, sauf qu'elles doivent s'exécuter dans le futur plutôt qu'immédiatement.

Jusqu'à présent, une tâche dans l'état `waiting` était forcément prête à être exécutée.

Je vais devoir changer ce comportement pour prendre en compte qu'une tâche puisse être en attente mais pas encore prête à être exécutée tout de suite.

Pour cela, je vais ajouter un nouveau champ `scheduled_at` qui contiendra l'heure à partir de laquelle la tâche peut être exécutée, avec pour valeur par défaut l'heure actuelle pour que la tâche puisse se lancer immédiatement.

L'implémentation va au final toucher peu de code.

Tout d'abord la migration qui ajoute le champ et qui recrée un nouvel index sur les colonnes `statut` et `scheduled_at` pour accélerer la requête de sélection de la prochaine tâche à exécute tout en supprimant l'index existant.

migrations/06_add_scheduled_at.rb.

```
Sequel.migration do
  change do
    alter_table(:tasks) do
```

```
      add_column :scheduled_at, DateTime, null: false, default: Sequel.lit('
          LOCALTIMESTAMP')
      drop_index [:status, :created_at]
      add_index [:status, :scheduled_at]
    end
  end
end
```

Je préfère la base de donnée se charge de donner la valeur par défaut de la colonne (qui correspond à l'heure en cours) car cela évite d'avoir à dépendre du fait que les différents clients soient bien à l'heure. Comme la base de données est unique, cela garantit la cohérence des valeurs.

Ensuite l'ajout du paramètre permettant de spécifier la valeur lors de la création d'une tâche :

task_engine.rb.

```
module TaskEngine
  # @param [String] task_class
  # @param [Hash] task_parameters
  # @param [DateTime, nil] scheduled_at nil means an immediate execution
  def self.create_task(task_class, task_parameters, scheduled_at = nil)
    creation_params = {
        task_class: task_class,
        task_parameters: Sequel.pg_json_wrap(task_parameters)
    }
    # Don't send a null value to the DB so the DB use the default value
    unless scheduled_at.nil?
      creation_params[:scheduled_at] = scheduled_at
    end
    Task.create(creation_params)
    if scheduled_at.nil?
      DB.notify(NOTIFICATION_CHANNEL, payload: task_class)
    end
  end
end
```

Si le paramètre `scheduled_at` est spécifié, et que la tâche est programmée pour s'exécutée plus tard, il n'est pas nécessaire d'envoyer de notification car les notifications servent à réveiller des workers pour une exécution immédiate.

Le dernier élément à modifier est la requête qui cherche la prochaine tâche à exécuter, qui doit maintenant faire un tri sur `scheduled_at` plutôt que `created_at`, et qui doit également ignorer les tâches dont la valeur de `scheduled_at` est dans le futur.

Comme pour la valeur par défaut, je vais laisser la base définir l'heure actuelle. `Sequel.lit` indique à Sequel d'utiliser la valeur telle-quelle sans l'interpréter, elle est utile comme lorsqu'ici

on utilise des fonctions SQL.

task_engine.rb.

```
module TaskEngine
  class Worker
    # @return [TaskEngine::Task, nil]
    def try_acquire_task
      DB.transaction do
        task = Task.
            where(status: Task::STATUS_WAITING).
            where(Sequel.lit('scheduled_at < LOCALTIMESTAMP')).
            order(:scheduled_at).for_update.first
        unless task.nil?
          Task.where(id: task.id).update(
              status: Task::STATUS_RUNNING,
              instance: @engine.instance
          )
          task
        end
      end
    end
  end
end
```

On peut vérifier que si on créé des tâches dans le futur, elles ne sont pas immédiatement exécutées.

Je vais créer une tâche qui doit s'exécuter 30 secondes dans le futur. Pour cela j'ajoute la fraction `30 / 86400` (c'est-à-dire le nombre standard de secondes dans un jour) à l'heure actuelle.

```
$ bundle exec irb
require_relative 'task_engine'
TaskEngine.create_task(
    'TaskEngine::Tasks::WaitingTask',
    {
        waiting_time: 5
    },
    DateTime.now + Rational(30, 86400))
exit

$ TASK_ENGINE_INSTANCE=instance_01 rake start_engine
```

Tout d'abord comme prévu il ne se passe rien, car la date est dans le futur.

Allo la tâche ?

Mais au bout de 30 secondes il ne se passe rien non plus. Le problème c'est qu'en l'absence de notification aucun worker ne vérifie si la tâche est disponible pour être exécutée.

Si une autre tâche était créée en exécution immédiate après que les 30 secondes se soient passées, cela réveillerait un worker, et la tâche programmée serait traitée, mais tant qu'aucune tâche immédiate n'est créée, la tâche programmée restera à attendre.

Si de nouvelles tâches à exécution immédiates sont crées régulièrement, cette attente peut ne pas être un problème, mais peut-être que ça n'est pas le cas, ou que même si c'est le cas en principe, on peut préférer garantir que la tâche ne va pas attendre trop longtemps, en tous cas si des workers sont disponibles.

La manière idéale de faire serait faire en sorte de réveiller un worker dès qu'une tâche devient exécutable, mais il n'y a pas d'approche facile pour le faire avec les outils que j'ai choisis.

Une approche possible serait d'envoyer une notification lors de chaque création de ces tâches, et que les instances de moteur de tâches programment un réveil de worker pour chaque moment où au moins une des tâches doit s'exécuter.

Mais elle a cependant deux inconvénients.

Le premier est est que cela augmenterait la consommation mémoire des instances du moteur de tâches, avec un risque de dépassement mémoire si trop de tâches sont crées.

Le second est que le fonctionnement se mettrait à dépendre du fait que les instances de moteur de tâches ne soient pas redémarrées, car alors les informations de réveil seraient perdus.

Je vais plutôt partir du principe que si une tâche est programmée pour s'exécuter dans le futur, ce n'est pas grave si elle a quelques secondes de retard. Cela me permet de choisir une approche moins fine mais qui n'a pas les deux inconvénients cités plus hauts.

Ma solution sera de programmer à intervalle régulier des "réveils" qui reproduiront le même mécanisme que la réception d'une notification en poussant un élément dans la `Queue` où attendent les workers disponibles. Cela signifie envoyer un message pour réveiller un éventuel worker disponible, qui ira voir si une tâche est prête à être exécutée.

Pour cela je vais créer un nouveau `Thread` qui sera dédié à cet usage :

task_engine.rb.

```
module TaskEngine
  class Engine
    def initialize(instance)
```

```
      # …

      # Scheduled wake-ups
      Thread.new do
        while true do
          LOGGER.info('Scheduled wake-up')
          @queue.push('Scheduled wake-up')
          sleep(10)
        end
      end
      # …
    end
  end
end
```

Le temps entre deux réveil, ici fixé à 10 seconde, correspond au temps maximum d'attente pour une tâche programmée, si des workers sont disponibles.

En effet si les workers sont déjà tous occupées à traiter un arriéré de tâches, il peut s'écouler un certain temps avant qu'une tâche programmée qui vient de devenir exécutable soit effectivement lancée.

Maintenant qu'il est possible de créer des tâches qui s'exécuteront plus tard, je vais pouvoir m'en servir pour mettre en place la gestion d'erreur.

Partie 10 : la gestion d'erreurs

Parfois les choses se passent mal, et le code plante. Cela se produit même plus que parfois quand il s'agit de faire des appels réseaux, c'est-à-dire typiquement le genre de choses que fait un moteur de tâches.

Il faut donc que le moteur de tâches puisse prendre en compte cette possibilité.

Tout d'abord il faut que, autant que possible, une erreur dans une tâche n'ait pas de conséquence sur le moteur de tâches. Cela vaut bien entendu pour les erreurs applicatives, on ne peut pas grande chose face un crash de la machine virtuelle pour cause de mémoire insuffisante.

Ensuite il faut pouvoir monitorer ce qui s'est passé, en stockant les messages d'erreurs et les heures d'exécutions pour pouvoir aider à comprendre.

En finalement il faut pouvoir relancer les tâches, en effet l'expérience prouve que de relancer les tâches automatiquement un peu plus tard est souvent la bonne solution car beaucoup de problèmes sont temporaires.

Une tâche qui plante

La première chose à faire est d'avoir une tâche qui plante pour pouvoir tester ce qui va suivre.

Jusqu'ici le seul type de tâche qui existait était la `WaitingTask`, je vais donc ajouter une `FailingTask` :

tasks.rb.

```
module TaskEngine
  module Tasks
    class FailingTask
      # @param [Hash] parameters
      # @return [Hash, nil]
      def execute(parameters)
        1 / 0
      end
```

```
    end
  end
end
```

Comme il n'y aucune gestion d'erreur pour le moment, la division par zéro fait planter l'instance du moteur de tâches.

```
#<Thread:0x00007fa2531e5a30 /task_engine.rb:123 run> terminated with exception (
    report_on_exception is true):
Traceback (most recent call last):
    3: from /task_engine.rb:124:in `block in initialize'
    2: from /task_engine.rb:144:in `execute'
    1: from /tasks.rb:18:in `execute'
/tasks.rb:18:in `/': divided by 0 (ZeroDivisionError)
^CI, [2020-09-18T21:16:54.385296 #53807]  INFO -- : Worker 2 is awaken
I, [2020-09-18T21:16:54.385604 #53807]  INFO -- : Worker 2 is stopping
I, [2020-09-18T21:16:54.385467 #53807]  INFO -- : Worker 4 is awaken
I, [2020-09-18T21:16:54.385714 #53807]  INFO -- : Worker 4 is stopping
I, [2020-09-18T21:16:54.385403 #53807]  INFO -- : Worker 0 is awaken
I, [2020-09-18T21:16:54.385818 #53807]  INFO -- : Worker 0 is stopping
I, [2020-09-18T21:16:54.385541 #53807]  INFO -- : Worker 3 is awaken
I, [2020-09-18T21:16:54.386093 #53807]  INFO -- : Worker 3 is stopping
I, [2020-09-18T21:16:54.385996 #53807]  INFO -- : Joining worker 1
rake aborted!
ZeroDivisionError: divided by 0
/tasks.rb:18:in `/'
/tasks.rb:18:in `execute'
/task_engine.rb:144:in `execute'
/task_engine.rb:124:in `block in initialize'
Tasks: TOP => start_engine
```

Pour mettre en place la gestion d'erreur je vais commencer par enrichir le modèle de données : dans la table `task_executions` je vais ajouter un `status` pour déclarer que l'exécution a eu une erreur, et une `error` pour pouvoir enregistrer les informations sur l'erreur.

migrations/07_add_errors_to_task_execution.rb.

```
Sequel.migration do
  change do
    extension :pg_enum
    create_enum(:task_execution_status, ['running', 'success', 'failure'])
    alter_table(:task_executions) do
      add_column :status, :task_execution_status, null: false
      add_column :error, 'JSONB', null: true
    end
  end
end
```

Le `status running` va correspondre aux tâches en cours d'exécution. Jusque là il était possible de déterminer qu'une exécution n'était pas terminée en regardant si colonne `stopped_at` était

vide. Mais comme je préfère que la colonne `status` ne soit pas nullable pour éviter les oublis, j'ai besoin d'un `status` pendant l'exécution.

La première chose est d'ensuite de déclarer les `status` comme des constantes dans la classe `TaskExecution` :

task_engine.rb.

```
module TaskEngine
  class TaskExecution < Sequel::Model
    STATUS_RUNNING = 'running'
    STATUS_SUCCESS = 'success'
    STATUS_FAILURE = 'failure'
    many_to_one :task
  end
end
```

Et je peux maintenant ajouter le début de la gestion d'erreur : quand l'exécution d'une tâche renvoie une exception, elle sera déclarée en erreur et l'erreur sera enregistrée sous la forme du message d'erreur, de la classe de l'exception, et de la pile d'appel.

task_engine.rb.

```
module TaskEngine
  class Worker
    def execute
      # …
      while (@engine.status == Engine::ENGINE_STATUS_RUNNING) && (task =
          try_acquire_task)
        starting_time = DateTime.now
        LOGGER.info("Worker #{@worker_index} starting task #{task.id}")

        task_execution = nil
        DB.transaction do
          task_execution = TaskExecution.create(
              task: task,
              started_at: starting_time,
              status: TaskExecution::STATUS_RUNNING
          )
        end

        task_class_name = task.task_class
        begin
          task_class = Object.const_get(task_class_name)
          task_instance = task_class.new
          task_result = task_instance.execute(task.task_parameters)

          stopping_time = DateTime.now
          elapsed_time = (stopping_time - starting_time).to_f * MILLISECONDS_IN_A_DAY
          LOGGER.info("Worker #{@worker_index} finished successfully task #{task.id}
```

```
              in #{elapsed_time}ms, result is #{result}")
            DB.transaction do
              task_execution.update(
                  stopped_at: stopping_time,
                  result: Sequel.pg_json_wrap(task_result),
                  status: TaskExecution::STATUS_SUCCESS
              )
                task.update(
                    status: Task::STATUS_STOPPED
              )
            end
          rescue Exception => e
            stopping_time = DateTime.now
            elapsed_time = (stopping_time - starting_time).to_f * MILLISECONDS_IN_A_DAY
            LOGGER.info("Worker #{@worker_index} failed task #{task.id} in #{
                elapsed_time}ms, error is #{e.inspect}")
            DB.transaction do
              task_execution.update(
                  stopped_at: stopping_time,
                  error: Sequel.pg_json_wrap(
                      {
                          exception: e.class,
                          message: e.message,
                          backtrace: e.backtrace
                      }),
                  status: TaskExecution::STATUS_FAILURE
              )
              task.update(
                  status: Task::STATUS_STOPPED
              )
            end
          end
        end
        # …
      end
    end
  end
end
```

Et voici le résultat quand je rééessaie d'exécuter une `FailingTask`{nsbp} :

```
Worker 0 starting task 1
(0.000353s) BEGIN
(0.003452s) INSERT INTO "task_executions" ("task_id", "started_at", "status", "
    created_at", "updated_at") VALUES (1, '2020-09-18 21:53:03.890717+0200', 'running
    ', '2020-09-18 21:53:03.893951+0200', '2020-09-18 21:53:03.893951+0200')
    RETURNING *
(0.000827s) COMMIT
Worker 0 failed task 1 in 9.501ms, error is #<ZeroDivisionError: divided by 0>
(0.000334s) BEGIN
(0.001450s) UPDATE "task_executions" SET "stopped_at" = '2020-09-19
    11:27:17.430026+0200', "updated_at" = '2020-09-19 11:27:17.431071+0200', "status"
```

```
     = 'failure', "error" = '{"exception":"ZeroDivisionError","message":"divided by
    0","backtrace":["/tasks.rb:18:in `/'","/tasks.rb:18:in `execute'","/task_engine
    .rb:149:in `execute'","/task_engine.rb:127:in `block in initialize'"]}'::jsonb
    WHERE ("id" = 1)
(0.000737s) UPDATE "tasks" SET "status" = 'stopped' WHERE ("id" = 1)
(0.000681s) COMMIT
```

Réessayer

Comme décrit plus haut, il est souvent bénéfique de relancer une tâche qui a échoué pour les cas où la source de l'erreur était transitoire.

Cela signifie que les tâches doivent être prévues pour cela, par exemple si une tâche échoue à la fin de son exécution alors qu'elle a déjà effectués certains traitements des traitements, le fait de la relancer ne doit pas avoir d'effet secondaires.

Parfois cela est facile à faire, par exemple si toute la tâche s'exécute dans une seule transaction, mais si la tâche interagit avec des systèmes extérieurs cela peut être plus compliqué.

Pour un moteur de tâches, une manière de résoudre ce problème est de rendre le réessai configurable, je vais commencer par implémenter des réessais systématiques, puis j'ajouterai la configurabilité.

Quand on veut réessayer une tâche dans l'espoir que le problème qui l'empêchait de fonctionner ne se produira plus, il faut que le réessai ne soit pas immédiat. En général on commence par réessayer un peu plus tard, puis encore un peu plus tard mais en attendant un peu plus, et ainsi de suite en augmentant à chaque fois la durée de l'attente.

Il s'agit de trouver un équilibre entre une attente plus longue, qui augmente les chances que le problème ait été résolu, et une attente trop longue qui deviendrait pénalisante.

Une approche souvent employée est d'utiliser des puissance de deux : le premier réessai se faut au bout de $2^0 = 1$ minute, le second au bout de $2^1 = 2$ minutes, puis 4, 8, 16...

Il faut aussi savoir abandonner, car au bout d'un moment réessayer ne sert plus à rien. Pour cela je vais définir un nombre maximum de réessais au bout duquel la tâche ne sera pas relancée. Le choix de la valeur est un peu arbitraire et dépend de l'environnement dans lequel on se trouve. Je vais choisir 10, ainsi la dernière tentative exécution se produira `1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256 + 512 = 1023` minutes après l'exécution initiale, soit un peu plus de 17 heures.

Si besoin, il serait possible de la rendre paramétrable, en utilisant par exemple le contexte que je vais ajouter juste en dessous.

Une tâche a besoin de savoir à quel numéro de réessai elle correspond, pour calculer l'heure du prochain rééssai et pour déterminer qu'il est temps de s'arrêter quand elle a atteint le nombre maximum.

Pour stocker cette information je pourrais utiliser les paramètres de la tâche, mais je préfère les réserver aux paramètres applicatifs.

Je vais donc ajouter une nouvelle colonne à la table `tasks`. Plutôt que d'ajouter une colonne spécifique qui ne contiendrait que cette unique valeur, je vais créer une colonne de contexte au format `jsonb`. L'idée est de pouvoir ensuite l'utiliser pour ajouter d'autre métadonnées si le moteur de tâches est étendu.

Je vais aussi avoir besoin d'un nouveau statut `failed` pour les tâches qui indiquer qu'elles ont épuisé leurs réessais.

Quand une tâche aura été relancée, il faudra donc distinguer le statut de la tâche des status de ses exécutions successives. Une tâche en status `waiting` avec des exécutions en `failed` aura donc déjà été lancée et sera donc en train d'attendre une nouvelle tentative.

migrations/08_add_task_context_and_task_failure.

```
Sequel.migration do
  change do
    extension :pg_enum
    add_enum_value(:task_status, 'failed')

    extension :pg_json
    alter_table(:tasks) do
      add_column :context, 'JSONB', null: false, default: Sequel.pg_json_wrap({})
    end
  end
end
```

Je déclare le nouveau statut{nsbp} :

task_engine.rb.

```
module TaskEngine
  class Task < Sequel::Model
      STATUS_WAITING = 'waiting'
      STATUS_RUNNING = 'running'
      STATUS_STOPPED = 'stopped'
      STATUS_FAILED = 'failed'
      one_to_many :task_executions
  end
end
```

Puis vient l'implémentation des réessais.

task_engine.rb.

```
module TaskEngine
  class Worker

    MAX_RETRIES_NUMBER = 10

    def execute
      # …
      rescue Exception => e
        stopping_time = DateTime.now
        elapsed_time = (stopping_time - starting_time).to_f * MILLISECONDS_IN_A_DAY
        LOGGER.info("Worker #{@worker_index} failed task #{task.id} in #{elapsed_time
            }ms, error is #{e.inspect}")
        DB.transaction do
          task_execution.update(
              stopped_at: stopping_time,
              error: Sequel.pg_json_wrap(
                  {
                      exception: e.class,
                      message: e.message,
                      backtrace: e.backtrace
                  }),
              status: TaskExecution::STATUS_FAILURE
          )
        end
        current_retry_number = task.context['retry_number'] || 0
        if current_retry_number < MAX_RETRIES_NUMBER
          task.context['retry_number'] = current_retry_number + 1
          # 2^current_retry_number numbers of minutes / number of minutes in a day
          retry_scheduled_at = DateTime.now + Rational(2 ** current_retry_number,
              1440)
          LOGGER.info("Worker #{@worker_index} task #{task.id} will be retried at #{
              retry_scheduled_at}")
          DB.transaction do
            task.update(
                status: Task::STATUS_WAITING,
                context: task.context,
                scheduled_at: retry_scheduled_at
            )
          end
        else
          LOGGER.info("Worker #{@worker_index} task #{task.id} retries exhausted")
          DB.transaction do
            task.update(
                status: Task::STATUS_FAILED
            )
          end
        end
      end
    end
```

```
  end
end
```

Ce qui donne :

```
Worker 0 starting task 2
(0.000135s) BEGIN
(0.001220s) INSERT INTO "task_executions" ("task_id", "started_at", "status", "
    created_at", "updated_at") VALUES (2, '2020-09-19 17:41:07.166732+0200', 'running
    ', '2020-09-19 17:41:07.167594+0200', '2020-09-19 17:41:07.167594+0200')
    RETURNING *
(0.000648s) COMMIT
Worker 0 failed task 2 in 3.49ms, error is #<ZeroDivisionError: divided by 0>
(0.000141s) BEGIN
(0.000911s) UPDATE "task_executions" SET "stopped_at" = '2020-09-19
    17:41:07.170222+0200', "updated_at" = '2020-09-19 17:41:07.170743+0200', "status"
     = 'failure', "error" = '{"exception":"ZeroDivisionError","message":"divided by
    0","backtrace":["/tasks.rb:18:in `/'","/tasks.rb:18:in `execute'","/task_engine
    .rb:156:in `execute'","/task_engine.rb:131:in `block in initialize'"]}'::jsonb
    WHERE ("id" = 2)
(0.000560s) COMMIT
Worker 0 task 2 will be retried at 2020-09-19T17:42:07+02:00
(0.000142s) BEGIN
(0.000636s) UPDATE "tasks" SET "updated_at" = '2020-09-19 17:41:07.173521+0200', "
    scheduled_at" = '2020-09-19 17:42:07.172737+0200' WHERE ("id" = 2)
(0.000685s) COMMIT
```

Réessayer mais pas tout le temps

Comme je le disais plus haut, parfois il n'est pas nécessaire de réessayer une tâche qui a échoué, et parfois il serait néfaste de le faire :

- Certains types de tâches ne sont pas compatibles du tout avec le réessai.
- Lors de l'exécution, certaines situations ne doivent pas donner lieu à des réessai, il faut donc pouvoir spécifier que quand cela arrive, il ne faut plus faire de ressais.

Pour les types de tâches qui ne doivent jamais être réessayer, cette information est liée au type de la tâche, et l'idéal est de pouvoir le spécifier au niveau de la classe qui implémente la classe, plutôt que lorsqu'on créé la tâche.

Pour cela je vais utiliser une approche inspirée de Java, en utilisant un module sans méthode. Ce module servira à marquer que ce type de tâche n'est jamais à relancer.

Utiliser un module plutôt qu'une classe évite d'ajouter une contrainte sur les classes qui implémentent les tâches en les empêchant d'hériter d'une autre classe.

Voici le module :

task_engine.rb.

```
module TaskEngine
  # …

  # Used to tag tasks classes that should not be retried
  module NoRetryTask
  end
end
```

Et voici une tâche qui va échouer et qui utilise ce module :

tasks.rb.

```
module TaskEngine
  module Tasks
    class FailingNotRetryTask
      include TaskEngine::NoRetryTask

      # @param [Hash] parameters
      # @return [Hash, nil]
      def execute(parameters)
        1 / 0
      end
    end
  end
end
```

Lors d'une exécution en erreur, je vais vérifier si la tâche en question inclus ce module :

task_engine.rb.

```
module TaskEngine
  class Worker
    def execute
      # …
      rescue Exception => e
        # …
        if (!task_instance.nil?) && (task_instance.is_a?(TaskEngine::NoRetryTask))
          LOGGER.info("Worker #{@worker_index} task #{task.id} should not be retried"
              )
          DB.transaction do
            task.update(
                status: Task::STATUS_FAILED
            )
          end
        else
          current_retry_number = task.context['retry_number'] || 0
          # …
        end
```

```
      end
    end
  end
end
```

Ce qui donne, en lançant une `TaskEngine::Tasks::FailingNotRetryTask` :

```
Worker 2 starting task 3
(0.000189s) BEGIN
(0.001801s) INSERT INTO "task_executions" ("task_id", "started_at", "status", "
    created_at", "updated_at") VALUES (3, '2020-09-19 17:59:19.561146+0200', 'running
    ', '2020-09-19 17:59:19.562535+0200', '2020-09-19 17:59:19.562535+0200')
    RETURNING *
(0.000665s) COMMIT
Worker 2 failed task 3 in 4.8260000000000005ms, error is #<ZeroDivisionError: divided
     by 0>
(0.000219s) BEGIN
(0.001036s) UPDATE "task_executions" SET "stopped_at" = '2020-09-19
    17:59:19.565972+0200', "updated_at" = '2020-09-19 17:59:19.566692+0200', "status"
     = 'failure', "error" = '{"exception":"ZeroDivisionError","message":"divided by
    0","backtrace":["/tasks.rb:28:in `/''","/tasks.rb:28:in `execute'","/task_engine
    .rb:161:in `execute'","/task_engine.rb:135:in `block in initialize'"]}'::jsonb
    WHERE ("id" = 3)
(0.000653s) COMMIT
Worker 2 task 3 should not be retried
(0.000154s) BEGIN
(0.000879s) UPDATE "tasks" SET "status" = 'failed', "updated_at" = '2020-09-19
    17:59:19.569792+0200' WHERE ("id" = 3)
```

Quand une tâche qui en principe devrait être réessayée ne devrait pas l'être dans certain cas, je vais m'appuyer sur l'exception qui est remontée en cas d'erreur. Comme pour les tâche, je vais utiliser un module pour permettre de marquer les classes d'exception dans ce cas.

task_engine.rb.

```
module TaskEngine
  # …

  # Used to tag exception classes that should not trigger a retry when they happen
  module NoRetryExceptionModule
  end
end
```

Et voici une tâche qui va échouer en utilisant une exception de ce type :

tasks.rb.

```
module TaskEngine
  module Tasks
    class FailingNotRetryExceptionTask
      class FailingNotRetryTaskException < Exception
```

```
        include TaskEngine::NoRetryException
      end

      # @param [Hash] parameters
      # @return [Hash, nil]
      def execute(parameters)
        raise FailingNotRetryTaskException.new('Oh no')
      end
    end
  end
end
```

La gestion de ce cas ressemble beaucoup au précédent :

task_engine.rb.

```
module TaskEngine
  class Worker
    def execute
      # …
      rescue Exception => e
        # …
        if (!task_instance.nil?) && (task_instance.is_a?(TaskEngine::NoRetryTask))
          # …
        elsif e.is_a?(NoRetryException)
          LOGGER.info("Worker #{@worker_index} task #{task.id} raised a #{e.class} so
              it should not be retried")
          DB.transaction do
            task.update(
                status: Task::STATUS_FAILED
            )
          end
        else
          current_retry_number = task.context['retry_number'] || 0
          # …
        end
      end
    end
  end
end
```

Ce qui donne, en lançant une `TaskEngine::Tasks::FailingNotRetryExceptionTask` :

```
Worker 0 starting task 6
(0.000201s) BEGIN
(0.001445s) INSERT INTO "task_executions" ("task_id", "started_at", "status", "
    created_at", "updated_at") VALUES (6, '2020-09-19 18:22:36.700020+0200', 'running
    ', '2020-09-19 18:22:36.701062+0200', '2020-09-19 18:22:36.701062+0200')
    RETURNING *
(0.000640s) COMMIT
Worker 0 failed task 6 in 4.019ms, error is #<TaskEngine::Tasks::
```

```
    FailingNotRetryExceptionTask::FailingNotRetryTaskException: Oh no>
(0.000173s) BEGIN
(0.000946s) UPDATE "task_executions" SET "stopped_at" = '2020-09-19
    18:22:36.704039+0200', "updated_at" = '2020-09-19 18:22:36.704737+0200', "status"
     = 'failure', "error" = '{"exception":"TaskEngine::Tasks::
    FailingNotRetryExceptionTask::FailingNotRetryTaskException","message":"Oh no","
    backtrace":["/tasks.rb:40:in `execute'","/task_engine.rb:165:in `execute'","/
    task_engine.rb:139:in `block in initialize'"]}'::jsonb WHERE ("id" = 6)
(0.000556s) COMMIT
Worker 0 task 6 raised a TaskEngine::Tasks::FailingNotRetryExceptionTask::
    FailingNotRetryTaskException so it should not be retried
(0.000151s) BEGIN
(0.000703s) UPDATE "tasks" SET "status" = 'failed', "updated_at" = '2020-09-19
    18:22:36.707380+0200' WHERE ("id" = 6)
I
```

Ce qui boucle la gestion des cas d'erreurs{nsbp} : les erreurs sont enregistrées, les tâches sont par défaut relancées, mais il est possible de gérer de manière fine les situation où cela n'est pas souhaitable.

Conclusion

Et voila c'est terminé, le moteur de tâches est fonctionnel, ou du moins il a toute les fonctionnalités de base qu'on attend à trouver dans ce genre d'outils.

Par rapport à un moteur de tâches plus complet il manque encore de nombreuses choses comme :

- Du monitoring de bout en bout, en utilisant par exemple des identifiant de corrélation transmis d'une tâche à l'autre et qui pourrait utiliser le contexte des tâches.
- Une gestion de la priorité, par exemple en ajoutant une colonne permettant de définir un niveau de priorité pour que certaines tâches ou certains types de tâches s'exécutent avant d'autres.
- La capacité d'enrichir le moteur à l'aide de hooks pour avoir notamment des pré-traitement et des post-traitements.

Mais ces ajouts ne changeraient pas la structure du code, et leur implémentation n'aiderait probablement à mieux comprendre ou mieux utiliser un moteur de tâches, elles sont donc en dehors du périmètre que j'ai choisi de traiter.

J'espère que la lecture aura permis de démystifier les moteur de tâches, et qu'elle en aura fait un outil "comme les autres" dont vous n'hésiterez pas à vous servir.

Si vous avez des remarques, des critiques ou des questions, contactez-moi !

Annexe : le code source

Voici le code source complet de l'outil. Il est également disponible en ligne ici.

Gemfile.

```
source 'https://rubygems.org'

gem 'rake', '~> 13.0'
gem 'sequel', '~> 5.34'
gem 'pg', '~> 1.2.3'
```

task_engine.rb.

```
require 'date'
require 'logger'
require 'sequel'

module TaskEngine

  LOGGER = Logger.new(STDOUT)
  NOTIFICATION_CHANNEL = 'task-engine'
  WORKERS_NUMBER = 5
  # Database connexion path
  DATABASE_URL = 'postgres://task_engine@localhost/task_engine'
  # Open the connexions to the database
  # Use at most
  # - one connexion per worker
  # - one connexion for notifications
  # - one connexion for the heartbeat
  DB = ::Sequel.connect(DATABASE_URL, max_connections: WORKERS_NUMBER + 2, logger:
      LOGGER)
  DB.extension :pg_json

  # The `created_at` and `updated_at` attributes should be managed by Sequel
  Sequel::Model.plugin :timestamps, force: true, update_on_create: true

  # Used to tag tasks classes that should not be retried
  module NoRetryTask
  end

  # Used to tag exception classes that should not trigger a retry when they happen
  module NoRetryException
```

```
end

class Task < Sequel::Model
  STATUS_WAITING = 'waiting'
  STATUS_RUNNING = 'running'
  STATUS_STOPPED = 'stopped'
  STATUS_FAILED = 'failed'
  one_to_many :task_executions
end

class TaskExecution < Sequel::Model
  STATUS_RUNNING = 'running'
  STATUS_SUCCESS = 'success'
  STATUS_FAILURE = 'failure'
  many_to_one :task
end

MILLISECONDS_IN_A_DAY = 24 * 60 * 60 * 1000

# @param [String] task_class
# @param [Hash] task_parameters
# @param [DateTime, nil] scheduled_at nil means an immediate execution
# @param [Hash] task_context
def self.create_task(task_class, task_parameters, scheduled_at = nil)
  creation_params = {
      task_class: task_class,
      task_parameters: Sequel.pg_json_wrap(task_parameters)
  }
  # Don't send a null value to the DB so the DB use the default value
  unless scheduled_at.nil?
    creation_params[:scheduled_at] = scheduled_at
  end
  Task.create(creation_params)
  if scheduled_at.nil?
    DB.notify(NOTIFICATION_CHANNEL, payload: task_class)
  end
end

class Engine
  ENGINE_STATUS_RUNNING = 'running'
  ENGINE_STATUS_STOPPING = 'stopping'

  attr_reader :instance, :status, :queue

  def initialize(instance)
    unless Task.where(
        status: ENGINE_STATUS_RUNNING,
        instance: instance).empty?
      raise "Found running tasks with same instance name in the database [#{
          instance}]"
    end
```

```
      @instance = instance
      @queue = Queue.new
      @workers = []
      LOGGER.info('Starting engine')
      @status = ENGINE_STATUS_RUNNING

      Signal.trap('SIGTERM') do
        signal_trap
      end
      Signal.trap('SIGINT') do
        signal_trap
      end

      # Notifications
      Thread.new do
        DB.listen(NOTIFICATION_CHANNEL, loop: true) do |_channel, _pid, task_class|
          LOGGER.info("Received notification for a task [#{task_class}]")
          unless @queue.num_waiting == 0
            LOGGER.info('Notifying worker')
            @queue.push(task_class)
          end
        end
      end

      # Scheduled wake-ups
      Thread.new do
        while true do
          LOGGER.info('Scheduled wake-up')
          @queue.push('Scheduled wake-up')
          sleep(10)
        end
      end

      0.upto(WORKERS_NUMBER - 1) do |worker_index|
        @workers << Worker.new(self, worker_index)
      end
      @workers.each do |worker|
        LOGGER.info("Joining worker #{worker.worker_index}")
        worker.thread.join
      end
    end

    def signal_trap
      @status = ENGINE_STATUS_STOPPING
      @queue.close
    end
  end

  class Worker
    MAX_RETRIES_NUMBER = 10
```

```
    attr_reader :thread, :worker_index

    # @param [TaskEngine::Engine] engine
    # @param [Integer] worker_index
    def initialize(engine, worker_index)
      @engine = engine
      @worker_index = worker_index
      LOGGER.info("Starting worker #{worker_index}")
      @thread = Thread.new do
        execute
      end
    end

    private

    def execute
      while @engine.status == Engine::ENGINE_STATUS_RUNNING
        while (@engine.status == Engine::ENGINE_STATUS_RUNNING) && (task =
            try_acquire_task)
          starting_time = DateTime.now
          LOGGER.info("Worker #{@worker_index} starting task #{task.id}")

          task_execution = nil
          DB.transaction do
            task_execution = TaskExecution.create(
                task: task,
                started_at: starting_time,
                status: TaskExecution::STATUS_RUNNING
            )
          end

          task_class_name = task.task_class
          task_instance = nil
          begin
            task_class = Object.const_get(task_class_name)
            task_instance = task_class.new
            task_result = task_instance.execute(task.task_parameters)

            stopping_time = DateTime.now
            elapsed_time = (stopping_time - starting_time).to_f *
                MILLISECONDS_IN_A_DAY
            LOGGER.info("Worker #{@worker_index} finished successfully task #{task.id
                } in #{elapsed_time}ms, result is #{result}")
            DB.transaction do
              task_execution.update(
                  stopped_at: stopping_time,
                  result: Sequel.pg_json_wrap(task_result),
                  status: TaskExecution::STATUS_SUCCESS
              )
              task.update(
                  status: Task::STATUS_STOPPED
```

```
            )
          end
        rescue Exception => e
          stopping_time = DateTime.now
          elapsed_time = (stopping_time - starting_time).to_f *
              MILLISECONDS_IN_A_DAY
          LOGGER.info("Worker #{@worker_index} failed task #{task.id} in #{
              elapsed_time}ms, error is #{e.inspect}")
          DB.transaction do
            task_execution.update(
                stopped_at: stopping_time,
                error: Sequel.pg_json_wrap(
                    {
                        exception: e.class,
                        message: e.message,
                        backtrace: e.backtrace
                    }),
                status: TaskExecution::STATUS_FAILURE
            )
          end
          if (!task_instance.nil?) && (task_instance.is_a?(TaskEngine::NoRetryTask)
              )
            LOGGER.info("Worker #{@worker_index} task #{task.id} should not be
                retried")
            DB.transaction do
              task.update(
                  status: Task::STATUS_FAILED
              )
            end
          elsif e.is_a?(NoRetryException)
            LOGGER.info("Worker #{@worker_index} task #{task.id} raised a #{e.class
                } so it should not be retried")
            DB.transaction do
              task.update(
                  status: Task::STATUS_FAILED
              )
            end
          else
            current_retry_number = task.context['retry_number'] || 0
            if current_retry_number < MAX_RETRIES_NUMBER
              task.context['retry_number'] = current_retry_number + 1
              # 2^current_retry_number numbers of minutes / number of minutes in a
                  day
              retry_scheduled_at = DateTime.now + Rational(2 **
                  current_retry_number, 1440)
              LOGGER.info("Worker #{@worker_index} task #{task.id} will be retried
                  at #{retry_scheduled_at}")
              DB.transaction do
                task.update(
                    status: Task::STATUS_WAITING,
                    context: task.context,
```

```
                    scheduled_at: retry_scheduled_at
                  )
                end
              else
                LOGGER.info("Worker #{@worker_index} task #{task.id} retries
                    exhausted")
                DB.transaction do
                  task.update(
                      status: Task::STATUS_FAILED
                  )
                end
              end
            end
          end
        end
        # Wait until the notification awakes the worker
        LOGGER.info("Worker #{@worker_index} is sleeping")
        # We don't care about the message we got from the queue
        # since we just want to be awaken
        @engine.queue.pop
        LOGGER.info("Worker #{@worker_index} is awaken")
      end
      LOGGER.info("Worker #{@worker_index} is stopping")
    end

    # @return [TaskEngine::Task, nil]
    def try_acquire_task
      DB.transaction do
        task = Task.
            where(status: Task::STATUS_WAITING).
            where(Sequel.lit('scheduled_at < LOCALTIMESTAMP')).
            order(:scheduled_at).for_update.first
        unless task.nil?
          Task.where(id: task.id).update(
              status: Task::STATUS_RUNNING,
              instance: @engine.instance
          )
          task
        end
      end
    end
  end
end
```

Rakefile.

```
desc 'Start the engine'

TASK_ENGINE_INSTANCE = 'TASK_ENGINE_INSTANCE'

task :start_engine do
```

```
  unless ENV.key?(TASK_ENGINE_INSTANCE)
    raise "Missing env variable #{TASK_ENGINE_INSTANCE}"
  end
  instance = ENV.fetch(TASK_ENGINE_INSTANCE)
  require_relative 'task_engine'
  require_relative 'tasks'
  TaskEngine::Engine.new(instance)
end

desc 'Create tasks'
task :create_tasks do
  require_relative 'task_engine'
  100.times do
    TaskEngine.create_task(
        'TaskEngine::Tasks::WaitingTask',
        {
            waiting_time: 5
        })
  end
end

task default: :start_engine
```

tasks.rb.

```
module TaskEngine
  module Tasks
    class WaitingTask
      PARAMETER_WAITING_TIME = 'waiting_time'

      # @param [Hash] parameters
      # @return [Hash, nil]
      def execute(parameters)
        sleep(parameters[PARAMETER_WAITING_TIME])
        nil
      end
    end

    class FailingTask
      # @param [Hash] parameters
      # @return [Hash, nil]
      def execute(parameters)
        1 / 0
      end
    end

    class FailingNotRetryTask
      include TaskEngine::NoRetryTask

      # @param [Hash] parameters
      # @return [Hash, nil]
```

```
      def execute(parameters)
        1 / 0
      end
    end

    class FailingNotRetryExceptionTask
      class FailingNotRetryTaskException < Exception
        include TaskEngine::NoRetryException
      end

      # @param [Hash] parameters
      # @return [Hash, nil]
      def execute(parameters)
        raise FailingNotRetryTaskException.new('Oh no')
      end
    end
  end
end
```

migrations/01_create_tasks.rb.

```
Sequel.migration do
  change do
    # Load enum-related methods
    extension :pg_enum

    # Declare the enum with the list of allowed values
    create_enum(:task_status, ['waiting', 'running'])

    create_table(:tasks) do
      primary_key :id
      # The status column is using the task_status enum type
      task_status :status, null: false, default: 'waiting'

      DateTime :created_at, null: false
      DateTime :updated_at, null: false
    end
  end
end
```

migrations/02_add_instance_name.rb.

```
Sequel.migration do
  change do
    alter_table(:tasks) do
      add_column :instance, String, null: true, text: true
    end
  end
end
```

migrations/03_add_parameters.rb.

```ruby
Sequel.migration do
  change do
    alter_table(:tasks) do
      add_column :task_class, String, null: false, text: true
      add_column :task_parameters, 'JSONB', null: false
    end
  end
end
```

migrations/04_create_task_executions.rb.

```ruby
Sequel.migration do
  change do
    add_enum_value(:task_status, 'stopped')

    create_table(:task_executions) do
      primary_key :id
      foreign_key :task_id, :tasks, null: false
      DateTime :started_at, null: false
      DateTime :stopped_at, null: true
      column :result, 'JSONB', null: true

      DateTime :created_at, null: false
      DateTime :updated_at, null: false
    end
  end
end
```

migrations/05_add_scheduled_at.rb.

```ruby
Sequel.migration do
  change do
    alter_table(:tasks) do
      add_index [:status, :created_at]
    end
  end
end
```

migrations/06_add_scheduled_at.rb.

```ruby
Sequel.migration do
  change do
    alter_table(:tasks) do
      add_column :scheduled_at, DateTime, null: false, default: Sequel.lit('
          LOCALTIMESTAMP')
      drop_index [:status, :created_at]
      add_index [:status, :scheduled_at]
    end
  end
end
```

migrations/07_add_errors_to_task_execution.rb.

```
Sequel.migration do
  change do
    extension :pg_enum
    create_enum(:task_execution_status, ['running', 'success', 'failure'])
    alter_table(:task_executions) do
      add_column :status, :task_execution_status, null: false
      add_column :error, 'JSONB', null: true
    end
  end
end
```

migrations/08_add_task_context_and_task_failure.rb.

```
Sequel.migration do
  change do
    extension :pg_enum
    add_enum_value(:task_status, 'failed')

    extension :pg_json
    alter_table(:tasks) do
      add_column :context, 'JSONB', null: false, default: Sequel.pg_json_wrap({})
    end
  end
end
```