SCIENCES SORBONNE UNIVERSITÉ

Master 1 SAR - UE SRCS
Projet de fin de semestre
Moteur d'exécution de workflows

Jonathan Lejeune

Mai 2020

Objectif

L'objectif ultime de ce projet est de produire un service distribué permettant d'exécuter des workflows sur un ensemble de nœuds de calcul. Un workflow peut être vu comme un ensemble de tâches définies statiquement (c'est à dire avant l'exécution) avec des dépendances entre elles. Un workflow se modélise comme un graphe orienté acyclique où les sommets de ce graphe représentent les tâches du workflow et un arc partant d'une tâche A vers une tâche B signifie que B ne peut s'exécuter une fois que A s'est terminée. En revanche deux tâches qui n'ont pas de lien de dépendance peuvent s'exécuter en parallèle.

La figure 1 donne un exemple de workflow. On peut remarquer que :

- T_0, T_1 et T_4 peuvent s'exécuter en parallèle dès le départ car elles ne dépendent d'aucune autre tâche.
- T_2 ne peut s'exécuter qu'une fois que T_0 et T_1 se soient terminées.
- T_5 ne peut s'exécuter qu'une fois que T_4 se soit terminée.
- T_3 ne peut s'exécuter qu'une fois que T_2 et T_5 se soient terminées.
- T_6 ne peut s'exécuter qu'une fois que T_5 se soit terminée.

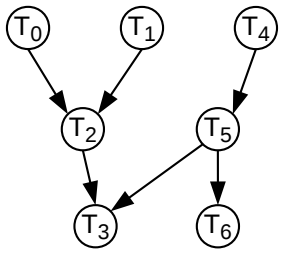

FIGURE 1 – Exemple de Workflow avec 7 tâches et 6 dépendances

Consignes générales

- Ce travail est à faire **seul**. Toute détection de similitude improbable entre deux rendus entrainera une note nulle.
- Les livrables sont à rendre sur Moodle
- Il est attendu de rendre 2 fichiers :
  - ◇ Une archive de **vos sources java**. Il est inutile de rendre les fichiers de test unitaire qui vous sont fournis.
  - ◇ Un fichier pdf dans lequel vous devez répondre aux questions posées dans le sujet. **Il attendu des réponses synthétiques, claires, pédagogiques et relues**.

- Au niveau programmation, ce sujet est incrémental c'est- à-dire que vous devez valider les tests unitaires d'un exercice pour passer au suivant.
- Les tests JUnit compileront au fur et à mesure de votre avancement dans le sujet. Il vous est donc conseillé de les intégrer dans votre projet Eclipse progressivement.
- L'environnement de correction sera : Ubuntu 18.04, Java 8, Eclipse 2020-03 et JUnit 4. Assurez-vous que votre environnement de développement soit compatible.

Exercice 1 – Définition de graphes

Un workflow étant caractérisé par un graphe dirigé acyclique de tâches , il est donc nécessaire de pouvoir définir un graphe. Dans cet exercice, il est demandé de produire une interface `Graph<T>` et sa classe d'implantation `GraphImpl<T>`. Le choix d'implantation est libre, le seul objectif est de respecter les spécifications de l'interface `Graph<T>`. La variable de type `T` désigne le type d'un nœud du graphe. Cette interface est composée des méthodes suivantes :

- `void addNode(T n);` : ajoute un nouveau nœud dans le graphe. Si le nœud existe déjà dans le graphe, une `IllegalArgumentException` est jetée.
- `void addEdge(T from, T to);` : ajoute un nouveau lien unidirectionnel partant du nœud `from` et arrivant au nœud `to`. Une `IllegalArgumentException` est jetée si ni `from`, ni `to` n'existent ou bien si le lien existe déjà.
- `boolean existEdge(T from, T to);` : teste l'existence d'un arc entre `from` et `to` dans le graphe
- `boolean existNode(T n);` : teste l'existence d'un nœud dans le graphe.
- `boolean isEmpty();` : teste si le graphe ne contient aucun nœud.
- `int size();` : renvoie nombre de nœuds présents dans le graphe
- `List<T> getNeighborsOut(T from);` : renvoie la liste des nœuds voisins du nœud `from` via les arcs sortants. Une `IllegalArgumentException` est jetée si `from` n'existe pas dans le graphe.
- `List<T> getNeighborsIn(T to);` : renvoie la liste des nœuds voisins du nœud `to` via les arcs entrants. Une `IllegalArgumentException` est jetée si `to` n'existe pas dans le graphe.
- `Set<T> accessible(T from);` : renvoie la liste des nœuds accessibles à partir de `from`. Une `IllegalArgumentException` est jetée si `from` n'existe pas dans le graphe.
- `boolean isDAG();` : teste si le graphe est acyclique. On notera que ceci est vrai si aucun des nœuds ne peut être accessible à partir de lui-même.

On notera également que l'interface `Graph<T>` étend l'interface `Iterable<T>` de l'API Java permettant ainsi de parcourir l'ensemble des nœuds du graphe dans une boucle foreach.

Synthèse de l'exercice :

- **But** : pouvoir définir un graphe.
- **Package de travail** : `srcs.workflow.graph`
- **Fichiers attendus** : *Graph.java* et *GraphImpl.java*
- **Test JUnit de validation** : `TestGraph`

Exercice 2 – Définition de Jobs

Maintenant que l'on peut définir un graphe d'objets quelconques, il est maintenant nécessaire de pouvoir offrir à l'utilisateur un moyen d'exprimer son calcul. Dans un premier temps, nous allons définir une classe abstraite `Job` qui est la classe mère de n'importe quel calcul. Dans un deuxième temps nous allons offrir des outils pour que l'utilisateur puisse définir les tâches du job ainsi que leurs dépendances. Enfin dans un dernier temps, nous allons offrir un utilitaire permettant de vérifier que l'expression du Job par l'utilisateur est conforme.

Définition de la classe `Job`

La classe Job est abstraite et se définit par deux attributs constants (`final`) :
- un nom de type `String` accessible via le getter `getName()`
- un contexte de type `Map<String,Object>` accessible via le getter `getContext()` et qui permet d'associer un nom de paramètre du job à une valeur. Cet attribut permet de paramétrer le job, utile notamment pour paramétrer les tâches racines du graphe.

Définition d'une API d'expression de tâches

Pour exprimer un workflow, l'utilisateur devra définir une classe qui étend la classe `Job` définie précédemment. Le code des tâches du job sera défini dans les méthodes d'instance de la classe qui sont annotées par l'annotation `Task`. Cette annotation prend un argument obligatoire de type `String` et qui désigne l'identifiant de la tâche au sein du job. Les paramètres d'une méthode annotée par `Task` sont les paramètres de la tâche. Un paramètre d'une tâche peut être :

1) soit une donnée définie dans le contexte du job. Dans ce cas le paramètre devra être annoté par l'annotation `Context`. Cette annotation possède un attribut obligatoire de type string qui référence une valeur dans le contexte du job.
2) soit le résultat d'une tâche précédente : c'est ainsi que l'on définit les dépendances entre tâches. Dans ce cas, le paramètre devra être annoté par l'annotation `LinkFrom`. Cette annotation possède un attribut obligatoire de type string qui référence l'identifiant de la tâche précédente censée délivrer le résultat.

La classe `ArithmeticJob` (fournie) est un exemple de job conforme qui sera utilisé par la suite pour les tests.

Définition d'un validateur

Pour être certain qu'il n'y ait pas d'incohérence dans la définition du Job, il est nécessaire de définir une classe qui valide la conformité d'un job. Un job est conforme s'il respecte les contraintes suivantes :
- il existe au moins une méthode annotée `Task`
- les méthodes annotées `Task` doivent être des méthodes d'instance
- les méthodes annotées `Task` ne doivent pas renvoyer `void`
- deux méthodes annotées `Task` ne doivent pas avoir le même identifiant de tâche
- tout paramètre d'une méthode annotée `Task` doit être soit annoté par `Context` ou par `LinkFrom`
- toute annotation `LinkFrom` doit référencer une tâche existante
- toute annotation `Context` doit référencer un objet existant dans le contexte du job
- il doit y avoir une compatibilité de type entre un paramètre annoté `LinkFrom` et le retour de la méthode correspondante.
- il doit y avoir une compatibilité de type entre un paramètre annoté `Context` et l'objet correspondant dans le contexte
- le graphe de tâches doit être acyclique

Pour ce faire nous allons programmer la classe `JobValidator` qui offre :
- un constructeur qui prend un `Job` en paramètre et qui jette une `ValidationException` (classe à définir) si le job n'est pas conforme. La construction d'un `JobValidator` ne peut donc aboutir si le job passé en paramètre n'est pas conforme.
- la méthode `Graph<String> getTaskGraph()` qui renvoie le graphe de tâches correspondant au job. Dans le graphe les tâches sont référencées par leur identifiant.
- la méthode `Method getMethod(String id)` qui pour un identifiant de tâche donné renvoie sa méthode dans la classe d'implantation du job.
- la méthode `Job getJob()` qui renvoie le job associé et validé.

Synthèse de l'exercice :
- **But** : pouvoir définir un graphe de tâches conforme
- **Package de travail** : `srcs.workflow.job`
- **Fichiers attendus** : *Job.java*, *Task.java*, *Context.java*, *LinkFrom.java*, *JobValidator.java*, *ValidationException.java*
- **Test JUnit de validation** : `TestJobValidator`
- **Autre condition de validation** : les classes `JobForTest`, `JobTest`, `JobTests` et `ArithmeticJob` doivent compiler.
- **Question**

1. Pourquoi ne peut-on pas utiliser le nom de la méthode en tant qu'identifiant de tâche ?

Exercice 3 – Exécution séquentielle en local

Maintenant que nous pouvons exprimer des jobs conformes, nous allons pouvoir les exécuter. Nous allons définir une classe abstraite `JobExecutor` qui possède un attribut de type `Job` (renseigné lors de l'appel au constructeur) et qui offre la méthode abstraite `Map<String,Object> execute()throws Exception`. L'appel à cette méthode exécute le job et renvoie une map qui associe pour chaque tâche son résultat.

La première implantation de `JobExexutor` sera la classe `JobExecutorSequential` qui permet d'exécuter un job séquentiellement sur la machine locale. Pour exécution séquentielle nous entendons une exécution monothreadée qui exécute les tâches séquentiellement en respectant l'ordre des dépendances.

Synthèse de l'exercice :
- **But** : exécuter un job séquentiellement sur la machine locale
- **Package de travail** : `srcs.workflow.executor`
- **Fichiers attendus** : *JobExecutor.java*, *JobExecutorSequential.java*
- **Test JUnit de validation** : `TestJobLocalSequential`
- **Question** :

1. Donner les grandes lignes de votre algorithme de la méthode `execute`. Il est attendu un algorithme décrit de manière synthétique et non un copier/coller de votre code.

Exercice 4 – Exécution parallèle en local

Nous allons à présent écrire `JobExecutorParallel`, une nouvelle implantation de `JobExecutor` qui permet d'exécuter le job en local mais en multithreadé. On considérera qu'un thread est dédié à une tâche. Votre solution ne doit pas empêcher que deux tâches indépendantes puissent s'exécuter parallèlement.

Synthèse de l'exercice :
- **But** : exécuter un job sur la machine locale
- **Package de travail** : `srcs.workflow.executor`
- **Fichiers attendus** : *JobExecutorParallel.java*
- **Test JUnit de validation** : `TestJobLocalParallel`
- **Question** :

1. Décrire comment vous avez assuré le parallélisme des tâches indépendantes.

Exercice 5 – Exécution à distance sur un serveur central

Nous allons maintenant déporter l'exécution des jobs sur un serveur. Dans le cadre des tests de ce projet, ce serveur sera déployé dans un autre JVM sur votre machine locale. Sur le même principe que l'exécuteur de l'exercice précédent, le serveur exécutera parallèlement chaque job reçu. Dans cet exercice vous devrez fournir :

- l'implantation `JobExecutorRemoteCentral` de `JobExecutor` qui sera le code client du serveur.
- la classe `JobTrackerCentral` qui offrira une méthode `public static void main(String args [])` et qui déploiera le service d'exécution de job. Ce programme java n'attend pas d'argument (`args.length = 0`).

Dans un deuxième temps, vous étendrez votre solution pour permettre au client d'avoir un état d'avancement sur l'exécution de son job. Le serveur devra donc renvoyer au client d'un job donné le nombre de tâches totales qui ont terminé leur exécution. Dans votre implantation, il est attendu que le client écrive la valeur reçue sur sa sortie standard. Vous pourrez en effet remarquer dans le test `TestJobRemoteCentralFeedback` que la sortie standard sera temporairement redirigée vers un fichier qui servira de contrôler cette fonctionnalité.

Synthèse de l'exercice :

- **But** : exécuter un job sur un serveur distant
- **Package de travail** : `srcs.workflow.executor.central`
- **Fichiers attendus** : *JobTrackerCentral.java*, *JobExecutorRemoteCentral.java*
- **Tests JUnit de validation** : `TestJobRemoteCentral`, `TestJobRemoteCentralFeedback`
- **Questions** :
  1. Décrire votre protocole de communication
  2. Justifier votre choix de l'API de communication que vous avez utilisée.
  3. Quelle hypothèse doit-on faire sur le type des objets du contexte ? Justifiez.
  4. Quel mécanisme avez-vous utilisé pour la notification d'avancement au client ?

Exercice 6 – Exécution à distance sur un cluster de machines

La version ultime de l'exécuteur sera une exécution des tâches de tout job sur un ensemble de serveurs. Il est supposé qu'il existe une JVM maître (classe principale `JobTrackerMaster`) dont le rôle sera de recevoir les jobs des clients et d'attribuer les tâches sur un ensemble de JVM esclaves (classe principale `TaskTracker`) qui les exécuteront. La JVM maître n'exécute aucune tâche et a juste un rôle de coordinateur qui assure une répartition équitable des tâches entre les esclaves. Une JVM esclave prend deux arguments :

- un nom de type String qui fera office d'identifiant (on suppose que deux esclaves ne peuvent pas avoir le même nom)
- un nombre maximal de tâches qu'elle peut exécuter à un instant donné. Il faudra s'assurer que le nombre de tâches en cours d'exécution sur l'esclave ne dépasse pas cette borne. Il faut également considérer que la valeur de ce paramètre est propre à un esclave et peut être différente d'un esclave à l'autre.

La classe cliente sera une implantation de `JobExecutor` nommée `JobExecutorRemoteDistributed`. Vos choix d'implantation sont libres.

Votre solution sera testée selon trois niveaux d'évolution :

- **Niveau 1** : votre solution est capable d'exécuter des jobs un par un, c'est-à-dire qu'il n'est pas possible d'exécuter un job tant qu'il en existe un en cours d'exécution.
- **Niveau 2** : votre solution est capable d'exécuter plusieurs jobs en même temps sur le système.
- **Niveau 3** : votre solution a atteint le niveau 2 et est tolérante aux pannes des esclaves. Le maître doit être capable de détecter la panne d'un esclave et de réaffecter les tâches perdues ailleurs. On supposera pour simplifier que le maître ne peut pas tomber en panne et que toute panne d'esclave est définitive (pas de redémarrage possible).

Synthèse de l'exercice :

- **But** : exécuter un job sur un ensemble de nœuds de calcul
- **Package de travail** : `srcs.workflow.executor.distributed`
- **Fichiers attendus** : *JobTrackerMaster.java*, *TaskTracker.java*, *JobExecutorRemoteDistributed.java*
- **Tests JUnit de validation** :
  - ◇ `TestJobRemoteDistributed` (niveau 1)
  - ◇ `TestJobRemoteDistributedRafale` (niveau 2)
  - ◇ `TestJobRemoteDistributedRafaleWithCrashes` (niveau 3)
- **Questions** :

1. Quel est le protocole de communications entre maître-esclaves et éventuellement entre esclave-esclave ?
2. Comment le maître affecte ses tâches équitablement sur les esclaves ?
3. Comment les esclaves sont assurés de respecter leur borne de tâche courante ?
4. Comment gérez-vous la communication d'un résultat entre deux tâches dépendantes ? Donner les avantages et les inconvénients de votre solution.
5. (niveau 2) Comment gérez-vous le fait d'avoir plusieurs jobs en cours d'exécution sur le cluster ?
6. (niveau 3) Décrire le mécanisme qui permet de détecter la panne d'un esclave. Comment gérez-vous la réaffectation des tâches perdues ?