3259.1 Paradigmes de program. avancés II– Rapport technique – ISC3il-b

|  |
| --- |
| **Laboratoire 1** |

|  |
| --- |
| Étudiants participant à ce travail :  **Nicolas Aubert, ISC3il-b**  Présenté à :  **Aïcha Rizzotti**  Restitution du rapport : **21.07.2023**  Période : **2023**  École : **HE-Arc, Neuchâtel** |

**Framework pour le monitoring de la concurrence en Java**

Table des matières

[1 - Glossaire 2](#_Toc132901275)

[2 - Introduction 4](#_Toc132901276)

[2.1 - Contexte 4](#_Toc132901277)

[3 - Structure du framework 5](#_Toc132901278)

[3.1 - Diagramme de classes 5](#_Toc132901279)

[3.2 - Classe Person 5](#_Toc132901280)

[3.3 - Classe Document 5](#_Toc132901281)

[3.4 - Classe Timer 5](#_Toc132901282)

[3.5 - Classe Log 5](#_Toc132901283)

[3.6 - Classe WaitingLogger 6](#_Toc132901284)

[3.7 - Déroulement du programme 6](#_Toc132901285)

[4 - Résultats 7](#_Toc132901286)

[4.1 - 4-2-2 7](#_Toc132901287)

[4.2 - 1-2-2 7](#_Toc132901288)

[4.3 - 1-0-4 7](#_Toc132901289)

[4.4 - 1-4-0 7](#_Toc132901290)

[5 - Limitations et perspectives 8](#_Toc132901291)

[6 - Conclusion 9](#_Toc132901292)

[7 - Annexes I](#_Toc132901293)

[7.1 - Guide utilisateur I](#_Toc132901294)

[7.2 - Cahier des charges II](#_Toc132901295)

[7.3 - Table des illustrations III](#_Toc132901296)

[7.4 - Bibliographies et références IV](#_Toc132901297)

[7.4.1 - Sites Web IV](#_Toc132901298)

[7.4.2 - Livres IV](#_Toc132901299)

[7.4.3 - Autres IV](#_Toc132901300)

1. Glossaire

|  |  |
| --- | --- |
| **Database** | Une classe qui représente la base de données de documents à traiter par les threads, et offre des méthodes pour initialiser la base de données, obtenir un document aléatoire et obtenir la liste des noms de documents stockés. |
| **Log** | Une classe qui représente une action concurrente à afficher, avec un type (WAITING, REMOVE ou FINISHED) et une référence à la personne concernée. |
| **Main** | Une classe qui représente le thread principal gérant le bon déroulement et le démarrage de l'application, avec une FutureTask contenant le déroulement du programme pouvant être arrêté sur demande, et une méthode pour générer une population de personnes avec des propriétés aléatoires et attribuer à chacune un document aléatoire de la base de données. |
| **Paradigme Lecteurs-Rédacteur** | Un modèle de synchronisation dans la programmation concurrente où plusieurs threads peuvent lire simultanément une ressource partagée, mais seulement un thread à la fois peut écrire (rédiger) sur cette ressource. |
| **Person** | Une classe qui représente à la fois les lecteurs et les rédacteurs, implémente l'interface Runnable pour être exécutée comme un thread, et offre des méthodes pour gérer l'activité du thread, la pause et la reprise du timer, et le temps passé à traiter le document. |
| **ReentrantReadWriteLock** | Une classe Java qui implémente le paradigme lecteurs-rédacteurs en fournissant un verrou réentrant permettant à plusieurs threads de lire simultanément une ressource partagée, mais n'autorisant qu'un seul thread à la fois à écrire sur cette ressource. |
| **Singleton** | Un modèle de conception en programmation où une classe ne peut avoir qu'une seule instance dans tout le programme, avec un point d'accès global à cette instance. |
| **Timer** | Une classe qui garde une trace du temps passé dans l'application de manière générale et non par thread concurrent. |
| **WaitingLogger** | Une classe qui stocke l'état actuel des files d'attente des threads sur les objets de synchronisation, et offre des méthodes pour ajouter, supprimer et traiter les logs de files d'attente. |

1. Introduction
   1. Contexte

Le monitoring de la concurrence est un enjeu majeur dans le développement d'applications multi-threadées en Java. Dans ce contexte, nous avons été chargés de concevoir un framework pour le monitoring des files d'attente sur les éléments de synchronisation, en utilisant le pattern Lecteurs-Rédacteurs. Ce rapport présente notre solution pour visualiser les files d'attente des différents objets de synchronisation, où des lecteurs et des rédacteurs partagent des documents.

Nous utilisons la classe java.util.concurrent.locks.ReentrantReadWriteLock pour assurer l'accès concurrent aux documents. Notre solution s'appuie sur plusieurs classes déjà présentes dans les sources, telles que WaitingLogger, Database, Document, Person, Timer et Main, pour implémenter le monitoring de la concurrence de manière efficace.

Dans ce rapport, nous détaillerons les objectifs pédagogiques de notre solution, ainsi que les problématiques à résoudre et les interfaces minimales des classes impliquées. Nous expliquerons également les choix de conception que nous avons faits pour l'interface utilisateur, et nous mettrons en évidence les fonctionnalités clés de notre solution de monitoring de la concurrence.

1. Présentation du framework
   1. Diagramme de classes
   2. Classe Person

Cette classe représente une entité (personne) dans le programme. Lors de la création d’une personne, celle-ci se verra attribuer un document à traiter, ainsi qu’un type, pouvant prendre comme valeurs : Reader ou Writer.

Une personne de type Reader se verra charger de lire le contenu du document qui lui a été attribué. Inversement, une personne de Writer devra modifier le contenu de son document.

* 1. Classe Document

Représente une ressource à laquelle les personnes vont accéder, en lecture ou en écriture, pour effectuer des opérations.

Dans une application lambda utilisant ce schéma, la gestion de la concurrence aurait été effectuée dans les méthodes getDocumentContentou setDocumentContent de la classe Document. Cependant, dans ce projet, ce sont les lecteurs et rédacteurs qui doivent gérer cette concurrence. Pour cela, chaque document met à un disposition un verrou (lock) pour l’écriture et un verrou pour la lecture. Ces verrous proviennent d’une instance de la classe ReentrantReadWriteLock, provenant du package java.util.concurrent.locks, propre à chaque document.

* 1. Classe Timer

Cette classe permet principalement d’obtenir le temps écoulé depuis le lancement du programme. Cette classe est un singleton, cela signifie que tous les threads disposent d’une instance unique, et donc d’un temps écoulé qui est global à l’application (et non pas spécifique à un thread).

Ce temps est notamment utilisé lors du log de certaines actions, telles que :

* L’attente d’accès à un document,
* Le début du traitement d’un document,
* Et la fin de ce traitement.
  1. Classe Log

Cette classe représente une action effectuée par une personne, à un temps donné. Elle comprend notamment les informations suivantes :

* La durée de l’action (et non pas le temps écoulé depuis le lancement de l’application)
* La personne ayant effectué l’action, ainsi que le document sur lequel l’action a été effectuée,
* Le type de l’opération :
  + WAITING : Attente d’avoir accès à son document
  + REMOVE : Sortie de la file d’attente, début du traitement du document, verrouillage des locks du document
  + FINISHED : Fin de traitement du document, déverrouillage des locks du document
  1. Classe WaitingLogger
* Chaque personne possède une instance commune de cette classe
* Lors d’une action effectuée par une personne, celle-ci peut logguer son action grâce à cette classe
* Lorsqu’une opération est logguée à l’aide du WaitingLogger, le Log est ajouté dans une file (contenant tous les logs jusqu’à présent). Cette file / liste une blockingqueue, ce qui permet d’éviter les problèmes de concurrence en cas d’accès parallèle de la part de plusieurs threads
* Elle possède toutes les opérations effectuées jusqu’à présent. L’utilisateur peut, en saisissant NEXT dans la console (ou alors juste en appuyant sur la touche ENTER) afficher le log se trouvant en tête de file (donc le plus ancien)
* Une fois le log le plus ancien récupéré, un traitement différent en fonction de son type lui est réservé :
  + WAITING : Ajout du log dans une liste contenant les personnes en attente d’accès à leur document (waitingLists)
  + REMOVE : Suppression du log de la waitingLists, et ajout dans une liste contenant les personnes en cours de traitement (processingList)
  + FINISHED : Suppression du log de la processingList, et ajout dans une liste contenant les personnes ayant terminé de traiter leur document (finishedList)
* Une fois le log traité et les files mises à jour, l’état du programme (lors du log de l’action) est affiché dans la console
  1. Déroulement du programme

Ce schéma résume les chapitres précédents présentant les différentes classes du programme.

1. Résultats

Ci-dessous sont présentés des exemples d’utilisation du framework de monitoring. Ceux-ci comprennent notamment les configurations suivantes :

* 4 documents & 4 personnes, dont 2 producteurs et 2 consommateurs,
  + 4-2-2
* 1 document & 4 personnes, dont 2 producteurs et 2 consommateurs,
  + 1-2-2
* 1 document et 4 personnes, dont 4 consommateurs,
  + 1-0-4
* 1 document et 4 personnes, dont 4 producteurs,
  + 1-4-0
  1. 4-2-2
  2. 1-2-2
  3. 1-0-4
  4. 1-4-0

1. Limitations et perspectives

* Les couleurs ne fonctionnent pas dans une invite de commande Windows (cmd) ou un prompt Powershell,
* L’interface en ligne de commande n’est pas forcément très instinctive ni simple d’utilisation, il pourrait être intéressant de créer une interface graphique afin d’augmenter la facette « user-friendly » du framework. Pour cela, une application développée à l’aide de Swing pourrait être mise en place.
* Étendre le framework à d’autres variantes du pattern lecteurs-rédacteurs, notamment un changement de priorité en fonction du type de personne. Par exemple, donner la priorité aux lecteurs, ou inversement, donner la priorité aux consommateurs.
* Mettre en place des tests unitaires ou tests d’intégration (ou autres types de tests) afin de s’assurer de la qualité et du bon fonctionnement de l’application.
* Ajouter des possibilités de configuration plus poussée. Pour l’instant, l’utilisateur peut définir le nombre de documents à traiter, et le nombre de personne. Cependant, de type des personnes ainsi que l’attribution des documents se fait de manière aléatoire. Il pourrait être intéressant de laisser l’utilisateur définir ces paramètres lui-même, afin de pouvoir tester des cas précis.

1. Conclusion

En fonction de la longueur des premières pages du rapport (avant la table des matières), il se peut que le numéro de la dernière page courante (normalement celle-ci) ne corresponde pas au nombre de pages total.

Il faut alors modifier ce dernier champ (numéro de page total, à droite du « / »).

* Placer la souris sur ce champ, qui sera alors grisé
* Appuyer les touches SHIFT + F9 pour afficher le code du champ
* Modifier le nombre à soustraire au nombre de pages total (à droite du signe « - »)
* Appuyer sur les touches SHIFT + F9 pour rebasculer en mode normal
* Framework fonctionnel
* Possibilité de monitorer l’accès concurrents à des ressources, en se basant sur le pattern lecteurs-rédacteurs simple
* Interface console simple, mais efficace -> Utilisation de couleurs pour simplifier la lecture et la compréhension du diagramme
* Projet intéressant, c’est sympa de pouvoir suivre l’évolution de threads concurrents dans un exemple concret

1. Annexes
   1. Guide utilisateur
   2. Cahier des charges
   3. Table des illustrations
   4. Bibliographies et références
      1. Sites Web
      2. Livres
      3. Autres