Programmation asynchrone en Java

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-01

Table des matières

1 la programmation asynchrone en Java 1
2 Objectifs pédagogiques 1
3 Différences entre Programmation Synchrone et Asynchrone 2
4 Méthodes de Programmation Asynchrone en Java 2
4.1 Les Threads en Java 2
4.2 Les Callbacks en Java 3
4.3 Objets Future et ExecutorService en Java 3
5 CompletableFuture en Java (Promises) 3
6 ForkJoinPool 4
7 Virtual Threads (Java 21) 4
8 Patterns 5
9 Tests Unitaires avec JUnit 5 5
10 Problèmes de la Programmation Asynchrone 6
11 Solutions et Bonnes Pratiques 7
12 Ressources 7

1 la programmation asynchrone en Java

- Qu'est-ce que la Programmation Asynchrone ?
  - **Programmation Asynchrone** : Exécution de tâches indépendantes sans bloquer le flux principal.
  - **Objectif** : Améliorer la réactivité et l'efficacité des applications.
- Importance
  - **Réactivité** : Répondre rapidement aux événements utilisateurs.
  - **Efficacité** : Utilisation optimale des ressources système.
  - **Scalabilité** : Gérer un grand nombre de tâches simultanément.

2 Objectifs pédagogiques

À la fin de ce cours, les étudiants seront capables de :

- Comprendre les concepts fondamentaux de la programmation asynchrone
- Maîtriser les différentes APIs asynchrones de Java
- Implémenter des solutions asynchrones performantes
- Tester du code asynchrone
- Choisir la meilleure approche selon le contexte

3 Différences entre Programmation Synchrone et Asynchrone

Programmation Synchrone

- Les tâches sont exécutées séquentiellement, une à la fois.
- **Avantages** :
  - Simplicité de compréhension et de débogage.
- **Inconvénients** :
  - Blocage du flux principal.
  - Moins efficace pour les opérations I/O.

Programmation Asynchrone

- Les tâches sont exécutées de manière non bloquante, permettant à d'autres tâches de s'exécuter simultanément.
- **Avantages :**
  - Meilleure réactivité.
  - Utilisation efficace des ressources.
- **Inconvénients :**
  - Complexité accrue.
  - Gestion des erreurs plus difficile.

4 Méthodes de Programmation Asynchrone en Java

- **Threads** : Utilisation de threads pour exécuter des tâches en parallèle.
- **Callbacks** : Utilisation de callbacks pour gérer les opérations asynchrones.
- **Futures** : Utilisation de futures pour récupérer le résultat d'une opération asynchrone.
- **CompletableFuture** : Utilisation de CompletableFuture pour chaîner des opérations asynchrones.

4.1 Les Threads en Java

- **Thread** : Unité d'exécution légère permettant l'exécution parallèle.
- **Runnable** : Interface fonctionnelle représentant une tâche pouvant être exécutée par un thread.
- **États d'un Thread** :

- NEW: créé mais pas démarré
- RUNNABLE: en cours d'exécution
- BLOCKED: bloqué (verrou)
- WAITING: en attente
- TIMED_WAITING: attente temporisée
- TERMINATED: terminé

- **Opérations sur un Thread** :
  - start(): démarre le thread
  - run(): code à exécuter
  - sleep(): pause
  - join(): attend la fin
  - interrupt(): interrompt
  - yield(): cède le processeur

4.2 Les Callbacks en Java

- **Callback** : Méthode passée en argument à une autre méthode pour exécuter du code après une opération asynchrone.
- Utilisé pour la programmation asynchrone et événementielle.

4.3 Objets Future et ExecutorService en Java

- `Future` est une interface de Java pour représenter le résultat d'une opération asynchrone
- Permet de vérifier si une tâche est terminée et de récupérer son résultat
- `ExecutorService`
  - Création et Gestion des Pools de Threads
  - FixedThreadPool : Pool avec un nombre fixe de threads.
  - CachedThreadPool : Pool avec un nombre variable de threads, créant de nouveaux threads si nécessaire.
  - SingleThreadExecutor : Pool avec un seul thread.

5 CompletableFuture en Java (Promises)

- `CompletableFuture` est une classe de Java 8 pour la programmation asynchrone
- Permet de chaîner des opérations asynchrones
- Supporte la gestion des erreurs et des
- Méthodes Principales :
  - **thenApply** : Transformation du résultat.
  - **thenCompose** : Chaînage de futures.
  - **thenCombine** : Combinaison de deux futures.

- **thenAccept** : Consommation du résultat.
- **thenRun** : Exécution d'une tâche après la complétion.
- **exceptionally** : Gestion des erreurs.
- **get** : Récupération bloquante du résultat.
- **get(timeout, unit)** : Récupération avec timeout.

---

6 ForkJoinPool

- Implémentation avancée de `ExecutorService` pour les tâches de type « divide-and-conquer ».
- Améliorer l'efficacité des tâches parallèles en utilisant le « work-stealing ».
- Fonctionnement
  - **Work-Stealing** : Technique où les threads inactifs volent des tâches des threads occupés pour équilibrer la charge de travail.
  - **Divide-and-Conquer** : Diviser une tâche complexe en sous-tâches plus petites, les résoudre en parallèle, puis combiner les résultats.
- Nombre de Threads Créés
  - **Nombre de Threads** : Le nombre de threads dans un `ForkJoinPool` est déterminé par le nombre de processeurs disponibles.
  - **Par défaut** : `ForkJoinPool` utilise `Runtime.getRuntime().availableProcessors()` pour déterminer le nombre de threads.
  - **Personnalisation** : Vous pouvez spécifier le nombre de threads lors de la création d'un `ForkJoinPool`.
- Classes Principales
  - **RecursiveTask** : Classe abstraite pour les tâches qui retournent un résultat.
  - **RecursiveAction** : Classe abstraite pour les tâches qui ne retournent pas de résultat.

---

7 Virtual Threads (Java 21)

- Threads légers gérés par la JVM (et non par le système d'exploitation)
- Coût mémoire très faible (~1KB vs ~1MB pour un thread platform)
- Permet de créer des millions de threads virtuels
- Optimisé pour les opérations I/O bloquantes
- Comparaison avec les Platform Threads

Caractéristique	Platform Thread	Virtual Thread
Gestion	OS	JVM
Coût mémoire	~1MB	~1KB
Nombre max	Milliers	Millions
Performance I/O	Bloquant	Non-bloquant
Stack size	Fixe	Dynamique

8 Patterns

- Patterns de Communication
  - **Observer Pattern**
    - Notification automatique des changements d'état
    - Découplage Publisher/Subscriber
    - Idéal pour les événements locaux
  - **Pub/Sub Pattern**
    - Version distribuée de l'Observer
    - Communication via message broker
    - Découplage spatial et temporel
- Patterns de Résilience
  - **Circuit Breaker**
    - Protection contre les défaillances en cascade
    - États: Fermé → Ouvert → Semi-Ouvert
    - Seuils et timeouts configurables
  - **Bulkhead**
    - Isolation des ressources système
    - Partitionnement des pools de threads
    - Prévention de la propagation des pannes
  - **Retry Pattern**
    - Gestion automatique des erreurs transitoires
    - Stratégies: délai fixe, exponentiel, jitter
    - Paramètres: max retries, delay, timeout

9 Tests Unitaires avec JUnit 5

- **Tests Asynchrones**
  - `@Timeout` pour les tests longs
  - `assertTimeout()` pour vérifier la durée
  - Tests parallèles avec `@Execution(CONCURRENT)`

    - Utilisation de `CompletableFuture.get()` avec timeout
- **Outils de Profiling**
- **VisualVM**
    - Outil visuel pour monitoring JVM en temps réel
    - Analyse CPU, mémoire, threads et GC
    - Profiling des applications asynchrones
    - Support des thread dumps et heap dumps
    - Documentation VisualVM
- **JFR (Java Flight Recorder)**
    - Outil de diagnostic à faible impact
    - Collecte de métriques détaillées :
        - Latence des threads
        - Allocation mémoire
        - Garbage collection
        - I/O et réseau
    - Analyse post-mortem avec JMC (Java Mission Control)
    - Guide JFR Oracle

10 Problèmes de la Programmation Asynchrone

- **Complexité**
    - **Gestion manuelle des callbacks et des threads** : La gestion des callbacks et des threads peut rapidement devenir complexe et difficile à maintenir.
    - **Callback Hell** : L'imbrication excessive de callbacks peut rendre le code illisible et difficile à déboguer.
    - **Synchronisation** : La synchronisation des threads pour éviter les conditions de course et les deadlocks ajoute une couche de complexité supplémentaire.
- **Difficulté de Composition**
    - **Combinaison d'opérations asynchrones** : Il est souvent difficile de combiner et d'enchaîner des opérations asynchrones de manière fluide.
    - **Coordination des tâches** : La coordination de plusieurs tâches asynchrones peut être complexe, surtout lorsqu'elles dépendent les unes des autres.
    - **Propagation des résultats** : La propagation des résultats entre différentes étapes asynchrones peut être difficile à gérer.
- **Gestion des Erreurs**
    - **Détection des erreurs** : La détection des erreurs dans un contexte asynchrone est plus complexe que dans un contexte synchrone.
    - **Propagation des exceptions** : La propagation des exceptions à travers les différentes couches de callbacks ou de threads peut être difficile à implémenter correctement.

- **Gestion centralisée** : Il est difficile de centraliser la gestion des erreurs pour les opérations asynchrones, ce qui peut entraîner une duplication du code de gestion des erreurs.

11 Solutions et Bonnes Pratiques

- **Utilisation de Frameworks et Bibliothèques**
  - **CompletableFuture** : Simplifie la gestion des opérations asynchrones et permet de chaîner les opérations de manière fluide.
  - **Project Reactor** : Fournit des abstractions pour la programmation réactive, facilitant la composition et la gestion des flux de données asynchrones.
  - **RxJava** : Offre des outils pour la programmation réactive, permettant de gérer les opérations asynchrones de manière déclarative.
- **Gestion des Callbacks**
  - **Utilisation de Lambdas** : Simplifie la syntaxe des callbacks et améliore la lisibilité du code.
  - **Éviter le Callback Hell** : Utiliser des techniques comme les Promises ou les Futures pour éviter l'imbrication excessive des callbacks.
- **Gestion des Erreurs**
  - **Centralisation de la Gestion des Erreurs** : Utiliser des mécanismes centralisés pour gérer les erreurs, comme les handlers globaux ou les mécanismes de propagation des exceptions.
  - **Utilisation de Méthodes de Gestion des Erreurs** : Utiliser des méthodes comme `exceptionally` ou `handle` avec `CompletableFuture` pour gérer les erreurs de manière fluide.
- **Composition des Opérations Asynchrones**
  - **Chaînage des Opérations** : Utiliser des méthodes comme `thenApply`, `thenCompose`, et `thenCombine` pour chaîner les opérations asynchrones.
  - **Coordination des Tâches** : Utiliser des outils comme les `CompletableFuture` ou les flux réactifs pour coordonner les tâches asynchrones de manière efficace.

12 Ressources

- Java Concurrency in Practice - Livre de référence
- Java API Documentation - CompletableFuture
- Project Loom Documentation
- JEP 444: Virtual Threads