Les threads en Java

Licence DANT

PITTON Olivier

Les threads

Qu’est-ce qu’un thread ?

Les threads

Qu’est-ce qu’un thread ?

Un thread est une unité d’exécution d’un programme.

Autonome et parallèle

Les threads

2 types de threads :

- Les threads natifs

- Les threads de la VM

Les threads

La JVM crée ses propres threads :

- Le main thread pour l'exécution du programme

- Les threads pour le GC

Les threads

Est-ce que l’utilisation des threads améliorent les performances ?

Thread et Runnable

Thread et Runnable

Il existe 2 manières de faire executer du code en parallèle avec la classe *Thread*.

Créer un *Runnable* et le soumettre à un *Thread*

Créer une classe héritant de *Thread*, et redéfinir la méthode run()

Runnable

L'interface *Runnable* doit être implémentée par toute classe avec des traitements à effectuer.

Une méthode à redéfinir : void run()

Lors de la creation d'un thread, celui-ci utilisera notre *Runnable*

Runnable

```
public class MonTraitement implements Runnable {
 public void run() {
  int i = 0;
  for (i = 0; i > 10; i++) {
   System.out.println("" + i);
  }
 }
}
```

Thread

La classe *Thread* répresente un thread VM.

Elle implémente l'interface *Runnable*

Possède des méta données : priorité, nom, daemon …

Thread – les méthodes

void start() démarre le thread en ‘parallèle’

void run() exécute le traitement dans le thread courant

void join() attend la fin de l’execution

void sleep(long millis) endort le thread pendant X ms

Thread – cycle de vie

NEW Pas encore démarré. En attente de start()

RUNNABLE En cours d’execution

BLOCKED En attente d’un verrou / moniteur

WAITING En attente d’un réveil (sleep, wait …)

TERMINATED Fin d’exécution

Thread – création

En live coding io.dant.thread.creation

Thread - daemon

Il existe deux categories de threads :

- User thread

- Daemon thread

Le programme s’arrête lorsque tous les user threads sont terminés

Thread - daemon

Lors le dernier user thread s’arrête, tous les daemon threads sont tués.

Appeler void setDaemon(Boolean) pour créer un daemon thread.

Thread - daemon

En live coding io.dant.thread.daemon

Thread - metadata

long getId() □ Renvoie l'identifiant unique du thread

Thread.State getState() □ Renvoie le status du thread

int getPriority() □ Renvoie la priorité du thread

boolean isDaemon() □ Renvoie vrai si le thread est un daemon

Thread - metadata

En live coding io.dant.thread.metadata

Thread - join

La méthode join permet d’attendre la fin d’exécution du thread

En live coding io.dant.thread.join

Restrictions

- On ne peut pas relancer un thread terminé
- On ne peut pas créer une infinite de threads
- On ne peut pas cloner un thread

Stack / Heap

La stack

- Chaque thread a une pile (stack). Un espace mémoire dédié à chaque thread.

- La stack garde une trace des invocations successives de méthodes

La stack

La stack stocke :

- Les variables locales primitives et références
- Chaque invocation d'une méthode ajoute une entrée nommée frame en haut de la stack

Heap

Le tas (heap) est un espace mémoire où les objets sont alloués.

Faire un ‘new’ crée un objet dans la heap

Lorsque les références sur cet objet disparaisse, il est nettoyé par le GC

Heap

La heap est partagée par tous les threads.

La stack est propre à chaque thread

Les threads peuvent donc partager des objets

La stack

Lorsque l'exécution de la méthode est terminée, la frame est retirée de la stack. Les variables qu'elle contient sont supprimées

- si ces variables sont des objets, leurs références sont supprimées mais ils existent toujours dans le heap.
- Si aucune autre référence sur ces objets existe, le ramasse-miettes les détruira à sa prochaine exécution.

Heap / Stack

En live coding io.dant.thread.heapstack

Heap / Stack - Exception

StackOverFlowError Dépassement de la taille de la pile

OutOfMemoryError Le système ne peut plus allouer de mémoire

Le framework Executor

Le framework Executor

La classe *Thread* est bas niveau et difficile à gérer.

A utiliser dans des cas précis(un timer, …)

Executor permet la creation de pool de threads

L’interface Executor

L’interface Executor décrit les fonctionnalités permettant l'exécution différée de tâches implémentées sous la forme de Runnable.

void execute(Runnable) Fait exécuter la tâche fournie

L’interface ExecutorService

L'interface ExecutorService décrit les fonctionnalités d'un service d'exécution de tâches. Elle hérite de l'interface Executor.

Future submit(Runnable) Fait exécuter la tâche fournie

Future submit(Callable) Fait exécuter la tâche fournie avec un retour

L’interface Callable<T>

L'interface Callable<T> est similaire à Runnable, mais renvoie un résultat et peut lancer une exception

T call() □ Exécute une tâche et renvoie un résultat

L’interface Future

L'interface Future permet d’attendre le résultat d’un processus asynchrone.

void get() Bloque jusqu’à ce que la tâche soit terminée

boolean isDone() Renvoie vrai si la tâche est finie

boolean isCancelled() Renvoie vraie si la tâche a été annulée

L’interface ExecutorService

Toujours penser à fermer les ressources en appellant shutdown()

Entièrement configurable

Peuvent être créés via Executors.

L’interface ExecutorService

En live coding io.dant.thread.pool

Dans l’ordre : HelloWorldPool / CallablePool / ManyThreadPool / BlockedThreadPool

L’interface CompletionService

Nous devons attendre le résultat de tout le batch soumis avant d’avoir les résultats.

L’interface CompletionService permet de récupérer les résultats dès qu’ils sont disponibles

L’interface CompletionService

Future poll() → Obtenir l'instance de type Future de la prochaine tâche qui se terminera. Null si aucune n’est terminée

Future submit(Callable) → Demander l'exécution de la tâche fournie en paramètre

Future poll() → Obtenir l'instance de type Future de la prochaine tâche qui se terminera. Bloquante jusqu’à ce qu’une tâche soit terminées

L’interface CompletionService

En live coding io.dant.thread.pool.CompletionPool

Plus besoin de gérer les Future !