CRAC vs GraalVM, pour un démarrage plus rapide

Date : 13 avril 2023
Speaker : Lilian BENOIT
Format : Conférence (45mn)
Blog : https://www.lilian-benoit.fr/

Introduction

Les 2 projets CRaC et GraalVM poursuivent le même objectif : faire démarrer plus vite une application.
A l'issue de la présentation, Lilian mettra à disposition 2 repos GitHub avec le code source de la présentation. Surveiller https://github.com/lilian-benoit/

Le temps de la JVM est connu pour être lent. En dév comme en prod, on ne démarre pas son application toutes les minutes. Ce n'était jusqu'alors pas si problématique.
L'essor des microservices rabat les cartes : l'élasticité du redimensionnement des pods Kubernetes fait que les applis doivent rapidement démarrer. Lilian note une diminution de l'usage des serveurs d'applications. Le temps de démarrage compte également dans les architectures Serverless où l'on paie au temps d'usage du CPU et où l'on veut que la fonction réponde rapidement.

Fonctionnement de la JVM

Lilian rappelle que le code source Java est compilé en bytecode qui sera interprété au runtime par la JVM
La JVM permet de préchauffer le code. Les compilateurs C1 et C2 du JIT permettent de compiler du code en natif. On passe progressivement du tout interprété à du code. C'est ce qu'on appelle la préchauffe. Ce préchauffage est assuré par un système de profilage et d'optimisations de la JVM.

Usage de Java

- Java est un langage dynamique : on peut charger les classes de manière dynamique depuis les JAR.
- Les développeurs Java sont friands d'annotations (ex : Entity JPA). L'analyse de ces annotations au démarrage de la JVM demande du temps
- L'initialisation des blocs / constructeurs statiques
- Initialisation du contexte applicatif comme avec CDI (Weld) ou Spring

GraalVM

Projet venant d'Oracle Labs disposant d'un compilateur Graal proposant la compilation vers un exécutable natif. Le compilateur AOT compile le code en avance de phase, au moment de

la phase de build de l'application. GraalVM nécessite de connaitre l'ensemble des classes qui sont nécessaires au runtime.
Démo avec une application Quarkus et un endpoint JAX-RS, d'abord en Java puis en construisant l'image native.
Compilation très lente en 5mn48 basée sur Java 17. Utilisation du compilateur gcc.
Statistiques sur le nombre de classes et de champs utilisés pour éliminer le code mort.

Les contraintes du natif

GraalVM génère un binaire linux. Ce n'est plus la JVM qui exécute le code, mais Substrate VM qu'on connait moins. Le garbage collector utilisé par Substrate est le Serial GC, l'un des GC les moins performant. Pour une fonction lambda Servless, cela n'a pas d'importance. Pour un autre type d'application, çà l'est moins.
La réflexion et le dynamisme : fichier de conf nécessaire pour que Graal native image sache comment est utilisée la réflexion.
La sérialisation Java ne fonctionne pas.
Les optimisations à chaud du code et le GC G1 faible latence ne sont disponibles que dans la version Enterprise de GraalVM.
GraalVM est sorti il y'a 5 ans. CRAC est plus récent.
Debugger C pour debugger l'image native.

CRaC Coordinated Restore at Checkpoint

CRaC est un projet en incubation faisant partie de l'OpenJDK.
Il repose sur 2 mécanismes : **checkpoint** et **restore**.
La JVM est très performante et s'améliore à chaque version de Java.
Une fois que la JVM est chaude, on **fait une image** pour **stocker l'état de la JVM** et **toutes les optimisations**. Le restore permet de restaurer l'état de la JVM au moment du checkpoint.

Démo 1

Lilian poursuit avec une démo utilisant une version spéciale de java 17 embarquant CRaC.
On indique au démarrage de la JVM où positionner le fichier :
java -XX :CRaCCheckpointTo=/tmp/crac -jar mon-archive.jar

Commande pour effecteur le checkpoint :
jcmd mon-archive.jar JDK.checkpoint

Pour restaurer le checkpoint :
java -XX :CRaCCheckpointFrom=/tmp/crac -jar

CRaC vient avec une API Java (**package javax.crac**) permettant d'afficher des infos pendant la démo.
Démo du code mettant en cache les nombres premiers au démarrage de l'application puis se mettant en pose.

Besoin d'enregistrer la classe au niveau de CrAC :

Core.getGlobalContext().register(this);

L'exécution de la demande de checkpoint interrompt la JVM. Les fichiers de dump sont générés.

Démo 2

Seconde démo avec une appli qui démarre un serveur Jetty.
Contraintes : l'état des connexions bases de données et des requêtes HTTP ne doit pas être conservé. Au niveau de la méthode beforeCheckpoint(), on demande à Jetty de s'arrêter. On fait redémarrer Jetty après la restauration.

Démo 3

3ième démo basée sur Spring Boot
@SprinBootApplication avec main et un contrôleur REST HelloController.
Rien de spécial à ajouter dans le code.
Dans le pom.xml on exclue le tomcat embbeded tiré par Spring Boot et on ajoute la dépendance Maven du tomcat spécialement buildée pour CRaC :
io.github.crac.org.apache.tomcat.embed/tomcat-embed-core