Loom nous protègera-t-il du Braquage Temporel ?

Date : 20 avril 2022
Speakers : José Paumard (Oracle) et Rémi Forax (Professeur à l'université)
Format : université de 3h

Présentation

Le projet **Loom** est en chantier depuis 4 ou 5 ans.
Disclaimer : Loom n'est intégré au JDK 19 qu'en mode preview et son fonctionnement / API peut changer d'ici sa sortie.
Rémi a travaillé sur plusieurs spécifications du JDK : Invoke Dynamic, Lambda, Module, Record, Valhalla, Constant dynamic.
José fait la promotion du site https://dev.java/ sorti à l'occasion de Java 17.
Java 18 est sorti récemment et les premières bétas de Java 19 sont en cours.

Historique

Loom est un nouveau projet de programmation concurrente. La plateforme Java compte une évolution de ce genre tous les 10 ans :

1. En 1995, on avait les Thead, les Runnable les join et les blocs synchronized.
2. En 2004, avec Java 5, sort le package java.util.concurrent avec les Callable, le modèle ExecutorService et les Future, les Lock / ReentrantLock, des structures de données concurrentes ConcurrentMap/BlockingQueue …
3. En 2011 (Java 7) et 2014 (Java 8), on a les fork / join puis les parallel Stream, les CompetionStage/CompletableFuture

Un thread permet d'aller regarder dans une autre file s'il y'a du travail.
La relation entre le thread et la tâche est forte.
Loom revisite cette idée : un thread peut être libéré de sa tâche de façon temporaire et peut y revenir plus tard.

Lorsqu'on fait du calcul, disposer d'autant de threads que de CPU est suffisant.
Par contre, lorsqu'on fait des opérations en lecture/écriture (IO), disposer de plus de threads que de nombre de CPU est intéressant car les threads ne font pas grand-chose.
Un thread envoie sa requête sur le réseau puis attend le retour (échelle de temps de 10ms).
L'occupation CPU est minime. Le thread dort 99.9999% du temps (idle).

En Java, l'utilisation d'un thread a un coût :

- 1ms de démarrage
- 2048 Ko de stack (à vide : taille de la pile)
- 100 microsecondes pour switcher de threads

1 millions de threads demanderait 2mn de démarrage et requièrerait 2 To de RAM.
La limitation du nombre de threads vient donc de la taille de la stack.

Démo d'un code Java créant 100 000 threads + boucle pour attendre la fin des threads.
Au bout de 4067 threads lancés, un *OutOfMemoryError* survient. Sur un serveur on peut monter sur 10 000 (en fonction de la RAM).
Guardsize : taille max d'une fonction de 16Ko

Loom va apporter des threads virtuels. Ces threads auront une pile mais ne sont pas associées aux threads POSIX du système.

Pour démarrer un thread virtuel :

```
1.  var thread : Thread = Thread.startVirtualThread(() -> {
2.      System.out.println(Thread.currentThread());
3.  });
4.  thread.join() ;  // on attend que le thread virtuel démarre
```

Sortie : VirtuelThread[#15]/runnable@ForkJoinPool-1-worker-1

Attention : ce n'est pas le ForkJoinPool du common fork join pool utilisé pour les threads parallèles.

Remarque : l'utilisation d'une méthode statique masque son implémentation. Rémi fait remarquer que cet usage est de plus en plus répandu dans le JDK.
Le type reste inchangé : *java.lang.Thread* : partout où l'on avait des vrais threads, on peut désormais utiliser des threads virtuels.
En Java 19, pour accéder aux fonctionnalités en preview, il faut les activer explicitement via l'option de JVM : **--enable-preview**
On remarque que la Thread créée est la 15^{ième}. La JVM en a déjà créée d'autres au démarrage.
Les threads virtuels utilisent les threads de l'OS. Le thread virtuel peut se balader entre plusieurs threads réels. Ce modèle nécessite de continuer à gérer la concurrence d'accès aux données.
Il s'agit d'un modèle différent de NodeJS ou de Vert.x.

Autre façon de créer un thread :

```
1. var thread = Thread.ofVirtual() versus Thread.ofPlatform()
2.   .unstarted(() -> system out}).
3. thread.start();
4. thread.join();
```

Rémi reprend le 1er exemple créant 100 000 threads avec des **Thread.ofVirtual().** L'exemple est désormais passant.
Créer un thread virtuel reste 1000x plus lent que l'instanciation d'un objet.

Running a Thread

Platform/OS thread (starts in ms)
- Creates a 2MB stack upfront
- System call to ask the OS to schedule the thread

Virtual thread (starts in µs)
- Grow and shrink the stack dynamically
- Use a specific fork-join pool of platform threads (carrier threads)
- One platform thread per core

Sous le capot, les threads virtuels utilisent la classe ***jdk.internal.vm.Continuation*** :

```
public interface _5_continuation {
  static void main(String[] args)  {
    var scope = new ContinuationScope( name: "hello");
    var continuation = new Continuation(scope, () -> {
      System.out.println("C1");
      Continuation.yield(scope);
      System.out.println("C2");
      Continuation.yield(scope);
      System.out.println("C3");
    });

    System.out.println("start");
    continuation.run();
    System.out.println("came back");
    continuation.run();
    System.out.println("back again");
    continuation.run();
    System.out.println("back again again");
```

La méthode ***Continuation.yield()*** vient du Python. Elle permet de sortir d'une méthode tout en la laissant sur la pile pour y revenir plus tard.
Tout le code de Loom repose sur la classe *Continuation*.
Pour freezer la pile, on peut faire une capture et la copier sur le tas. Le *run* d'une Continuation reprend ce qu'il y'a sur le tas et le remet sur la pile. Les variables locales sont sauvegardées. En Java on peut copier des morceaux de pile pour le remettre ailleurs. En C et en C# on ne peut pas car on a des adresses mémoires, le fameux opérateur & (adresse de la pile).
Lorsque le code Java appel du code C on ne peut pas utiliser des virtuels threads.
Dans le JDK il y'avait du code C pour implémenter :

- les anciennes sockets (avant NIO)
- la réflexion
- le mot clé *synchronized*

L'intérêt des virtuels threads est que l'on peut changer le modèle et utiliser un thread virtuel par requête HTTP. Un thread virtuel ne coûte pas cher : il n'y a donc pas besoin de gérer un pool.

Alternative aux threads virtuels :
- *Mono*/*Flux* de Spring
- *Uni*/*Multi* de Quarkus
- *Async*/*await* de C# et Kotlin
- *CompletableFuture* du JDK.

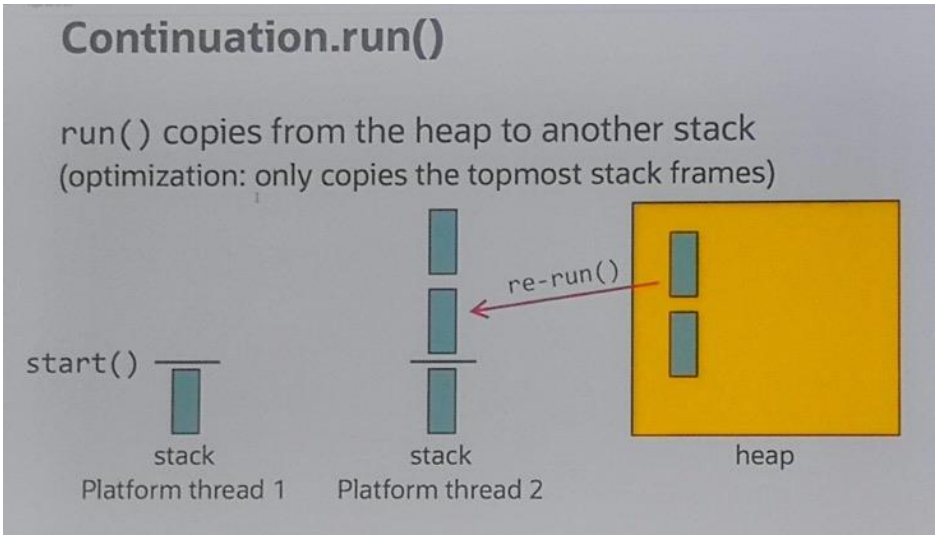
Une problématique majeure de la programmation asynchrone ou reactive est le debuggage. La stacktrace contient toute la machinerie de la stack réactive.
A ce jour, aucun profiler n'arrive à profiler ce type de code.

L'utilisation d'**async** pose un problème dit des fonctions colorées : on doit en mettre partout (référence à la diarrhée).

Techniquement, la notion de virtuel thread existe en Erlang (process) et en Go (goroutine)

Pour la concurrence, on a 2 stratégies : compétitive ou coopérative.
Loom utilise les 2 : compétition entre threads de l'OS et coopération entre threads virtuels.

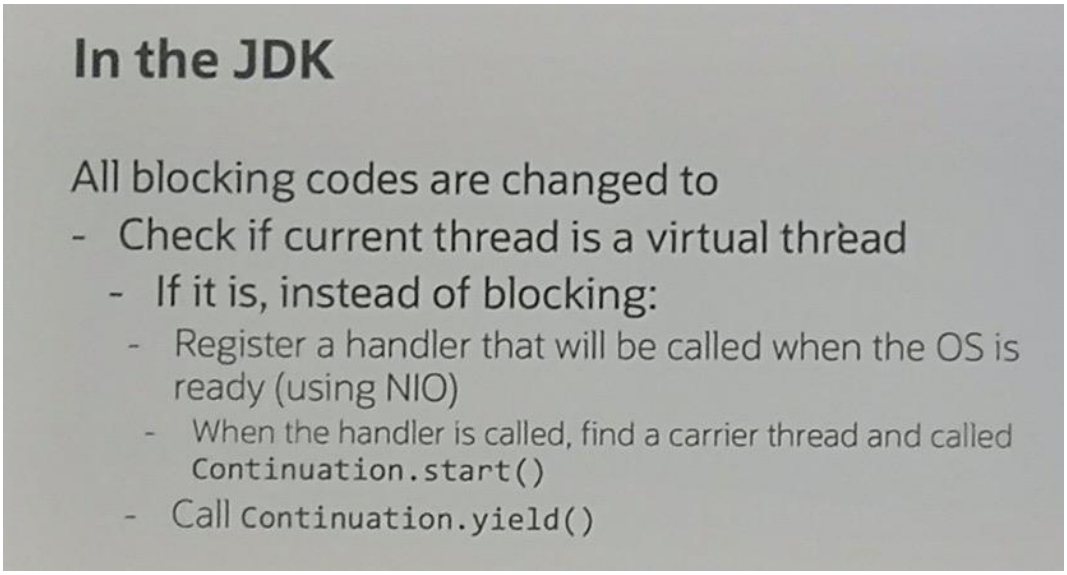
Lors de l'appel à *Thread.sleep()*, s'il s'agit d'une thread virtuel, la JVM fait appel à un *yieldContinuation()* après avoir parqué le thread virtuel. Cette méthode renvoie un booléen permettant de savoir si le yield a réussi (permet de gérer le cas d'appel à du C).



Toute la pile n'est pas recopiée (optimisations faites par le JVM).

Loom n'est pas implémenté par la JVM. Il est écrit en Java (et un peu de C) dans le JDK.

Dans la JVM, tous les appels avec du code bloquant a été changé pour vérifier si le thread en cours était virtuel ou de la plateforme. S'il est virtuel, on enregistre un handler pour continuer le traitement.



Lorsque sur la pile on a du code qui référence du code de la pile.
Depuis Java 13, le JDK a été réécrit en Java pour ne plus avoir ce type de code :

- JEP-353 Legacy Socket API
- JEP-373 Reimplement the Legacy Datagram Socket API
- JEP-374 Deprecate and Disable Biased Locking
- JEP-416 Reimplement Core Reflection with Method Handles

Problèmes restants à traiter :

- Blocs synchronizés lents : utiliser du ReentrantLock. Le JDK a été réécrit mais pas encore de nombreux frameworks comme Hadoop ou Spark.

Les virtuels threads peuvent être utilisés mais on ne va avoir aucun gain.

Structured Concurrency

José prend pour exemple une agence de voyage avec une page présentant le voyage incluant sa Quotation et son Weather Forevast.
Options : --add-modules jdk.inculabor.concurrent --enable-preview

La classe StructuredTaskScope est proche d'un ExecutorService :

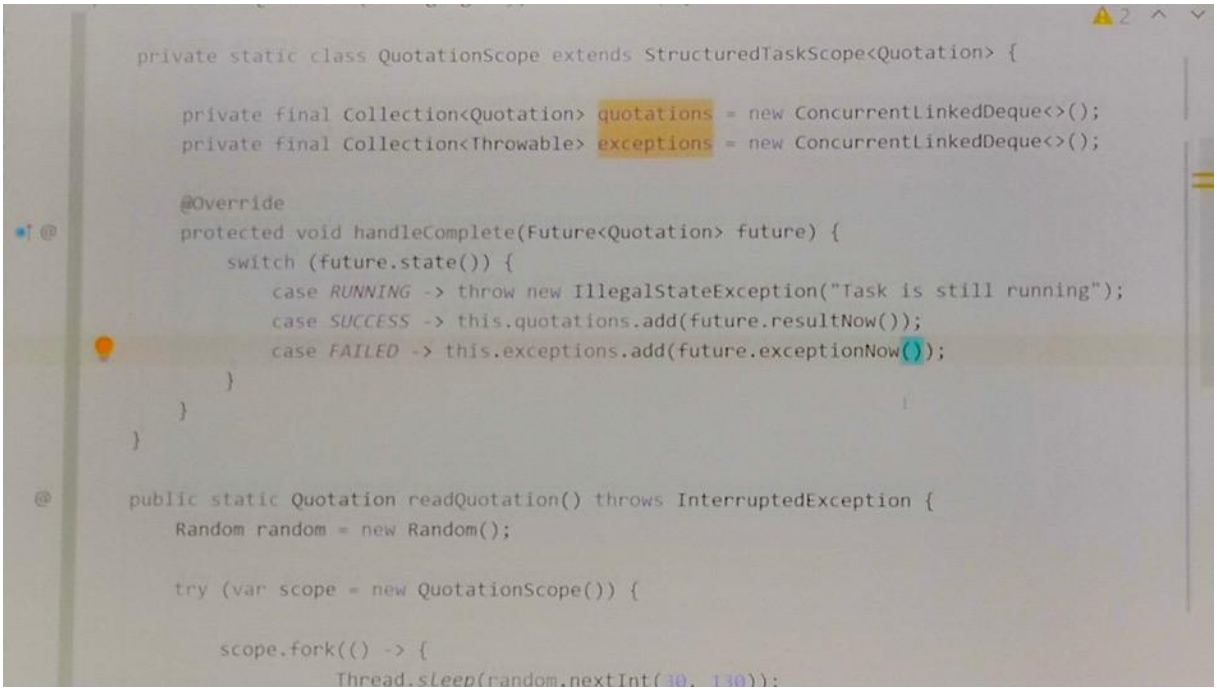
```
1.  try (var scope = new StructuredTaskScope<Weather>()) {
2.        Future<Weather> futureA = scope.fork(() -> new Weather("WA", "Sunny"));
3.        scope.join();
4.  }
```

StructuredTaskScope.ShutdownOnSuccess : dès qu'une tâche répond, interrompt toute les autres et retourne le résultat :
Weather weather = scope.result() ;

La méthode ***joinUntil*** permet de mettre une deadline.
scope.joinUntil(Instant.now().plusMillis(100)) ;

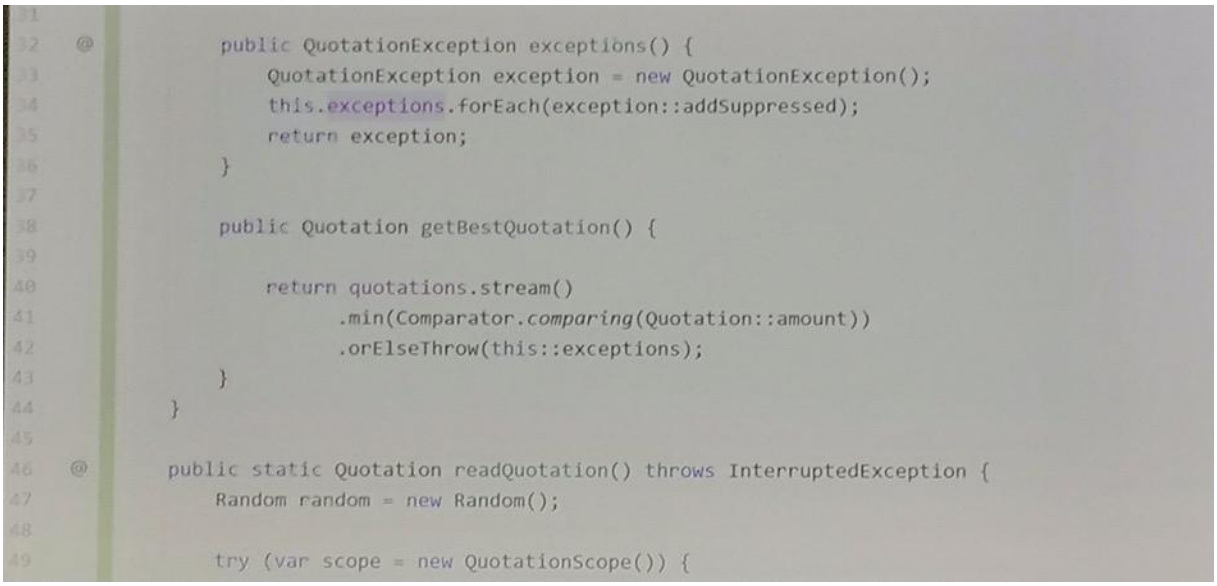
Contrairement à un *ExecutorService* dont le cycle de vie est généralement calé sur celui de l'application, un scope se crée quand on le souhaite ; on l'utilise puis on l'oublie.

José crée une classe *QuotationScope* qui redéfinit la méthode *handleComplete* appelée à chaque fois que la *Future* a terminé sa tâche.



```
    private static class QuotationScope extends StructuredTaskScope<Quotation> {

        private final Collection<Quotation> quotations = new ConcurrentLinkedDeque<>();
        private final Collection<Throwable> exceptions = new ConcurrentLinkedDeque<>();

        @Override
        protected void handleComplete(Future<Quotation> future) {
            switch (future.state()) {
                case RUNNING -> throw new IllegalStateException("Task is still running");
                case SUCCESS -> this.quotations.add(future.resultNow());
                case FAILED -> this.exceptions.add(future.exceptionNow());
            }
        }
    }

    public static Quotation readQuotation() throws InterruptedException {
        Random random = new Random();

        try (var scope = new QuotationScope()) {

            scope.fork(() -> {
                    Thread.sleep(random.nextInt(30, 130));
```

Rémi rappelle que l'utilisation du mot clé *final* sur *quotations* (ou du mot clé volatile) est nécessaire sur architecture ARM (car lu et écrit par plusieurs threads).



```
        public QuotationException exceptions() {
            QuotationException exception = new QuotationException();
            this.exceptions.forEach(exception::addSuppressed);
            return exception;
        }

        public Quotation getBestQuotation() {

            return quotations.stream()
                    .min(Comparator.comparing(Quotation::amount))
                    .orElseThrow(this::exceptions);
        }
    }

    public static Quotation readQuotation() throws InterruptedException {
        Random random = new Random();

        try (var scope = new QuotationScope()) {
```

A noter ci-dessus l'utilisation de la méthode *Throwable*::*addSuppressed*

Lorsqu'un *TimeOutException* survient on peut faire appel à un ***shutdown()*** du scope.

L'utilisation de *ThreadLocal* (table de hachage dans chaque thread) est remplacée par le ***ScopeLocal***. L'implémentation du *where()* fait que le JIT peut optimiser son usage.

Exemple :

```
1.  ScopeLocal.where(API_KEY, "KEY_A", task);
```