STRUCTURED CONCURRENCY WITH PROJECT LOOM

SARIKA SINHA
IBM
ECLIPSE PLATFORM & JDT CO-LEAD
AGENDA

What is Project Loom all about?

Difference between Threads and Virtual Threads

How to use Virtual threads to implement concurrency on a finer-grained level than threads

Using Deadlines to stop the running tasks

Debugger Support for virtual threads
PROJECT LOOM

• High-throughput lightweight concurrency achieved by simpler constructs
  • Delimited Continuations
  • Virtual Threads
  • Tail-call elimination
A recursive function is tail recursive when a recursive call is the last thing executed by the function.

Tail-call elimination is a compile-level optimization that is aimed to avoid stack overflow when calling a recursive method.

Tail-call elimination support in Java is still a future work.
A Continuation (similar to coroutine) is a stack of function calls.

It can be stopped and stored in the heap at some point (with yield) and restarted afterward (with run).

A continuation has no scheduler, so you have to write your own.
public static void main(String[] args) {
    var scope = new ContinuationScope("example5");
    var schedulable = new ArrayDeque<Continuation>();

    IntStream.range(0, 2).forEach(id -> {
        var continuation = new Continuation(scope, () -> {
            for(int i = 0; i < 2; i++) {
                System.out.println("id" + id + " " + i);

                schedulable.add(Continuation.getCurrentContinuation(scope));
                Continuation.yield(scope);
            }
        });
        schedulable.add(continuation);
    });

    while(!schedulable.isEmpty()) {
        schedulable.poll().run();
    }
}
Virtual threads (Fibre) are just threads that are scheduled by the Java virtual machine rather than the operating system.

- Normal Priority
- Daemon Threads
- No permissions with Security Manager
- Inactive threads in a thread group
- No support from Thread suspend, resume and stop APIs.
Main task splits into several concurrent tasks

Spawning threads must terminate before the main thread

Executors API to create an ExecutorService that starts a new virtual thread for each task.
  • ExecutorServices API Executors.newVirtualThreadExecutor()

ExecutorService defines submit methods to execute tasks for execution. The submit methods don't block, instead they return a Future object that can be used to wait for the result or exception.

No impact on ForkJoinPool APIs
public static void main(String[] args) throws Exception {
    Thread thread = Thread.ofVirtual().start(() -> System.out.println("Hello"));
    thread.join();

    var queue = new SynchronousQueue<String>();

    int maxThreads = 18;
    for (int i = 0; i < maxThreads; i++) {
        int number = i;
        Thread thread2 = Thread.ofVirtual().start(new Task(number, queue));
        thread2.setName("Virtual thread # " + i);
    }

    for (int i = 0; i < maxThreads; i++) {
        String msg = queue.take();
        System.out.println(msg);
    }

    try (ExecutorService executor = Executors.newVirtualThreadExecutor()) {
        // Submits a value-returning task and waits for the result
        Future<String> future = executor.submit(() -> "foo");
        String result = future.join();
    }
}
DEADLINES

• Instead of saying that the Continuation scope is cancellable, we can give it a deadline and say that we are only willing to wait up until that deadline for any of the threads.

• An ExecutorService can be wrapped with a deadline
  • Executors.newVirtualThreadExecutor(Instant deadline)

• The deadline can expire before the executor has terminated

• Deadlines apply to nested usages
```java
var deadline = Instant.now().plusMillis(8);
try (ExecutorService executor = Executors.newVirtualThreadExecutor(deadline)) {
    // Submits two value-returning tasks to get a Stream that is lazily populated
    // with completed Future objects as the tasks complete
    Stream<Future<String>> stream = executor.submit(List.of() -> "foo", () -> "bar"
        stream.filter(Future::isCompletedNormally)
        .map(Future::join)
        .forEach(System.out::println);
}
```
Virtual threads are just objects in the heap, there may be millions of them. There is no API support in the debugger API for enumerating all virtual threads.

Virtual threads are not active members of a thread group.

JDWP EventRequest/Set defines a new PlatformThreadsOnly filter that can be used when requesting THREAD_START and THREAD_END events.

JDI ThreadReference defines isVirtual() to test if a thread is a virtual thread.
Temporarily, VirtualMachine defines `supportVirtualThreads()` to test if the target VM supports virtual threads.

**Temporary JDWP options**

- `enumeratevthreads=y|n` with default as “n”
- `notifyvthreads=y|n` with default as “y”
REFERENCES

- Project Loom Wiki Page https://wiki.openjdk.java.net/display/loom
- Early Access Builds on Java 18 https://jdk.java.net/loom/
EVALUATE THE SESSION

Please help by rating and giving the feedback.
JOIN THE CONVERSATION:

@EclipseCon | #EclipseCon
THANK YOU!