Main

Project Loom is to intended to explore, incubate and deliver Java VM features and APIs built on top of them for the purpose of supporting easy-to-use, high-throughput lightweight concurrency and new programming models on the Java platform. This is accomplished by the addition of the following constructs:

- Fibers (lightweight user-mode threads)
- Delimited continuations
- Tail-call elimination

This OpenJDK project is sponsored by the HotSpot Group.

Source Code

https://github.com/openjdk/loom

Early Access Binaries

http://jdk.java.net/loom/

Talks

Devoxx BE 2019 - Video

JVMLS 2019 - Video

Curry On 2019 - Video

QCon London 2019 - Video and Slides

FOSDEM 2019 - Video

Devoxx BE 2018 - Video | Slides

JVMLS 2018 – Video | Slides

JFokus 2018 – Video

Meetings

October 2018 - Slides

Mailing List

Subscribe | Archive

Project

Proposal | JEP | Members | Page

Note

Loom is under active development, which means that information and advice given here might change in the future.

Supported Platforms

Mac and Linux on x86-64

Download and Build from Source

```
$ git clone https://github.com/openjdk/loom
$ cd loom
$ git checkout fibers
$ sh configure
$ make images
```

(Note that you must switch to the `fibers` branch before building)

How to Contribute

The most valuable way to contribute at this time is to try out the current prototype and provide feedback and bug reports to the loom-dev mailing list. In particular, we welcome feedback that includes a brief write-up of experiences adapting existing libraries and frameworks to work with Fibers.

If you have a login on the JDK Bug System then you can also submit bugs directly. We plan to use an Affects Version/s value of "repo-loom" to track bugs.

How to run the JDK tests

1. Download `jtreg` (the JDK test harness) and place its `bin` subdirectory on your path.
2. Create a debug JDK configuration (inside the top directory of the Loom repo) and build it. This step requires having `jtreg` on your path, or running the tests would fail:

    ```
    $ sh configure --with-jtreg --with-debug-level=fastdebug
    $ make images
    ```

3. Run the tests. The following example assumes a Mac build (replace `macosx` with `linux` for a Linux build), and the `java/lang/Continuation/Basic.java` test, which contains some basic `Continuation` tests. The `java/lang/Continuation` directory contains Continuation test, while the `java/lang/Continuation` directory contains fiber tests. Supplying just the directory name runs all tests in the directory.

    ```
    $ make run-test TEST=open/test/jdk/java/lang/Continuation/Basic.java CONF=macosx-x86_64-server-fastdebug
    ```

Missing Features

- Yielding while a native VM frame is on the stack in the case of a priviliged action, reflective invocation and `MethodHandle` invocation.
- Cloning continuations
- Serialization of fiber/continuation

Continuations

Design

The primitive continuation construct is that of a scoped (AKA multiple-named-prompt), stackful, one-shot (non-reentrant) delimited continuation. The continuation can be cloned, and thus used to implement reentrant delimited continuations. The construct is exposed via the `java.lang.Continuation` class. Continuations are intended as a low-level API, that application authors are not intended to use directly. They will use higher-level constructs built on top of continuations, such as fibers or generators.

A continuation object is constructed by passing two arguments to the constructor: a `Runnable` target that serves as the body of the continuation, and a `java.lang.ContinuationScope`. The scope is the delimited continuation's prompt, that allows continuations to be nested. One could think of such "scoped continuations" as nested `try/catch` blocks, where the scope is the type of the exception thrown, which determines the handler called.

A continuation is started by calling `Continuation.run`, which would start executing the body in the continuation's target, and returns either when the continuation terminates (the body runs to completion, and terminates either normally or abnormally), or when it yields on the continuation's scope. To query the reason for `run` returning, use `Continuation.isDone`, which returns `true` if the body has terminated, or `false` if it has yielded.

A call to the static `Continuation.yield` suspends the current continuation and all enclosing continuations up until the innermost one with the scope passd to `yield`, causing the `run` method of that continuation to return.

The scoping mechanism means that not only are Loom's continuations composable, but they are also well encapsulated. By keeping the scope object hidden (say, in a private static field), a construct using implementations can prevent its continuations from being yielded directly and circumventing the construct's API (e.g. it is impossible to directly yield a fiber's continuation because the scope of those continuations is private to the fiber implementation).

The `Continuation` class does not provide a mechanism of communication between `run` and `yield` (i.e., neither takes or returns a value that is passed to/received from the other), unlike most implementations of delimited continuations. However, implementing a class that does allow this kind of communication on top of the `Continuation` class is straightforward, and will likely be included in the JDK.

Implementation

The current prototype limits the situation in which a continuation can be yielded: a continuation cannot yield while a native method is on its stack (i.e. on the stack used by the continuation body when it attempts to yield) – this can happen when a VM or a JNI method is called from the continuation body and then calls back into Java code that attempts to yield – or while the body of the continuation holds a native monitor (i.e. a yield is called – directly or indirectly – from inside a synchronized method or a synchronized block). In these situations, the continuation is said to be *pinned* (to the mounted thread). Attempting to yield while pinned results in a call to the protected method `Continuation.onPinned`, which by default throws an exception (fibers override this behavior; see below).

It is likely that we will never support yielding with a JNI method on the stack, but we will support most common cases of VM methods – privileged actions (this may be resolved by implementing that security feature in Java), reflective invocation (this may also be a non-issue, as reflection does support a pure Java path by generating bytecode), and `MethodHandle` invocation. It is not yet decided whether or not we will choose to support yielding while holding native monitors.

The `Continuation` class is implemented natively in Hotspot (except for scoping; that is implemented in Java). Every continuation has its own stack. From the perspective of the implementation, starting or continuing a continuation *mounts* it and its stack on the current thread – conceptually concatenating the continuation's stack to the thread's – while yielding a continuation *unmounts* or *dismounts* it.

The current prototype implements the mount/dismount operations by copying stack frames from the continuation stack – stored on the Java heap as two Java arrays, an `Object` array for the references on the stack and a primitive array for primitive values and metadata. Copying a frame from the thread stack (which we also call the *vertical* stack, or the *v-stack*) to the continuation stack (also, the *horizontal* stack, or the *h-stack*) is called *freezing* it, while copying a frame from the h-stack to the v-stack is called *thawing*. The prototype also optionally thaws just a small portion of the h-stack when mounting using an approach called *lazy copy*; see the JVMLS 2018 talk as well as the section on performance for more detail.

Hypothetically, the most efficient implementation of the mounting operations involves merely linking the current thread stack to the continuation's (with dismounting translating to unlinking). When a continuation is mounted, its body executes "in" (i.e. using) the continuation's stack. The current prototype, however, employs copying due to technical constraints which make implementing the linking approach costly. Some of Hotspot's GCs cannot easily support heap objects that can store references in memory offsets that change throughout the lifetime of the heap object; changing that assumption would require deep changes to some of Hotspot's GCs. Storing h-stacks in some special off-heap memory area would not help because this means that the GCs would need to scan them (like they scan thread stacks) – which is potentially a slow operation, especially given the need to support a very large number of continuations – sometimes even during stop-the-world (STW) phases, like young-gen collection. Ordinarily, heap objects do not need to be scanned because writing to them updates efficient GC data-structures, but those updates (called write barriers) are not executed for stack writes in the execution "engines" (the interpreter and compilers); inserting those barriers when running a continuation body would require changing all four OpenJDK execution engines: the interpreter, C1, C2 and Graal. This, too, is costly.

However, we believe we can make the copying approach efficient enough, and that complex changes to the GC or the execution engines will prove unnecessary.

Performance

Current yield/continue performance is far from stellar. The reason is that we focused on getting a working prototype using existing Hotspot mechanisms, some of which have not been designed to be used so frequently. We are now working on improving performance both by optimizing the actual freeze/thaw logic, as well as optimizing those existing VM mechanisms.

One mechanism that is particularly slow is the one used to detect whether a frame is holding a native monitor (synchronized block). Because this mechanism is so slow, the monitor detection can be turned off (which will break fiber code that runs through synchronized blocks) by adding `-XX:-DetectLocksInCompiledFrames` to the `java` command line.

An important performance feature is lazy-copying of frames. This feature is currently turned off by default. To turn lazy copying on, add `-XX:+UnlockDiagnosticVMOptions -XX:+UseNewCode` to the `java` command line.

Fibers

Design

Fibers are user-mode lightweight threads that allow synchronous (blocking) code to be efficiently scheduled, so that it performs as well as asynchronous code, but is simpler to read and write, debug, monitor and profile.

There is a tension in the design of fibers between attempting to run as much existing code as possible and our desire to take the opportunity to rethink how modern threads should work, with an eye to new software requirements, programming styles and new/changing hardware. Even if we choose not to run every piece of existing code in fibers, we should provide an easy path to migration.

In the current prototype, fibers are implemented by the java.lang.Fiber API. The API supports scheduling a fiber to execute a task and waiting for a fiber to terminate (to retrieve the result of the task). Project Loom is exploring the concepts of Structured Concurrency so that fibers are scheduled in a scope that can not be exited until all fibers scheduled in the scope have terminated. The Fiber API also supports interoperability with code using the java.util.concurrent.Future or CompletableFuture APIs.

To allow existing code to run in the context of a fiber, the current prototype emulates Thread.currentThread() so that the existing Thread APIs, ThreadLocal, InheritedThreadLocal etc. work as before. In essence, all locals are *fiber local.*

Implementation

Fibers are implemented in the core libraries. A fiber is implemented as a continuation (of fiber scope) that is wrapped as a task and scheduled by a `j.u.c.Executor`. Parking (blocking) a fiber results in yielding its continuation, and unparking it results in the continuation being resubmitted to the scheduler. The scheduler worker thread executing a fiber (while its continuation is mounted) is called a *carrier* thread.

The continuations used in the fiber implementation override `onPinned` such that if a fiber attempts to park while its continuation is pinned (see above), it will block the underlying carrier thread.

The implementation of the networking APIs in the *java.net* and *java.nio.channels* packages have as been updated so that fibers doing blocking I/O operations park, rather than block in a system call, when a socket is not ready for I/O. When a socket is not ready for I/O it is registered with a background multiplexer thread. The fiber is then unpacked when the socket is ready for I/O.

Debugging

See the Virtual Thread Debugging Support page.

Tail Calls

Design

We envision tail-call elimination that pops one or perhaps even an arbitrary number of stack frames at explicitly marked call-sites. It is not the intention of this project to implement *automatic* tail-call optimization.

Implementation

The implementation of this feature requires cross-cutting changes to the VM, VM specification (bytecode), and possibly the front-end Java compiler (javac). As a result, in order not to delay the completion of continuations and fibers, we will only begin specifying and implementing this feature only when the project is at a more advanced phase.