State of Loom: Part 2

Ron Pressler, May 2020

Contents

State of Loom: Part 1 introduced virtual threads and explained how the JDK has been adapted to use them. With threads more numerous and shorter lived than before, new ways of managing them and distributing workload among them fall under the purview of Project Loom: New, flexible mechanisms for passing data among threads — like **channels** — might be desirable; herding a large number of threads could benefit from a means to organize and supervise them called **structured concurrency**; finally, we're exploring a construct more convenient and performant than thread-locals for context data that we provisionally call **scope variables**.

As always, feedback to the loom-dev mailing list *reporting on your experience using Loom* will be much appreciated.

Channels

When it comes to communicating a single result computed by a thread, the `java.util.concurrent.Future` interface is adequate, and the `CompletableFuture` class offers a nice way to bridge between the synchronous and asynchronous worlds: use the `thenXXX` methods for asynchronous, `get` for synchronous. When it comes to communicating multiple results, the `Flow` class offers a good solution for asynchronous code, and until more synchronous solutions are designed, `BlockingQueue`s can be used to communicate multiple values among threads (in particular, `LinkedTransferQueue`).

However, a `BlockingQueue` can't cleanly communicate "end-of-data" or errors, and the sender and receiver ends cannot be cleanly separated, which makes attaching filters, mappers and other compositions, or representing distributed messaging, rather clumsy. It might be nice to have a proper channel type — the synchronous counterpart to `Flow` — and Doug Lea has started working on its design. Pattern matching can make working with channels and messages quite appealing.

In one prototype, the channel type is called `Carrier` to distinguish it from NIO channels. Although likely to change, it looks like this at present:

```
Carrier<String> carrier = new Carrier<>();

Thread producer = Thread.startVirtualThread(() -> {
  Carrier.Sink<String> sink = carrier.sink();
  sink.send("message1");
  sink.send("message2");
  sink.closeExceptionally(new InternalError());
});

Thread consumer = Thread.startVirtualThread(() -> {
  try (Carrier.Source<String> source = carrier.source()) {
    while (true) {
      String message = source.receive();
      System.out.println(message);
    }
  } catch (IllegalStateException e) {
    System.out.println("consumer: " + e + " cause: " + e.getCause());
  }
});

producer.join();
consumer.join();
```

Structured Concurrency

With threads cheap and numerous, they could benefit from standard practices of herding them. We found one idea on how to do this particularly appealing: *structured concurrency*.[1]

Structured concurrency corrals thread lifetimes into code blocks. Similar to how structured programming confines control flow of sequential execution into a well-defined code block, structured concurrency does the same with concurrent control flow. Its basic principle is this: threads that are created in some code unit must all terminate by the time we exit that code unit; if execution splits to multiple thread inside some scope, it must join before exiting the scope. In particular, a method should not return after spawning a thread that can live on indefinitely.

To make the idea clearer, let's look at some code. In our current prototype we represent a structured concurrency scope, the code block that confines the lifetime of child threads, by making the `java.util.concurrent.ExecutorService` an `AutoCloseable`, with `close` shutting down the service and awaiting termination. This guarantees that all tasks submitted to the service will have terminated by the time we exit the try-with-resources (TWR) block, confining their lifetime to the code structure:

```
ThreadFactory vtf = Thread.builder().virtual().factory();
try (ExecutorService e = Executors.newUnboundedExecutor(vtf)) {
   e.submit(task1);
   e.submit(task2);
} // blocks and waits
```

Before we exit the TWR block, the current thread will block, waiting for all tasks — and their threads — to finish. Once outside it, we are guaranteed that the tasks have terminated.

Our code, then, directly lends itself to a very organized representation similar to some process calculi. If `;` is sequential composition and `|` is parallel composition, our code could be described like so: `a;(b|((c|d);e));f`, where the join point — where we wait for child threads to finish — is the closing parentheses after parallel operations and before the next sequential one, as in `(x|y|z);w`.

Aside from clarity, this structure imposes a powerful invariant: every thread (in a structured concurrency context) has some "parent" thread that is blocked waiting for it to terminate and so cares about when — and perhaps how — it does.[2] It has some powerful advantages when it comes for error propagation and cancellation.

Structured Interruption

An unstructured thread is detached from any context or clear responsibility. Since a structured thread clearly performs some work for its parent, when the parent is cancelled the child should also be cancelled.

Thus, if the parent thread is interrupted, we propagate the interruption its child threads. We can also give all tasks a deadline that will interrupt those children that have yet to terminate by the time it expires (as well as the current thread):

```
try (var e = Executors.newUnboundedExecutor(myThreadFactory)
                      .withDeadline(Instant.now().plusSeconds(30))) {
   e.submit(task1);
   e.submit(task2);
}
```

Structured Errors

An unstructured thread might suffer an exception and die alone without anyone noticing; a structured thread's failure will be observed by its watchful parent, and the failure can then be put in context, for example by stitching the child exception's stack trace to its parent's stack trace.

But error propagation poses some challenges. Suppose that an exception thrown by a child would automatically propagate to its parent that, as a result, would then cancel (interrupt) all of its other children. This may well be desirable in some situations, but that this should be the *default* behavior is not so clear. So for the time being, we're experimenting with more explicit error and result handling.

We can use the new `ExecutorService.submitTasks` and `CompletableFuture.stream`, that streams the result, successful or not, of each task as it completes (and also serves as a bridge to the asynchronous world of `CompletableFuture`) to wait for the first task that completes successfully, and then cancel all others:

```
try (var e = Executors.newUnboundedVirtualThreadExecutor()) {
  List<CompletableFuture<String>> tasks = e.submitTasks(List.of(
    () -> "a",
    () -> { throw new IOException("too lazy for work"); },
    () -> "b",
  ));

  try {
    String first = CompletableFuture.stream(tasks)
      .filter(Predicate.not(CompletableFuture::isCompletedExceptionally))
```

```
        .map(CompletableFuture::join)
        .findFirst()
        .orElse(null);
      System.out.println("one result: " + first);
    } catch (ExecutionException ee) {
      System.out.println("¯\\_(ツ)_/¯");
    } finally {
      tasks.forEach(cf -> cf.cancel(true));
    }
  }
```

Some common patterns could be served by helper methods, such as **ExecutorService**'s **invokeAll** or **invokeAny**. This example does the same thing as the more verbose one above:

```
try (var e = Executors.newUnboundedVirtualThreadExecutor()) {
  String first = e.invokeAny(List.of(
    () -> "a",
    () -> { throw new IOException("too lazy for work"); },
    () -> "b"
  ));
  System.out.println("one result: " + first);
} catch (ExecutionException ee) {
  System.out.println("¯\\_(ツ)_/¯");
}
```

These APIs are in the EA, but there are likely to be many changes in this area as we try to make managing threads more friendly.

Structured serviceability and observability

Structured concurrency does not only help organizing code. It helps give meaningful context when profiling and debugging. A thread dump of a million threads might not be useful, but if those threads could be presented in a tree arranged according to the structured concurrency scope hierarchy, more sense could be made of them; similarly, JFR could group threads and the operations they perform by SC scopes, allowing to zoom in or out on profiles. This work is unlikely to make it into the first Preview.

Scope Variables

We sometimes need to pass some context from a caller to a transitive callee in a way that's transparent to intermediate frames. For example, suppose that in a chain of calls, **foo**→**bar**→**baz**, **foo** and **baz** are application code while **bar** is library code or vice versa, and **foo** wants to share data with **baz** without **bar**'s participation. Today this is often accomplished with **ThreadLocal**s, but TLs, as we'll call them for short, have serious shortcomings.

For one, they're unstructured in a similar sense to the one we used above: once a TL value is set, it is in effect throughout the thread's lifetime or until it is set to some other value. In fact, we commonly see a usage pattern that tries to lend TLs structure (unfortunately, without any performance benefits):

```
var oldValue = myTL.get();
myTL.set(newValue);
try {
  ...
} finally {
  myTL.set(oldValue);
}
```

Without this imposed structure, when a thread is shared among multiple tasks, one task's TL values might leak into another. Virtual threads solve that problem by being lightweight enough to not require sharing. However, this unstructuredness also means that the TL implementation must rely on weak references to allow TLs that are no longer used to be cleaned up by the GC, and that makes their implementation significantly slower.

Another issue is inheritance. For example, those working with distributed tracing like OpenTracing, might want to inherit a tracing "span" from a parent thread. This can be achieved with **InheritableThreadLocal** (iTL). The iTL map in the thread must be *copied* when a thread is created because (i)TLs are mutable and so cannot be shared. This incurs both a footprint and a speed hit. Plus, because today's threads are unstructured, by the time a child thread accesses its inherited span, its parent might have closed it.

The problem of TL inheritance is only exacerbated by virtual threads because they encourage creating many small threads, some doing small tasks, like a single HTTP request, on behalf of their parent, thus increasing both the need for TL inheritance as well as the compounded footprint and speed costs.

If TLs were immutable after having been set once, inheritance could be made efficient, but consider a method that sets a TL; it would now throw an illegal state exception depending on whether any of its callers has set the same TL, significantly hurting code composability.

To solve these issues we're exploring an alternative, better in performance, footprint, structure and correctness, that we're tentatively calling *scope variables*. Like TLs, SVs introduce some implicit context, but unlike TLs, they are structured and in effect for the span of a code block, not for the entire lifetime of the thread. SVs are also immutable, although their value can be shadowed by nested scopes.[3]

Here is an example using the `java.lang.Scoped` API that's in the current EA prototype:

```
static final Scoped<String> sv = Scoped.forType(String.class);

void foo() {
    try (var __ = sv.bind("A")) {
    bar();
    baz();
    bar();
  }
}

void bar() {
  System.out.println(sv.get());
}

void baz() {
  try (var __ = sv.bind("B")) {
    bar();
  }
}
```

`baz` does not *mutate* `sv`'s binding but, rather introduces a new binding in a nested scope that shadows its enclosing binding. So `foo` will print:

```
A
B
A
```

Because the lifetime of an SV binding is well-defined, we do not need to rely on the GC for cleanup, and so we do not require weak references that slow us down.

What about inheritance? Because SVs are immutable, and because structured concurrency also gives us a syntax-confined thread lifetime, SV inheritance fits structured concurrency like a glove:

```
try (var context = Foo.openContext()) { // some temporary context that can be closed
  try (var __ = contextSV.bind(context);
       var executor = Executors.newUnboundedExecutor(myThreadFactory)) {
    executor.submit(() -> { ... });
    executor.submit(() -> { ... });
  }
}
```

The submitted tasks automatically inherit the `context` value of `contextSV`, and because the unbounded executor's scope is enclosed by `context`'s lifetime scope, the tasks can be certain that the context they obtain from `contextSV` has not been closed.

Other kinds of structured constructs, meaning constructs whose computation is confined to a syntax element, could also provide automatic SV inheritance. For example:

```
try (var __ = sv.bind(value)) {
    Set.of("a", "b", "c").stream().parallel()
       .forEach(s -> System.out.println(s + ": " + sv.get()));
}
```

Because the `forEach` operation of the stream is also entirely confined to the SV's binding scope, `value` can be inherited, even though `forEach` might execute its body on different stream elements in different threads.

Scope variables are still early in the design stage and are tied to more general changes we might introduce to try-with-resources (see here for some ideas). Even if we decide to go ahead with SVs, they will likely miss the first Preview and GA.

Processor Locals

Another use of thread-locals is not to associate data with a thread context but to “stripe” some write-heavy, mutable data structure to avoid contention (e.g. `LongAdder`, which doesn’t use the `ThreadLocal` class, but relies on a similar idea). This makes sense when the number of threads is not much greater than the number of cores, but with possibly millions of threads it is pure overhead. We’re exploring a “processor local” construct with CAS-like semantics, that is faster than even an uncontended CAS given appropriate OS support, such as Linux’s restartable sequences.

More on Interruption and Cancellation

Threads support a cooperative interruption mechanism comprised of the methods `interrupt()`, `interrupted()`, `isInterrupted()`, and `InterruptedException`. This is a rather complex mechanism: some thread calls `interrupt` on another, setting the target thread’s interrupted status. The target thread polls for its interrupted status, perhaps throwing an `InterruptedException` from blocking methods, but also *clears* that status, which is necessary for two reasons. First, the thread might be a shared resource in a pool. The current task could be interrupted, but the scheduler may want to reuse the thread to run other tasks, and so the interrupted status must be reset. This is unnecessary with virtual threads, as they are lightweight enough to not be reused for different tasks. But there is another reason: a thread might observe that it’s been interrupted, and *as part of its cleanup process*, may wish to call a blocking method. If the status were not cleared, the blocking method would immediately throw `InterruptedException`.

While this mechanism does address a real need, it is error-prone, and we’d like to revisit it. We’ve experimented with some prototypes, but, for the moment, don’t have any concrete proposals to present.

Forced Preemption

Despite what was said about scheduling, there can be special circumstances where forcefully preempting a thread that is hogging the CPU can be useful. For example, a batch processing service that performs complex data queries on behalf of multiple client applications may receive client tasks and run them each in its own virtual thread. If such a task takes up too much CPU, the service might want to forcefully preempt it and schedule it again when the service is under a lighter load. To that end, we plan for the VM to support an operation that tries to forcefully preempt execution at any safepoint. How that capability will be exposed to the schedulers is TBD, and will likely not make it to the first Preview.

Reviewed by Alan Bateman, Chris Plummer, Duncan MacGregor, Andrew Haley, Doug Lea, John Rose, Mark Reinhold, Alex Buckley, and Brian Goetz.

---

1. The term *Structured Concurrency* was coined by Martin Sústrik who introduced it in his blog post *Structured Concurrency* (followed by *Update on Structured Concurrency*), and further explained and popularized by Nathaniel J. Smith in his blog post *Notes on structured concurrency, or: Go statement considered harmful* (also related: *Timeouts and cancellation for humans*). These posts have also influenced several other languages to investigate this idea.↩
2. This is not unlike Erlang’s supervision mechanism, only the supervision hierarchy is reflected in the code’s syntactic structure.↩
3. They work like dynamic binding in Clojure, which has a long history in Lisp.↩