Structured Concurrency

The core concept of structured concurrency is that when control splits into concurrent tasks that they join up again. If a “main task” splits into several concurrent sub-tasks scheduled to be executed in fibers then those fibers must terminate before the main task can complete.

The main benefit of structured concurrency is abstraction. A caller of a method that is invoked to do a task should not care if the method decomposes the task and schedules a million fibers. When the method completes, any fibers scheduled by the method should have terminated.

Further reading:

Nathaniel J. Smith: Notes on structured concurrency, or: Go statement considered harmful

Martin Sustrik: Structured Concurrency

Current prototype

In Project Loom, a prototype API has been developed called *FiberScope* that is a scope in which fibers are scheduled. Here is a basic example:

```
try (var scope = FiberScope.cancellable()) {
    var fiber1 = scope.schedule(task);
    var fiber2 = scope.schedule(task);
}
```

A thread or fiber enters a scope by calling the FiberScope.cancellable method (we will explain cancellation later). It exits the scope when the code in the block completes and any fibers scheduled in the scope have terminated. The example schedules two fibers. The thread/fiber executing the above code may have to wait (in the FiberScope’s close method) until the two fibers have terminated.

FiberScopes can be nested, consider the following:

```
try (var scope1 = FiberScope.cancellable()) {
    scope1.schedule(task);
    try (var scope2 = FiberScope.cancellable()) {
        scope2.schedule(task);
    }
    scope1.schedule(task);
}
```

In this example, a thread or fiber enters scope1, schedules a fiber, then enters scope2 where it schedules a fiber in that scope. The execution cannot exit scope2 until the fiber scheduled in that scope terminates. When it exits, it is back in scope1, where it scheduled another fiber. It cannot exit scope1 until the two fibers scheduled in that scope have terminated.

More generally, the structured approach leads to *trees* of tasks. Consider the following method x that schedules two fibers to execute foo and bar. The code in foo and bar each schedule two fibers.

```
void x() {
    try (var scope1 = FiberScope.cancellable()) {
        var fiber1 = scope1.schedule(() -> foo());
        var fiber2 = scope1.schedule(() -> bar());
    }
}
void foo() {
    try (var scope2 = FiberScope.cancellable()) {
        scope2.schedule(() -> task());
        scope2.schedule(() -> task());
    }
}
void bar() {
    try (var scope3 = FiberScope.cancellable()) {
        scope3.schedule(() -> task());
        scope3.schedule(() -> task());
    }
}
```

fiber1 is scheduled in scope1 to execute foo. It enters scope2 and schedules two fibers. It exits scope2 (and returns to scope1) when the two fibers terminate. fiber2 is scheduled in scope1 to execute bar. It enters scope3 and schedules two fibers. It exits scope3 (and returns to scope1) when the two fibers terminate. The thread or fiber executing will not exit scope1 until fiber1 and fiber2 have terminated.

As an escape hatch, the FiberScope API defines the *detached()* method to return a scope which can be used to schedule fibers that are intended to outline the context where they are initially scheduled.

Cancellation

A fiber executing in *cancellable* scope may be cancelled by invoking its cancel method to set the fiber's cancel status and unpark the fiber. Cancellation works cooperatively: a Fiber needs to check for cancellation (say, when doing blocking operations), and throw an appropriate exception for the context that it is running in (which might be an InterruptedException in methods that throw this exception, an IOException or sub-class when in a method that does I/O).

At this time, the following blocking operations wakeup and throw an exception when the fiber is cancelled:

- java.lang.Thread.sleep
- java.net.Socket: connect, read, write
- java.net.ServerSocket: accept
- java.nio.channels.SocketChannel: connect, read, write
- java.nio.channels.ServerSocketChannel: accept
- java.nio.channels.DatagramChannel: read, receive
- java.nio.channels.Pipe.SourceChannel.read
- java.nio.channels.Pipe.SinkChannel.write

There are rare, but important, cases such as shutdown or recovery steps where a fiber may need to be shielded from cancellation. To support this, FiberScope defines the nonCancellable() method to enter a non-cancellable scope. A Fiber that checks for cancellation in a non-cancellation scope will get back "false". If its cancel status is set then it will observe this when it pops back to a cancellable scope.

Deadline/Timeouts

Decomposing deadline or timeouts is very difficult to get right.

FiberScope supports entering a scope with a deadline, expressed as a *java.time.Instant*. If the deadline expires before the thread/fiber exits the scope then all fibers scheduled in the scope are cancelled and the close method throws an exception (or it gets added as a suppressed exception when exiting with an exception).

Deadlines work with nested scopes. Consider the following:

```
        var deadline = Instant.now().plusSeconds(10);
        try (var scope1 = FiberScope.withDeadline(deadline)) {
            try (var scope2 = FiberScope.cancellable()) {
                scope2.schedule(() -> task());
            }
        }
```

scope1 is entered with a deadline that is now + 10s. It schedules a fiber in scope2 and cannot exit to scope1 until the fiber terminates. If the deadline expires in the meantime then the fiber will be cancelled and the thread/fiber will throw CancelledException("Deadline expired") when existing scope1. If the inner scope had a deadline that was further into the future that the deadline then the deadline for the outer scope will expire first.

In addition to withDeadline, FiberScope also defines *withTimeout* to enter a scope with a timeout, expressed as a *java.time.Duration*. If the timeout expires before thread/fiber exits the scope then all fibers scheduled in the scope are cancelled.

Further reading:

Nathaniel J. Smith: Timeouts and cancellation for humans

Termination queues

Fibers communicate to other fibers or threads using queues or other mechanisms. They may also return a result, retrieved by invoking the Fiber's *join* method. As a convenience when scheduling fibers in a FiberScope, the API defines *schedule* methods that specify a *termination queue* that the fiber is queued to when it terminates. The following example schedules fibers to executes tasks that return Strings. The fibers are queued to a termination queue then they terminate. The results are obtained by invoking the Fiber::join method.

```
try (var scope = FiberScope.cancellable()) {
    var queue = new FiberScope.TerminationQueue<String>();

    Arrays.stream(tasks).forEach(task -> scope.schedule(task, queue));

    IntStream.range(0, tasks.length)
            .mapToObj(x -> queue.takeUninterruptibly())
            .map(Fiber::join)
            .forEach(System.out::println);
}
```

FiberScope Variables/Locals

There is no support yet for making context available to all fibers scheduled in the scope. InheritedThreadLocals can be used in the mean-time.

Issues/Discussion/TBD

- The current FiberScope API is a prototype API to demonstrate and explore concepts. For now, FiberScope is an AutoCloseable and so encourages the use of the try-with-resources construct. The advantages of this approach is that it plays well with checked exceptions and it is easy to access variables in the enclosing scope. The downside is lack of guaranteed cleanup (a ThreadDeath, OOME, or other resource issues may abort the execution of the finally block and close). It is also possible to misuse, e.g. leak the scope to a callee that closes it explicitly. An alternative API that has also been prototyped

  ```
  FiberScope.cancellable(scope -> { ... });
  ```

  The downside with this approach is that checked exceptions are awkward to deal with. It also requires capturing of effectively final variables in the enclosing scope. The API surface is also larger as there are different consumer variants to allow for a return value (or none).
  We will re-evaluate this once we have more experience with the concepts.

- At this things stand, the thread/exit exiting the scope does not automatically cancel fibers scheduled in the scope. This was prototyped, but prior to adding termination queues, and may need to looked at again. More use-cases are needed for this and other items.
- The Python Trio library has automatic propagation of errors (to ensure that errors are never lost). An approximate equivalent was prototyped with FiberScope so that a fiber terminating with an exception propagates the exception to the thread/fiber that scheduled it. With nesting, the exception might propagate up a tree. A concern with the approach is that it’s “too magic”. For now, the API requires an explicit join to obtain an exception thrown by a fiber.

- At this time, a FiberScope object need to be guarded to avoid leaking to a callee that closes the scope. Another concern is the fibers method to obtain a stream of the fibers executing in the scope. The main use-cases for the fibers method is cancellation and debugging, both of these need to be re-examined.