JEP 428: Structured Concurrency (Incubator)

Authors	Alan Bateman, Ron Pressler
Owner	Alan Bateman
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	19
Component	core-libs
Discussion	loom dash dev at openjdk dot java dot net
Reviewed by	Alex Buckley, Brian Goetz
Created	2021/11/15 15:01
Updated	2023/06/08 16:05
Issue	8277129

Summary

Simplify multithreaded programming by introducing an API for *structured concurrency*. Structured concurrency treats multiple tasks running in different threads as a single unit of work, thereby streamlining error handling and cancellation, improving reliability, and enhancing observability. This is an incubating API.

Goals

- Improve the maintainability, reliability, and observability of multithreaded code.
- Promote a style of concurrent programming which can eliminate common risks arising from cancellation and shutdown, such as thread leaks and cancellation delays.

Non-Goals

- It is not a goal to replace any of the concurrency constructs in the `java.util.concurrent` package, such as `ExecutorService` and `Future`.
- It is not a goal to define the definitive structured concurrency API for Java. Other structured concurrency constructs can be defined by third-party libraries or in future JDK releases.
- It is not a goal to define a means of sharing streams of data among threads (i.e., *channels*). We might propose to do so in the future.
- It is not a goal to replace the existing thread interruption mechanism with a new thread cancellation mechanism. We might propose to do so in the future.

Motivation

Developers manage complexity by breaking tasks down into multiple subtasks. In ordinary single-threaded code, the subtasks execute sequentially. However, if the subtasks are sufficiently independent of each other, and if there are sufficient hardware resources, then the overall task can be made to run faster (i.e., with lower latency) by executing the subtasks concurrently. For example, a task that composes the results of multiple I/O operations will run faster if each I/O operation executes concurrently in its own thread. Virtual threads (JEP 425) make it cost-effective to dedicate a thread to every such I/O operation, but managing the huge number of threads that can result remains a challenge.

Unstructured concurrency with `ExecutorService`

The `java.util.concurrent.ExecutorService` API, introduced in Java 5, helps developers execute subtasks concurrently.

For example here is a method, `handle()`, that represents a task in a server application. It handles an incoming request by submitting two subtasks to an `ExecutorService`. One subtask executes the method `findUser()` and the other subtask executes the method `fetchOrder()`. The `ExecutorService` immediately returns a `Future` for each subtask, and executes each subtask in its own thread. The `handle()` method awaits the subtasks' results via blocking calls to their futures' `get()` methods, so the task is said to *join* its subtasks.

```
Response handle() throws ExecutionException, InterruptedException {
    Future<String>  user  = esvc.submit(() -> findUser());
    Future<Integer> order = esvc.submit(() -> fetchOrder());
    String theUser  = user.get();   // Join findUser
    int    theOrder = order.get();  // Join fetchOrder
    return new Response(theUser, theOrder);
}
```

Because the subtasks execute concurrently, each subtask can succeed or fail independently. (Failure, in this context, means to throw an exception.) Often, a task such as `handle()` should fail if any of its subtasks fail. Understanding the lifetimes of the threads can be surprisingly complicated when failure occurs:

- If `findUser()` throws an exception then `handle()` will throw an exception when calling `user.get()` but `fetchOrder()` will continue to run in its own thread. This is a *thread leak* which, at best, wastes resources; at worst, the `fetchOrder()` thread will interfere with other tasks.
- If the thread executing `handle()` is interrupted, the interruption will not propagate to the subtasks. Both the `findUser()` and `fetchOrder()` threads will leak, continuing to run even after `handle()` has failed.
- If `findUser()` takes a long time to execute, but `fetchOrder()` fails in the meantime, then `handle()` will wait unnecessarily for `findUser()` by blocking on `user.get()` rather than cancelling it. Only after `findUser()` completes and `user.get()` returns will `order.get()` throw an exception, causing `handle()` to fail.

In each case, the problem is that our program is logically structured with task-subtask relationships, but these relationships exist only in the developer's mind. This not only creates more room for error, but it makes diagnosing and troubleshooting such errors more difficult. Observability tools such as thread dumps, for example, will show `handle()`, `findUser()`, and `fetchOrder()` on the call stacks of unrelated threads, with no hint of the task-subtask relationship.

We might attempt to do better by explicitly cancelling other subtasks when an error occurs, for example by wrapping tasks with `try-finally` and calling the `cancel(boolean)` methods of the futures of the other tasks in the catch block for the failing task. We would also need to use the `ExecutorService` inside a `try-with-resources statement`, as shown in the examples in JEP 425, because `Future` does not offer a way to wait for a task that has been cancelled. But all this can be very tricky to get right, and it often makes the logical intent of the code harder to discern. Keeping track of the inter-task relationships, and manually adding back the required inter-task cancellation edges, is asking a lot of developers.

This need to manually coordinate lifetimes is due to the fact that `ExecutorService` and `Future` allow unrestricted patterns of concurrency. There are no constraints upon, or ordering of, any of the threads involved. One thread can create an `ExecutorService`, a second thread can submit work to it, and the threads which execute the work have no relationship to either the first or second thread. Moreover, after a thread has submitted work, a completely different thread can await the results of execution. Any code with a reference to a `Future` can join it (i.e., await its result by calling `get()`), even code in a thread other than the one which obtained the `Future`. In effect, a subtask started by one task does not have to return to the task that submitted it. It could return to any of a number of tasks — or even none.

Because `ExecutorService` and `Future` allow for such unstructured use they do not enforce or even track relationships among tasks and subtasks, even though such relationships are common and useful. Accordingly, even when subtasks are submitted and joined in the same task, the failure of one subtask cannot automatically cause the cancellation of another: In the above `handle()` method, the failure of `fetchOrder()` cannot automatically cause the cancellation of `findUser()`. The future for `fetchOrder()` is unrelated to the future for `findUser()`, and neither is related to the thread that will ultimately join it via its `get()` method. Rather than ask developers to manage such cancellation manually, we want to reliably automate it.

Task structure should reflect code structure

In contrast to the freewheeling assortment of threads under `ExecutorService`, the execution of single-threaded code always enforces a hierarchy of tasks and subtasks. The body block `{...}` of a method corresponds to a task, and the methods invoked within the block correspond to subtasks. An invoked method must either return to, or throw an exception to, the method that invoked it. It cannot outlive the method that invoked it, nor can it return or throw an exception to a different method. Thus all subtasks finish before the task, each subtask is a child of its parent, and the lifetime of each subtask relative to the others and to the task is governed by the syntactic block structure of the code.

For example, in this single-threaded version of `handle()` the task-subtask relationship is apparent from the syntactic structure:

```
Response handle() throws IOException {
    String theUser  = findUser();
    int    theOrder = fetchOrder();
    return new Response(theUser, theOrder);
}
```

We do not start the `fetchOrder()` subtask until the `findUser()` subtask has completed, whether successfully or unsuccessfully. If `findUser()` fails then we do not start `fetchOrder()` at all, and the `handle()` task fails implicitly. The fact that a subtask can return only to its parent is significant: It implies that the parent task can implicitly treat the failure of one subtask as a trigger to cancel all remaining subtasks and then fail itself.

In single-threaded code, the task-subtask hierarchy is reified in the call stack at run time. We thus get the corresponding parent-child relationships, which govern error propagation, for free. When observing a single thread, the hierarchical relationship is obvious: `findUser()` (and later `fetchOrder()`) appear subordinate to `handle()`.

Multithreaded programming would be easier, more reliable, and more observable if the parent-child relationships between tasks and their subtasks were expressed syntactically and reified at run time — just as for single-threaded code. The syntactic structure would delineate the lifetimes of subtasks and enable a runtime representation of the inter-thread hierarchy, analogous to the intra-thread call stack. That representation would enable error propagation and cancellation as well as meaningful observation of the concurrent program.

(Java already has an API for imposing structure on concurrent tasks, namely `java.util.concurrent.ForkJoinPool`, which is the execution engine behind parallel streams. However, that API is designed for compute-intensive tasks rather than tasks which involve I/O.)

Structured concurrency

Structured concurrency is an approach to multithreaded programming that preserves the readability, maintainability, and observability of single-threaded code. It embodies the principle that

> *If a task splits into concurrent subtasks then they all return to the same place, namely the task's code block.*

The term "structured concurrency" was coined by Martin Sústrik and popularized by Nathaniel J. Smith. Ideas from other languages, such as Erlang's hierarchical supervisors, inform the design of error handling in structured concurrency.

In structured concurrency, subtasks work on behalf of a task. The task awaits the subtasks' results and monitors them for failures. As with structured programming techniques for code in a single thread, the power of structured concurrency for multiple threads comes from two ideas: (1) well-defined entry and exit points for the flow of execution through a block of code, and (2) a strict nesting of the lifetimes of operations in a way that mirrors their syntactic nesting in the code.

Because the entry and exit points of a block of code are well defined, the lifetime of a concurrent subtask is confined to the syntactic block of its parent task. Because the lifetimes of sibling subtasks are nested within that of their parent task, they can be reasoned about and managed as a unit. Because the lifetime of the parent task is, in turn, nested within that of its parent, the runtime can reify the hierarchy of tasks into a tree. That tree is the concurrent counterpart of the call stack of a single thread, and observability tools can use it to present subtasks as subordinate to their parent tasks.

Structured concurrency is a great match for virtual threads, which are lightweight threads implemented by the JDK. Many virtual threads share the same operating-system thread, allowing for very large numbers of virtual threads. In addition to being plentiful, virtual threads are cheap enough to represent any concurrent unit of behavior, even behavior that involves I/O. This means that a server application can use structured concurrency to process thousands or millions of incoming requests at once: It can dedicate a new virtual thread to the task of handling each request, and when a task fans out by submitting subtasks for concurrent execution then it can dedicate a new virtual thread to each subtask. Behind the scenes, the task-subtask relationship is reified into a tree by arranging for each virtual thread to carry a reference to its unique parent, similar to how a frame in the call stack refers to its unique caller.

In summary, virtual threads deliver an abundance of threads. Structured concurrency ensures that they are correctly and robustly coordinated, and enables observability tools to display threads as they are understood by the developer. Having an API for structured concurrency in the JDK would improve the maintainability, reliability, and observability of server applications.

Description

The principal class of the structured concurrency API is `StructuredTaskScope`. This class allows developers to structure a task as a family of concurrent subtasks, and to coordinate them as a unit. Subtasks are executed in their own threads by *forking* them individually and then *joining* them as a unit and, possibly, cancelling them as a unit. The subtasks' successful results or exceptions are aggregated and handled by the parent task. `StructuredTaskScope` confines the lifetimes of the subtasks, or *forks*, to a clear lexical scope in which all of a task's interactions with its subtasks — forking, joining, cancelling, handling errors, and composing results — takes place.

Here is the `handle()` example from earlier, written to use `StructuredTaskScope` (`ShutdownOnFailure` is explained below):

```
Response handle() throws ExecutionException, InterruptedException {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        Future<String>  user  = scope.fork(() -> findUser());
        Future<Integer> order = scope.fork(() -> fetchOrder());

        scope.join();           // Join both forks
        scope.throwIfFailed();  // ... and propagate errors

        // Here, both forks have succeeded, so compose their results
        return new Response(user.resultNow(), order.resultNow());
    }
}
```

In contrast to the original example, understanding the lifetimes of the threads involved here is easy: Under all conditions their lifetimes are confined to a lexical scope, namely the body of the `try`-with-resources statement. Furthermore, the use of `StructuredTaskScope` ensures a number of valuable properties:

- *Error handling with short-circuiting* — If either the `findUser()` or `fetchOrder()` subtasks fail, the other is cancelled if it has not yet completed. (This is managed by the cancellation policy implemented by `ShutdownOnFailure`; other policies are possible).
- *Cancellation propagation* — If the thread running `handle()` is interrupted before or during the call to `join()`, both forks are cancelled automatically when the thread exits the scope.
- *Clarity* — The above code has a clear structure: Set up the subtasks, wait for them to either complete or be cancelled, and then decide whether to succeed (and process the results of the child tasks, which are already finished) or fail (and the subtasks are already finished, so there is nothing more to clean up).
- *Observability* — A thread dump, as described below, clearly displays the task hierarchy, with the threads running `findUser()` and `fetchOrder()` shown as children of the scope.

Like `ExecutorService.submit(...)`, the `StructuredTaskScope.fork(...)` method takes a `Callable` and returns a `Future`. Unlike `ExecutorService`, however, the returned future is not intended to be joined via its `get()` method or cancelled via its `cancel()` method. All forks in a scope are, rather, intended to be joined or cancelled as a unit. Two new `Future` methods, `resultNow()` and `exceptionNow()`, are designed to be used after subtasks complete, for example after calling `scope.join()`.

Using `StructuredTaskScope`

The general workflow of code using `StructuredTaskScope` is as follows:

1. Create a scope. The thread that creates the scope is its *owner*.
2. Fork concurrent subtasks in the scope.
3. Any of the forks in the scope, or the scope's owner, may call the scope's `shutdown()` method to request cancellation of all remaining subtasks.
4. The scope's owner joins the scope, i.e., all of its forks, as a unit. The owner can call the scope's `join()` method, which blocks until all forks have either completed (successfully or not) or been cancelled via `shutdown()`. Alternatively, the owner can call the scope's `joinUntil(java.time.Instant)` method, which accepts a deadline.
5. After joining, handle any errors in the forks and process their results.
6. Close the scope, usually implicitly via `try`-with-resources. This shuts down the scope and waits for any straggling forks to complete.

If the owner is a member of an existing scope (i.e., created as a fork in one), then that scope becomes the parent of the new scope. Tasks thus form a tree, with scopes as the intermediate nodes and threads as the leaves.

Every fork runs in its own newly created thread, which by default is a virtual thread. The forks' threads are owned by the scope, which in turn is owned by its creating thread, thus forming a hierarchy. Any fork can create its own nested `StructuredTaskScope` to fork its own subtasks, thus extending the hierarchy. That hierarchy is reflected in the code's block structure, which confines the lifetimes of the forks: All of the forks' threads are guaranteed to have terminated once the scope is closed, and no thread is left behind when the block exits.

Any fork in a scope, any fork in a nested scope, and the scope's owner can call the scope's `shutdown()` method at any time to signify that the task is complete — even while other forks are still running. The `shutdown()` method interrupts the threads of all forks that are still active in the scope. All forks should, therefore, be written in a way that is responsive to interruption. In effect, `shutdown()` is the concurrent analog of the `break` statement in sequential code.

When `join()` returns, all forks have either completed (successfully or not) or been cancelled. Their results or exceptions can be obtained, without any additional blocking, via their futures' `resultNow()` or `exceptionNow()` methods. (These methods throw an `IllegalStateException` if called before the future completes.)

Calling either `join()` or `joinUntil()` within a scope is mandatory. If a scope's block exits before joining then the scope will wait for all forks to terminate and then throw an exception.

It is possible for a scope's owning thread to be interrupted either before or while joining. For example, it could be a fork of an enclosing scope that has been shut down. If this occurs then `join()` and `joinUntil(Instant)` will throw an exception because there is no point in continuing. The `try`-with-resources statement will then shut down the scope, which will cancel all the forks and wait for them to terminate. This has the effect of automatically propagating the cancellation of the task to its subtasks. If the `joinUntil(Instant)` method's deadline expires before either the forks terminate or `shutdown()` is called then it will throw an exception and, again, the `try`-with-resources statement will shut down the scope.

The structured use of `StructuredTaskScope` is enforced at run time. For example, attempts to call `fork(Callable)` from a thread that is not in the tree hierarchy of the scope — i.e., the owner, the forks, and forks in nested scopes — will fail with an exception. Using a scope outside of a `try`-with-resources block and returning without calling `close()`, or without maintaining the proper nesting of `close()` calls, may cause the scope's methods to throw a `StructureViolationException`.

`StructuredTaskScope` enforces structure and order upon concurrent operations. Thus it does not implement the `ExecutorService` or `Executor` interfaces since instances of those interfaces are commonly used in a non-structured way (see below). However, it is straightforward to migrate code that uses `ExecutorService`, but would benefit from structure, to use `StructuredTaskScope`.

StructuredTaskScope resides in an incubator module, excluded by default

The examples above use the `StructuredTaskScope` API, so to run them on JDK XX you must add the `jdk.incubator.concurrent` module and, also, enable preview features in order to enable virtual threads:

- Compile the program with `javac --release XX --enable-preview --add-modules jdk.incubator.concurrent Main.java` and run it with `java --enable-preview --add-modules jdk.incubator.concurrent Main`; or,
- When using the source code launcher, run the program with `java --source XX --enable-preview --add-modules jdk.incubator.concurrent Main.java`; or,
- When using jshell, start it with `jshell --enable-preview --add-modules jdk.incubator.concurrent`

Shutdown policies

When dealing with concurrent subtasks it is common to use *short-circuiting patterns* to avoid doing unnecessary work. Sometimes it makes sense, for example, to cancel all subtasks if one of them fails (i.e., *invoke all*) or, alternatively, if one of them succeeds (i.e., *invoke any*). Two subclasses of `StructuredTaskScope`, `ShutdownOnFailure` and `ShutdownOnSuccess`, support these patterns with policies that shut down the scope upon the first fork failure or success, respectively. They also provide methods for handling exceptions and successful results.

Here is a `StructuredTaskScope` with a shutdown-on-failure policy (used also in the `handle()` example above) that runs a collection of tasks concurrently and fails if any one of them fails:

```
<T> List<T> runAll(List<Callable<T>> tasks) throws Throwable {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        List<Future<T>> futures = tasks.stream().map(scope::fork).toList();
        scope.join();
        scope.throwIfFailed(e -> e);  // Propagate exception as-is if any fork fails
        // Here, all tasks have succeeded, so compose their results
        return futures.stream().map(Future::resultNow).toList();
    }
}
```

Here is a `StructuredTaskScope` with a shutdown-on-success policy that returns the result of the first successful subtask:

```
<T> T race(List<Callable<T>> tasks, Instant deadline) throws ExecutionException {
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<T>()) {
        for (var task : tasks) {
            scope.fork(task);
        }
        scope.joinUntil(deadline);
        return scope.result();  // Throws if none of the forks completed successfully
    }
}
```

As soon as one fork succeeds this scope automatically shuts down, cancelling the remaining active forks. The task fails if all of the forks fail or if the given deadline elapses. This pattern can be useful in, for example, server applications that require a result from any one of a collection of redundant services.

While these two shutdown policies are provided out of the box, developers can create custom policies that abstract other patterns by extending `StructuredTaskScope` and overriding the `handleComplete(Future)` method.

Fan-in scenarios

The examples above focused on *fan-out* scenarios, which manage multiple concurrent outgoing I/O operations. `StructuredTaskScope` is also useful in *fan-in* scenarios, which manage multiple concurrent incoming I/O operations. In such scenarios we typically create an unknown number of forks in response to incoming requests. Here is an example of a server that forks subtasks to handle incoming connections inside a `StructuredTaskScope`:

```
void serve(ServerSocket serverSocket) throws IOException, InterruptedException {
    try (var scope = new StructuredTaskScope<Void>()) {
        try {
            while (true) {
                var socket = serverSocket.accept();
                scope.fork(() -> handle(socket));
            }
        } finally {
            // If there's been an error or we're interrupted, we stop accepting
            scope.shutdown();  // Close all active connections
            scope.join();
        }
    }
}
```

Because all of the connection-handling subtasks are created within the scope, a thread dump will display them as children of the scope's owner.

Observability

We extend the new JSON thread-dump format added by JEP 425 to show `StructuredTaskScope`'s grouping of threads into a hierarchy:

```
$ jcmd <pid> Thread.dump_to_file -format=json <file>
```

The JSON object for each scope contains an array of the threads forked in the scope, together with their stack traces. The owning thread of a scope will typically be blocked in a join method waiting for subtasks to complete; the thread dump makes it easy to see what the subtasks' threads are doing by showing the tree hierarchy imposed by structured concurrency. The JSON object for a scope also has a reference to its parent so that the structure of the program can be reconstituted from the dump.

The `com.sun.management.HotSpotDiagnosticsMXBean` API can also be used to generate such thread dumps, either directly or indirectly via the platform `MBeanServer` and a local or remote JMX tool.

Alternatives

- Do nothing. Leave it to developers to continue using the existing low-level `java.util.concurrent` APIs and continue having to carefully consider all of the exceptional conditions and lifetime-coordination problems that arise in concurrent code.
- Enhance the `ExecutorService` interface. We prototyped an implementation of this interface that always enforces structure and restricts which threads can submit tasks. However, we found it to be problematic because most uses of `ExecutorService` (and its parent interface `Executor`) in the JDK and in the ecosystem are not structured. Reusing the same API for a far more restricted concept is bound to cause confusion. For example, passing a structured `ExecutorService` instance to existing methods that accept this type would be all but certain to throw exceptions in most situations.

Dependencies

- JEP 425: Virtual Threads (Preview)