ISSN: 2754-6705

Journal of Mathematical & Computer Applications

Review Article

Open Access

Structured Concurrency in Java

Nilesh Jagnik

Los Angeles, USA

ABSTRACT

Concurrency is vital for high performance software applications. The Java language has support for concurrency but it requires developers to handle execution of concurrent tasks. This adds a learning curve and makes concurrent code harder to read, write and debug. To solve this issue, Java 21 introduces a preview feature called structured concurrency. In this paper, we review the idea behind structured concurrency and discuss how the new language feature works in Java.

***Corresponding author**

Nilesh Jagnik, Los Angeles, USA.

Received: November 01, 2023; **Accepted:** November 13, 2023, **Published:** November 17, 2023

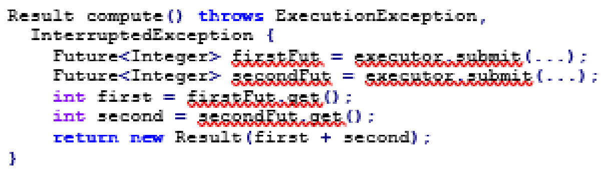
Keywords: Concurrency, Asynchronous Programming, Structured Concurrency, Multi-Threading

Introduction

In a typical program, code executes sequentially. If the program reaches a statement that blocks while waiting on I/O, then no progress is made while this happens. However, if a program has other parts that can be executed in parallel, then it would be faster to break the program into subtasks that be executed concurrently. This is referred to as concurrent programming. In concurrent programming, tasks are broken down into subtasks and executed concurrently. Their results are then collected by the parent task and composed into a final result. Concurent programming is an essential tool for improving the performance of software applications by utilizing compute and memory resources efficiently. Almost all languages offer ways to achieve concurrent execution of subtasks. In Java, this support is offered by the java.util.concurrent package constructs such as ExecutorService and Future.

The downside of concurrent programming is the loss of structure in code. Complexity is introduced due to breaking tasks into multiple subtasks. The relationship between the overall task and the subtasks is managed by the developer. This complexity can lead to programming errors. Developers need to account for a lot of corner cases. Failing to do so may introduce errors and unexpected behavior.

To address this issue, Java released a preview feature in JDK 21 that introduces support for structured concurrency. Structured concurrency allows writing concurrent code in a manner similar to single-threaded code. This removes the responsibility of subtask relationship management from the developer and is handled by the language instead. Structured concurrency also allows writing programs that are much simpler to read resulting in easier debugging and troubleshooting of issues.



```
Result compute() throws ExecutionException,
  InterruptedException {
    Future<Integer> firstFut = executor.submit(...);
    Future<Integer> secondFut = executor.submit(...);
    int first = firstFut.get();
    int second = secondFut.get();
    return new Result(first + second);
}
```

Figure 1: A Typical Concurrent Program

Problems with Unstructed Concurrency

There are several reasons behind the necessity for structured concurrency. Let us review the shortcomings of traditional (unstructured) concurrency. The example in Figure 1 demonstrates a simple unstructured concurrent program. In this example the computation is distributed into two subtasks. The results of computation of these two subtasks are stored in firstFut and secondFut respectively. The compute() method awaits the completion of firstFut and secondFut before producing the final result by combining the values obtained from the two subtasks.

Thread Leaks

Consider the case where firstFut results in an error (the computation of firstFut throws an exception). In this case, firstFut.get() raises an exception causing the compute() method to terminate with the exception. However, secondFut might still be computing. Since the program terminated, there is no way to collect the result of secondFut. The program also does not try to terminate the computation of secondFut. This is known as thread leak.

If the compute() method itself is interrupted, the cancellation would not be propagated to firstFut and secondFut. firstFut and secondFut would continue execution even though the parent task was terminated. This is another example of thread leaks.

Thread leaks can cause many issues. Threads are an expensive resource and holding it unnecessarily wastes resources. The computation inside secondFut might be heavy and contribute further to wasted resources (since the result is never used). Worst case, the computation inside secondFut might produce side effects which leave the system in an unexpected state. Debugging these types of issues can be difficult.

Waste of Time and Resources
Let us consider the case where secondFut results in an error but firstFut is still processing. In this case, because we wait for the result of firstFut before secondFut, firstFut continues to execute even though secondFut fails. The program doesn't check the result of secondFut until firstFut terminates. Ideally however, the compute() method should throw an exception right away. This wastes a lot of time and resources since we could have cancelled firstFut after secondFut results in an error. Inefficient use of resources could affect system performance when compounded over many processes.

Lack of Structure
In all the cases discussed above, the core issue lies in the fact that the language does not have core understanding of tasks and subtasks spawned by them. This relationship is tracked explicitly by the developer of code. This lack of semantic understanding of task-subtask relationships causes handling of exceptional cases to be non-trivial and prone to error. The example in Figure

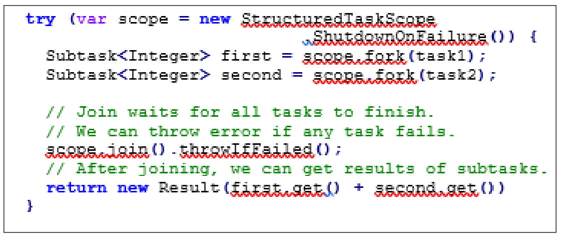
1 is a simple one, yet it still has several corner cases to explicitly handle. As the program complexity increases, handling corner cases becomes even tougher.

Poor Debugging Support
Thread dump and other observability tools do not show the relationship between tasks and subtasks. firstFut and secondFut show up on the call stacks of unrelated threads. This makes it very difficult to find out the state of system at the time the dump was captured.

Structure Concurrency
Now that we know of the issues with unstructured concurrency in Java, let us discuss Java support for structured concurrency and it solves these issues.

Structured Concurrency Principles
The main idea behind structured concurrency is that when subtasks are spawned from a task, they should all return to the same place after completion. Indirectly, this puts responsibility of managing subtask execution and result collection on the language runtime rather than the programmer. For this to be possible, the runtime should know about and keep track of task-subtask dependencies. The runtime can also manage the life cycle of subtasks and terminate them when appropriate.

StructuredTaskScope
The main class added for supporting structured concurrency is StructuredTaskScope. This is part of the java.util.concurrent package. Instead of executing subtasks in an ExecutorService, subtasks can be forked from the main task inside a scope defined by StructuredTaskScope class. The StructuredTaskScope API is ideally used with the try-with-resources statement.



```java
try (var scope = new StructuredTaskScope
                            .ShutdownOnFailure()) {
  Subtask<Integer> first = scope.fork(task1);
  Subtask<Integer> second = scope.fork(task2);

  // Join waits for all tasks to finish.
  // We can throw error if any task fails.
  scope.join().throwIfFailed();
  // After joining, we can get results of subtasks.
  return new Result(first.get() + second.get())
}
```

Figure 2: Structured Concurrency with StructuredTaskScope

Creating Subtasks
The fork() method can be used for forking subtasks. The input to this method is a Callable. The fork() method can only be called while the scope has not been shut down.

Task Completion
For collecting results, the join() method can be used. This method waits for all subtasks to finish execution, unless the task was interrupted or shutdown() method was called. There is also joinUntil() variant that times out after a specified duration.

Error Handling
The join() method throws exceptions for abnormal termination (normal termination is specified by shutdown policy). This simplifies error handling since developers only need to handle abnormal termination at one place.

Scope Shutdown
At the end of the try-with-resources statement, the scope is automatically shut down. There is also a shutdown() API that allows shutting down a scope without closing it. Shutting down a task scope prevents new subtasks (threads) from being started, cancels all unfinished subtasks, and invokes the join() method (for finishing execution).

Options for Shutdown Behavior
There are a couple of options for tweaking the behavior of shutting down a scope. ShutdownOnFailure and ShutdownOnSuccess are subclasses of the StructuredTaskScope class and offer custom shutdown policies. In addition, it also possible to implement your own shutdown policy. These options provide a lot of flexibility in the usage of structured concurrency.

Shutdown Policy: ShutdownOnFailure
This shutdown policy captures the exception that is raised by the first subtask that fails. After this, it shuts down the task scope so that all other subtasks are cancelled. This policy should be used in cases where it is critical for all subtasks to succeed.

Shutdown Policy: ShutdownOnSuccess
Conversely, the ShutdownOnSuccess policy waits for any subtask to complete and captures its result. One captured, it shuts down the task scope. This policy is useful when the result from one subtask is sufficient to proceed and other subtasks must be terminated when one succeeds.

Custom Shutdown Policies
Apart from the two default showdown policies discussed above, there is also an option for developers to specify custom shutdown policies. This is done by extending the StructuredTaskScope class and overriding the handleComplete() method.

Benefits of Structured Concurrency
Simpler Error Handling
The runtime handles subtask relationship and termination of subtasks. Developers are required to handle errors originating from the join() method. This is in contrast to error handling in unstructured concurrency where errors need to be handled by every subtask listener.

Cancellation Propagation
If the parent task is interrupted or a shutdown condition is met, all other subtasks are automatically cancelled. This prevents thread leaks and removes the need for manual cancellation handling.

Clarity
Code in structured concurrency looks very similar to single threaded (non-concurrent) code. This improves readability and makes it easier to understand code structure.

Observability
As opposed to unstructured concurrency, the thread dump shows relationships of tasks and subtasks. This makes it much easier to debug and troubleshoot issues.

Virtual Threads
The fork() method spawns new threads. For this reason, structured concurrency pairs very well with virtual threads, which introduce lightweight threads to the Java runtime. In fact, the default behavior of fork() is to use virtual threads.

Caveat of Preview Features
Structured concurrency is a preview feature introduced in Java 21. This means it is not available by default and is not suitable for production usage since it is experimental.

Conclusion
Unstructured concurrency has several drawbacks which can make code harder to read, write and debug. This not only adds a learning curve for developers but makes code more prone to errors. Most of these issues are solved by structured concurrency. In structured concurrency, code written is very similar to single threaded code, which makes it much simpler to read and write. Complexities of cancellation propagation are handled by the runtime and error handling is simplified. Structured concurrency is still a preview feature, so it is good to experiment with but not for deploying in production. Hopefully, this preview feature will get a full launch in the future [1-4].

References
1. Ron Pressler, Alan Bateman (2023) JEP 453: Structured Concurrency (Preview). OpenJDK https://openjdk.org/jeps/453.
2. (2023) Structured Concurrency. Core Libraries https://docs.oracle.com/en/java/javase/21/core/structured-concurrency.html.
3. Yuri Luiz de Oliveira (2023) Structured concurrency with Java 21 in 4 steps. Medium https://medium.com/wearewaes/structured-concurrency-with-java-21-in-4-steps-37e72997ed2a.
4. Nathaniel J Smith (2018) Notes on structured concurrency, or: Go statement considered harmful. njs blog https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/.