Structured concurrency

Structured concurrency is a programming technique that helps developers write concurrent programs in a more structured and easier-to-reason-about way. It involves organizing the concurrent parts of a program into a hierarchy of tasks, rather than using traditional threads and locks. This can make it easier to reason about the behavior of the program, and can also help prevent common concurrency errors such as race conditions and deadlocks.

The **structured concurrency** feature (JEP-428) aims to simplify Java concurrent programs by treating multiple tasks running in different threads (forked from the same parent thread) as a single unit of work. Treating all such child threads as a single unit will help in managing all threads as a unit; thus, canceling and error handling can be done more reliably.

The reliability in error handling and task cancellations will eliminate common risks such as thread leaks and cancellation delays.

1. Issues with Traditional Concurrency

1.1. Thread Leaks

In traditional multi-threaded programming (**unstructured concurrency**), if an application has to perform a complex task, it breaks the program into multiple smaller and independent units of sub-tasks. Then application submits all the tasks to *ThreadPoolExecutor*, generally with an *ExecutorService* that runs all tasks and sub-tasks.

In such a programming model, all the child tasks run concurrently, so each can succeed or fail independently. There is no support in the API to cancel all related subtasks if one of them fails. The application has no control over the subtasks and must wait for all of them to finish before returning the result of the parent task. This waiting is a waste of resources and decreases the application's performance.

For example, if a task has to fetch the details of an account and it requires fetching details from multiple sources such as account details, linked accounts, user's demographic data etc., then pseudo code a concurrent request processing will look like this:

```
Response fetch(Long id) throws ExecutionException, InterruptedException {
    Future<AccountDetails>  accountDetailsFuture  = es.submit(() ->
getAccountDetails(id));
    Future<LinkedAccounts> linkedAccountsFuture = es.submit(() ->
fetchLinkedAccounts(id));
    Future<DemographicData> userDetailsFuture = es.submit(() -> fetchUserDetails(id));


    AccountDetails accountDetails  = accountDetailsFuture.get();
    LinkedAccounts linkedAccounts  = linkedAccountsFuture.get();
    DemographicData userDetails    = userDetailsFuture.get();


    return new Response(accountDetails, linkedAccounts, userDetails);}
```

In the above example, all three threads execute independently.

- Suppose if there is an error in fetching the linked accounts then *fetch()* will return an error response. But the other two threads

will continue running in the background. This is a case of the thread leak.

- Similarly, if the user cancels the request from the front end and the *fetch()* is interrupted, all three threads will continue running in the background.

Though canceling the subtasks is programmatically possible, there is no straightforward way to do it, and there are chances of error.

1.2. Unrelated Thread Dumps and Diagnosis

In the previous example, if there is an error in the *fetch()* API then it is hard to analyze the thread dumps because these are running in 3 different threads. Making a relationship between the information in 3 threads is very difficult because there is no relationship between these threads at the API level.

When the call stack defines the task-subtask hierarchy, such as in sequential method executions, we get the parent-child relationship, which flows into error propagation.

Ideally, the task relationship should reflect at the API level to control child threads' execution and debug when necessary. This would allow a child to report a result or exception only to its parent — the unique task that owns all the subtasks — which, then, could implicitly cancel the remaining subtasks.

2. Structured Concurrency

2.1. Basic Concept

In structured multi-threaded code, if a task splits into concurrent subtasks, they all return to the same place i.e., the task's code block. This way the lifetime of a concurrent subtask is confined to that syntactic block.

In this approach, **subtasks work on behalf of a task that awaits their results and monitors them for failures**. At run time, structured concurrency builds a *tree-shaped hierarchy of tasks*, with sibling subtasks being owned by the same parent task. This tree can be viewed as the concurrent counterpart to the call stack of a single thread with multiple method calls.

2.2. Implementing with StructuredTaskScope

The `StructuredTaskScope` is a basic API for structured concurrency that supports cases where a task splits into several concurrent sub-tasks, to be executed in their own threads.

It enforces that the sub-tasks must complete before the main task continues. It ensures that the lifetime of a concurrent operation is confined by a syntax block.

Let us rewrite the previous example with *StructuredTaskScope* API. Note that the `fork()` method starts a virtual thread to execute a task, the `join()` method waits for all threads to finish, and the `close()` method closes the task scope.

The *StructuredTaskScope* class implements *AutoCloseable* interface so if we use the *try-with-resources* block then `close()` will be invoked automatically after the parent thread finishes execution.

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()()) {


    Future<AccountDetails> accountDetailsFuture = scope.fork(() -> getAccountDetails(id));
```

```
    Future<LinkedAccounts> linkedAccountsFuture = scope.fork(() ->
fetchLinkedAccounts(id));
    Future<DemographicData> userDetailsFuture = scope.fork(() -> fetchUserDetails(id));

    scope.join();       // Join all subtasks
    scope.throwIfFailed(e -> new WebApplicationException(e));

    //The subtasks have completed by now so process the result
    return new Response(accountDetailsFuture.resultNow(),
                linkedAccountsFuture.resultNow(),
                userDetailsFuture.resultNow());}
```

This solution resolves all the problems with unstructured concurrency as noted down in the first section.

3. Structured Concurrency and Virtual Threads

Virtual threads are JVM-managed lightweight threads for writing high throughput concurrent applications. As virtual threads are inexpensive compared to traditional OS threads, structured concurrency takes advantage of them for forking all new threads.

In addition to being plentiful, virtual threads are cheap enough to represent any concurrent unit of behavior, even behavior that involves I/O. Behind the scenes, the task-subtask relationship is maintained by associating each virtual thread with its unique owner, so it knows its hierarchy, similar to how a frame in the call stack knows its unique caller.

4. Conclusion

When combined with virtual threads, the structured concurrency promises long-awaited and much-needed features to Java that are already present in other programming languages (e.g., goroutines in Go and processes in Erlang). It will help in writing more complex and concurrent applications with excellent reliability and fewer thread leaks.

Such applications will be easier to debug and profile when errors occur.

Happy Learning !!