JAVA FULL STACK DEVELOPMENT PROGRAM

Week 8 Day 5: Asynchronous

OUTLINE

- Asynchronous
  - Callable - Future
  - Semephores
  - ExecutorService
  - @Async
    - CompletableFuture

SCENARIO

- In enterprise application, it is very common to talk about the performance of the application. Think about the following scenario:
  - In order to prepare the home page, we have to make 4 independent calls
    - UserInfo — it takes 2 seconds to get the result
    - AccountInfo — it takes 4 seconds to get the result
    - BillingInfo — it takes 1 seconds to get the result
    - UsageInfo — it takes 5 seconds to get the result
- Now we are calling the four services in sequence, and ignore the time costing for combine all results, how long will it take for UI to the response for home page?
- How can we improve the performance in this scenario?

Callable - Future

- Callables are functional interfaces just like runnables but instead of being void they return a value.

```java
ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2);

Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep( timeout: 3);
        return 123;
    } catch (InterruptedException e) {
        throw new IllegalStateException("Task interrupted");
    }
};

Future<Integer> response = executorService.submit(task);

System.out.println("Future done? " + response.isDone());

Integer data = response.get();
System.out.println("Data: " + data);

executorService.shutdown();
```

Executor Service

• Say I want to limit the number of thread that can be created in one application - **Thread Pool**

• The concurrency API introduces the concept of an ExecutorService as a higher level replacement for working with threads directly

• Executors are capable of running asynchronous tasks and typically manage a pool of threads, so we don’t have to create new thread manually

Executor Service

- Submit() - Submit method can take both Runnable and callable interface
- Shutdown() - Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. Invocation has no additional effect if already shut down
- ShutdownNow() -Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution
- awaitTermination() - block until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first

Executor Service

- Shutdown - will just tell the executor service that it can't accept new tasks, but the already submitted tasks continue to run

- ShutdownNow - will do the same AND will try to cancel the already submitted tasks by interrupting the relevant threads.

Executor Service

- InvokeAll - Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first.

- InvokeAny - Executes the given tasks, returning the result of one that has completed successfully(i.e. without throwing an exception), if any do before the given timeout elapses.

COMPLETABLE FUTURE

- CompletableFuture is used for asynchronous programming in Java
- Asynchronous programming is a means of writing *non-blocking* code by running a task on a separate thread than the main application thread and notifying the main thread about its progress, completion or failure

COMPLETABLE FUTURE

- **Creating completable future**
- CompletableFuture.runAsync(()->{}): returns void
- CompletableFuture.supplyAsync(()->{}): returns CompletableFuture<U>

- **CompletableFuture chaining callbacks**
- thenAccept(v->do somthing...): returns CompletableFuture<void>
- thenApply(v->return something...): returns CompletableFuture<U>

```
public static CompletableFuture<Void> runAsync(Runnable runnable)
public static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)
```

COMPLETABLE FUTURE

```
CompletableFuture<String> completableFuture = new CompletableFuture<String>();
```

- All the clients who want to get the result of this CompletableFuture can call
- completableFuture.get()
- completableFuture.join()
- The main difference between get() and join() is that get() will throw InterruptedException and ExecutionException as checked exception, while join() only throw unchecked exception.

COMPLETABLE FUTURE

```
static CompletableFuture<Void>   allOf(CompletableFuture<?>... cfs)
static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)
```

- CompletableFuture.allOf is used in scenarios when you have a List of independent futures that you want to run in parallel and do something after all of them are complete

- CompletableFuture.anyOf() as the name suggests, returns a new CompletableFuture which is completed when any of the given CompletableFutures complete, with the same result

Semaphores

- In addition to locks, the Concurrency API also supports counting Semaphores
- Whereas locks usually grant exclusive access to variables or resources, a semaphore is capable of maintaining whole sets of permits
- This is useful when you have to limit the amount of concurrent access to certain parts of your application
- If counter of the semaphore is 0, the semaphore puts the thread to sleep until the counter is greater than 0
(scenario: If there are 10 thread running in the application, how can you limit the access to a resource to 5 thread?)

ASYNCHRONOUS

- For independent tasks, we can run them in different threads and combine the result until all threads finish.
- Two ways to implement the asynchronous process
  - @Async
  - Messaging

@ASYNC

- Annotating a method of a bean with @Async will make it execute in a separate thread
  - It must be applied to public methods only
    - The method needs to be public so that it can be proxied
    - Self-invocation – calling the async method from within the same class – won't work
      - It bypasses the proxy and calls the underlying method directly
- To get started, we need to add @EnableAsync to either the Starter or Configuration classes.
- We can use Completable Future as the return type of the method annotated with *@Async*

@ASYNC

- Just as we can configure the connection pool for Database Connections, there is a way for us to configure the thread pool.
- Executors — The *Executors* helper class contains several methods for creation of pre-configured thread pool instances for you
  - Those classes are a good place to start with – use it if you don't need to apply any custom fine-tuning.
- By default, Spring uses a *SimpleAsyncTaskExecutor* to actually run these methods asynchronously
- We can override the the default at
  - Application level — rarely used since we may have different executor for different usage
  - Method level

Any Questions?