Coding is Fun

Ideas to Life

FUNCTIONAL PROGRAMMING, JAVA

Asynchronous programming in java

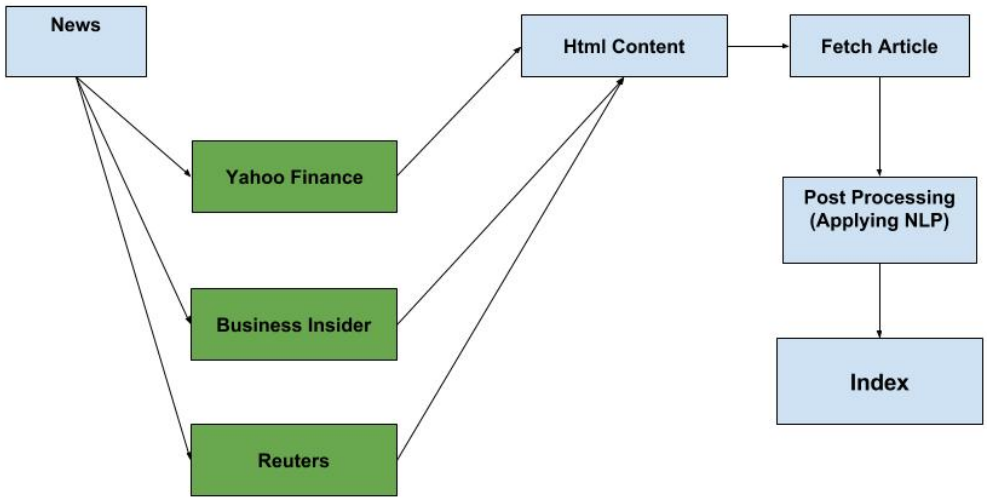
Date: August 9, 2016 Author: Kishore Karunakaran ☐ 0 Comments
Today I want to talk about Asynchronous programming, particularly how we can apply asynchronous programming using Java. Let start with simple question..

What is Asynchronous program ?

According to Wikipedia, Asynchronous programming is a means of parallel programming in which a unit of work runs separately from the main application thread and notifies the calling thread of its completion, failure or progress.

Let's understand with an example…

For instance, You have a product that brings a relevant information to the customers such as business news, stock information, etc. One of the module of this product is to scrap the news from various sources and understand it's content which mean you will be extracting various news articles, stock information from multiple sources, Yahoo, Business Insider, Reuters, etc.



Here we need to contact multiple services across the internet to get the relevant news for the customers. What you don't want to do is block your computations in order to get relevant information from the above services. For example, you shouldn't have to wait for data from Yahoo Finance to start processing the data coming from Business Insider. This represents the other side of the multitask-programming i.e. Parallel programming.

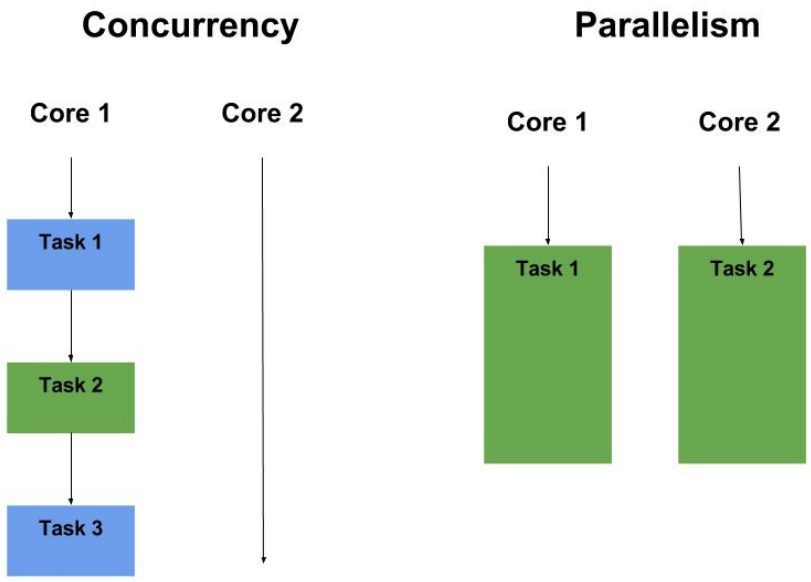
Conversely, when dealing with concurrency instead of parallelism, what you really want to achieve is to avoid blocking a thread and wasting its computational resources while waiting, potentially for quite a while, for a result from a remote service or from interrogating a database.

Difference between concurrency and parallelism

Concurrency is essentially applicable when we talk about two tasks or more. When an application is capable of executing two tasks virtually at same time, we call it concurrent application. Though here tasks run looks like simultaneously, but essentially they **may** not. They take advantage of CPU time-slicing feature of operating system where each task run part of its task and then go to waiting state. When first task is in waiting state, CPU is assigned to second task to complete it's part of task.

Parallelism does not require two tasks to exist. It literally physically run parts of tasks OR multiple tasks, at the same time using multi-core infrastructure of CPU, by assigning one core to each task or sub-task. Parallelism requires hardware with multiple processing units, essentially. In single core CPU, you may get concurrency but NOT parallelism.

A condition that arises when at least two threads are executing simultaneously.



If you like to know more about concurrency and parallelism, please refer following resources.

1. Concurrency vs Parallelism
2. Difference-between-concurrency-and-parallelism
3. Concurrency-is-not-Parallelism
4. From Concurrent to Parallel Brian Goetz(Java Architect)

Futures in Java

Java docs says, A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method **get when the computation has completed, blocking if necessary until it is ready**.

In asynchronous programming, main thread doesn't wait for any task to finished, rather it hand over the task to workers and move on. One way of asynchronous processing is using callback methods. Future is another way to write asynchronous code. By using Future you can write a method which does long computation but returns immediately. Those methods, instead of returning a result, return a Future object. You can later get the result by calling **Future.get()** method, which will return an object of type T, where T is what Future object is holding.

Implementing Scrapper Module using Future. The idea here is that you have a various sources in a text file. You need to process each sources(URL) where the processing involves extracting page-source, fetch the clean content, title and finally convert to Result object.

The above logic is implemented in **invokeCallable** method.

Future Limitations

Future interface provides methods to check if the asynchronous computation is complete (using the **isDone** method), to wait for its completion, and to retrieve its result. But these features aren't enough to let you write concise concurrent code. For example, it's difficult to express dependencies between results of a Future. In order to get result from future, we need to call **get** method which is blocking. What we need is

- Combining two asynchronous computations in one—both when they're independent and when the second depends on the result of the first.
- Reacting to a Future completion (that is, being notified when the completion happens and then having the ability to perform a further action using the result of the Future , instead of being blocked waiting for its result).
- Pro-grammatically completing a Future (that is, by manually providing the result of the asynchronous operation).

CompletionStage and CompletableFuture from Java 8

CompletionStage is an interface which abstracts units or blocks of computation which may or may not be asynchronous. It is important to realize that multiple CompletionStages, or in other words, units of works, can be piped together.

CompletionStage can abstract an asynchronous task and also you can pipe many asynchronous outcome in completion stage which lays the foundation of a reactive result processing which can have a valid use-case in virtually any area, from Gateways to Clients to Enterprise Apps to Cloud Solutions. Furthermore potentially, this reduces superfluous polling checks for the availability of result and/or blocking calls on futuristic results.

CompletableFuture is introduced in Java 8 which provides abstraction for async tasks in event driven programming. It s designated for executing long running operations (http requests, database queries, file operations or complicated computations).

This new and improved CompletableFuture has 2 main benefits:

- It can be explicitly completed by calling the complete() method without any synchronous wait. It allows values of any type to be available in the future with default return values, even if the computation didn't complete, using default / intermediate results.
- With tens of new methods, it also allows you to build a pipeline data process in a series of actions. You can find a number of patterns for CompletableFutures such as creating a CompletableFuture from a task, or building a CompletableFuture chain. The full list is available via Oracle's CompletableFuture documentation.

Let's implement above code using CompletableFuture

Let's look at the API's

static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)

Returns a new CompletableFuture that is asynchronously completed by a task running in the ForkJoinPool.commonPool() with the value obtained by calling the given Supplier.

static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)

Returns a new CompletableFuture that is asynchronously completed by a task running in the given executor with the value obtained by calling the given Supplier.

java.util.function.Supplier is a functional interface which accepts nothing and supplies an output. The supplyAsync() API expects that a result-producing task be wrapped in a Supplier instance and handed over to the supplyAsync() method, which would then return a CompletableFuture representing this task. This task would, by default, be executed with one of the threads from the standard java.util.concurrent.ForkJoinPool (public static ForkJoinPool commonPool()).

However, we can also provide custom thread pool by passing a java.util.concurrent.Executor instance and as such the Supplier tasks would be scheduled on threads from this Executor instance.

Similarly we can also supply Runnable instances

static CompletableFuture<Void> runAsync(Runnable runnable)

static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)

Chaining multiple CompletableFutures

The flexibility of asynchronous task processing actually comes by the virtue of chaining multiple tasks in a particular order, such that (asynchronous) completion of one CompletableFuture Task might fire asynchronous execution of another separate task which helps us to pipeline many asynchronous tasks.

You can see there how we can pipe init, Process A, Process B methods in a Asynchronous approach. This is just an sample there are plenty more methods where we can use to pipe the tasks, also please check then **whenComplete** method.

Handling Exceptions

CompletableFuture API provides the flexibility of handling situations when one asynchronous task completes exceptionally. Basically method exceptionally() comes handy for this purpose.

Example

Summarizing Completable Future API

Methods	Takes	Returns
thenApply(Async)	**Function**	**CompletionStage holding the result of the Function**
thenAccept(Async)	**Consumer**	**CompletionStage<Void>**
thenRun(Async)	**Runnable**	**CompletionStage<Void>**

Based on both Stages

thenCombine(Async)	BiFunction	CompletionStage holding the result of the Function
thenAcceptBoth(Async)	BiConsumer	CompletionStage<Void>
runAfterBoth(Async)	Runnable	CompletionStage<Void>

Based on Either one of the Stages

applyToEither(Async)	Function	CompletionStage holding the result of the Function
acceptEither(Async)	Consumer	CompletionStage<Void>
runAfterEither(Async)	Runnable	CompletionStage<Void>

Complete Code in Github

Conclusion

The flexibility of chaining multiple **CompletableFutures** such that the completion of one triggers execution of another CompletableFuture this opens up the paradigm of reactive programming in Java. Now there is no blocking call like **Future.get()** to retrieve the result of the future Task.

Resources

1. Gentle Introduction to Completable Future
2. How to use Completable Future
3. JVM concurrency: Java 8 concurrency basics
4. Back to the CompletableFuture
5. Functional-Style-Callbacks-Using-CompletableFuture