Building Better Concurrency with Kotlin Coroutines

Wenqing Ge
231502002@smail.nju.edu.cn

Abstract

The need for concurrent and synchronized operations gives rise to many solutions, like threads, callback functions, generators, fibers, etc. However most of these traditional solutions bear several problems that could lead to error-proneness, memory leaks or over-complicated platitude. As a modern language, Kotlin provides a framework for coroutine that is able to solve many traditional problems.

CCS Concepts: • **Software and its engineering → Coroutines**.

Keywords: Kotlin, Coroutine

ACM Reference Format:
Wenqing Ge. 2023. Building Better Concurrency with Kotlin Coroutines. In *Proceedings of null (PLP Workshop '23)*. ACM, New York, NY, USA, 3 pages. https://doi.org/114514.1919810

1 Introduction

In real-world applications, concurrent and parallel operations are utilized to maximize efficiency or to reduce blockage. Concepts like threads, callback functions, primitive coroutine libraries (like generators or fibers) can be used by a programmer to implement such requirements. However, these inventions all do not meet a developer's requirement for a "good" solution: lightweight, intuitive and memory-safe. Threads have the problem of being cost-heavy, callback functions are not intuitive enough and primitive coroutine implementations are prone to memory leak. Aimed to create better concurrency, Kotlin includes Kotlin Coroutine which is a revolutionary tool to implement concurrent operations. In the following sections, we will talk about how "good" Kotlin coroutine is compared to the traditional solution, namely in these three factors: light-weight, intuitive, memory-safe, by analysing the structure and feature of it.

PLP Workshop '23, December 23, 2023, Nanjing, Jiangsu
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/114514.1919810

2 Light-weight

Threads (or even subprocesses) are the classic solution to concurrent operations. However they are "cost-heavy" as they require a lot of resources to boot up or to maintain. The source of the problem lies in the fact that they are on system levels. Traditional solutions to cost-heaviness require using thread pools or other techniques which may result in extra coding or efficiency-impeding platitude.

However, Kotlin coroutines are light-weight. Because in Kotlin, coroutines are eventually compiled to Java Bytecode as a normal function with only few extra seasonings. Executed as normal functions means that they require fewer resources than threads. This increases the number of concurrent jobs that can be executed at the same time. Consider the code below. If it were threads, we might receive a *OutOfMemoryError* at any moment. But as we are using light-weight coroutines, we are totally fine.

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {
    repeat(50_000) { // launch a lot of coroutines
        launch {
            delay(5000L)
            print(".")
        }
    }
}
```

Listing 1. Coroutines are light-weight![1]

3 Intuitive

Another solution related to concurrency commonly used in Javascript or related languages is by utilizing callback functions. However, this breaks the sequential order of execution that is intuitive to programmers and users alike. This has many disadvantages. First of all, it is surely anti-intuitive which might lead to error-proneness. Secondly, it is awkward for developers to combine parallel and sequential operations. Thirdly, in a complex scenario, too many stacked callbacks may result in "Callback Hell" which leads to low readability and maintainability.

But Kotlin coroutines are intuitive on the contrary. In Kotlin coroutines, code is by default executed in the same sequential order as in normal functions, even in separate functions. The code is also considered to be in the same scope instead of different functions in callbacks so that fewer scoping problems of variables may happen.

Doing so has many benefits. First of all, this is easier for new learners to get started as it does not require knowledge of a completely novel design method. Secondly, this greatly helps integrating both sequential and concurrent operations as well as reflecting the programmer's intuitive logic.

Have a look at the example below and try to predict the output[1]:

```
// Sequentially executes doWorld
fun main() = runBlocking {
    doWorld()
    //doWorld() is created by coroutineScope which will not finish until all children are finished
    println("Done")
}

// Function will not end unless all children are finished
suspend fun doWorld() = coroutineScope {
    // this: CoroutineScope
    launch { //a child
        delay(2000L)
        println("World␣2")
    }
    launch { //another child
        delay(1000L)
        println("World␣1")
    }
    println("Hello")
}
```

Listing 2. Basic Concurrency[1]

The program above will display the following text when finished:

```
Hello
World 1
World 2
Done
```

This is exactly what we have expected. Moreover, the program clearly shows the flow of execution without requiring much explanation. However, to write this code using callback functions we might need counters and pass callback function as arguments like in the equivalent code below:

```
var counter=0
function checkCallback(callback){
    counter++
    if(counter==2){
        callback()
    }
}
function doWorld(callback){
    setTimeout(() => {
        console.log("World␣2")
        checkCallback(callback)
    },2000) //simulating child1
    setTimeout(()=>{
        console.log("World␣1")
        checkCallback(callback)
    },1000) //simulating child2
    console.log("Hello")
}

doWorld(() => {
    console.log("Done")
})
```

[1] *suspend* is a keyword in Kotlin to indicate that a function is used as/in a coroutine

Listing 3. Equivalent Code in Javascript Callbacks

As you can see not only does Javascript code require extra functions, the logic of it is also harder to trace, which is not considered "good" enough.

4 Memory-safe

Memory leaks or unnecessary extra resource consuming can happen when an operation is cancelled (e.g. user has closed the application or decide to undo the last move) but the underlying threads or coroutines are not terminated properly. However, terminating such coroutines in primitive libraries (like in Python generators) can be more easily forgotten by programmers which result in memory leaks.

Kotlin, on the other hand, implements modern and automatic controls for such task which is called *Structured Concurrency* that can prevent memory leaks. As shown in the structure illustration Figure 1, In Kotlin, multiple coroutines can form a tree structure in a parent-child style relationship. *CoroutineContext* and *CoroutineDispatcher* will be inherited in this way by default. A parent coroutine is not considered as finished unless all children coroutines are finished. When a parent coroutine is cancelled or timeout-ed, all children coroutines are also automatically recursively cancelled or timeout-ed. As this is a automatic process, developers do not to put much efforts to prevent potential leaks.

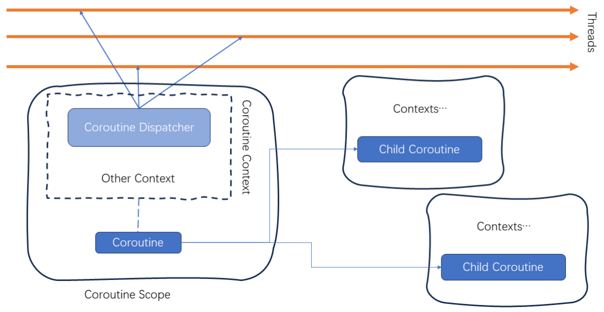

Figure 1. A basic runtime structure of Kotlin coroutines

5 Conclusion

In conclusion, Kotlin coroutines provide a light-weight, memory-safe, intuitive framework for developers, which should make it a more favorable choice to developers using threads, callback functions or traditional coroutine implementations.

References

[1] Jetbrains. 2023. *Coroutines basics.* Retrieved Nov 20, 2023 from https://kotlinlang.org/docs/coroutines-basics.html