Introduction to Kotlin Coroutines

Coroutines for asynchronous programming

Satya Pavan Kantamani
Aug 9, 2020 · 8 min read ★

Introduction

In most of the programming languages, it's quite a common thing of doing synchronous tasks like hitting an API and waiting for the result to process the next steps, waiting for fetching data from the database, etc.

We basically started handling this using the **callback** mechanism. We do something like hitting an API and wait for the callback to get invoked where we process the result.

A **callback** is a function that will be executed after another function has finished executing. And if there are a series of things to be done synchronously then we will fall into **callback hell** that can lead us to ambiguity in understanding the code. It depends on the number of steps and logic we have in our applications.

To avoid this callback hell and with the difficulty of managing multiple threads at a time, we adopted Rx Java in Android where the code looks cleaner and easily understood.

Also, exception handling and disposing of things can be handled in a good way. The main problem with Rx is like exploring its list of operators in-depth while performing complicated operations and apply them correctly.

> *It is like learning another programming language called Rx Java to do simple synchronous programming spending more time.*

What are Coroutines

This is the place where coroutines come into the play. Simply saying coroutines are nothing but **light-weight threads**. Coroutines provide us an easy way to do synchronous and asynchronous programming.

Coroutines were added to Kotlin in version 1.1 and are based on established concepts from other programming languages. `kotlinx.coroutines` is a library for coroutines developed by JetBrains.

From Kotlin docs:

> *One can think of a coroutine as a light-weight thread. Like threads, coroutines can run in parallel, wait for each other and communicate. The biggest difference is that coroutines are very cheap, almost free: we can create thousands of them, and pay very little in terms of performance. True threads, on the other hand, are expensive to start and keep around. A thousand threads can be a serious challenge for a modern machine.*

Why do we need Coroutines

As earlier discussed in the intro section it helps us to get rid of callback hell and using Rx with simple structures replaced to understand. Let’s check this with an example

Let’s take the example of a user shopping online. So when a user clicks on a product we need to fetch the data and show that to the user. And basically we write the following methods to do that

```kotlin

fun fetchItemDetails(itemId: Int): Item {
    // make a network call to fetch the data providing required things
    // return the item fetched
}

fun displayItemDetails(item: Item) {
    // display item details to user
}

fun onItemClicked() {
    val item = fetchItemDetails(1)
    displayItemDetails(item)
}

data class Item(
    @SerializedName("item_name")
    val itemName: String,
    @SerializedName("image_url")
    val imageURL: String?
)
```

Now let’s check how we do the above stuff using **callbacks**. We do the fetchItemDetails task on the background thread and we pass the result through the callback.

```kotlin

fun fetchItemDetails(id:Int,item:(Item)->Unit){
    //Makes API request for item details and  invokes the callback when needed
    // return
}

fun onItemClicked() {
    fetchItemDetails(1) { item ->
        displayItemDetails(item)
    }
}
```

corcallbk.kt hosted with ❤ by GitHub view raw

Now let’s check the same thing with Rx. Here we use the scheduler and observe on to specify the threads where the work needs to be done

```
     fun fetchItemDetails(id: Int): Single<Item> {
          //Makes API request for item details
          // return
      }

      fun onItemClicked() {
          fetchItemDetails(1)
              .subscribeOn(Schedulers.io())
              .observeOn(AndroidSchedulers.mainThread())
              .subscribe({ item ->
                  displayItemDetails(item)
              }, {
                  //Handle the exception
              })

      }
```

corrx.kt hosted with ♡ by GitHub view raw

Now finally let's check the code with Coroutines using suspend functions

```
     suspend fun fetchItemDetails(id: Int): Item {
        // Makes API hit and suspends
        //return item when received
      }

      suspend fun onItemClicked() {
         val item = fetchItemDetails(1)
          displayItemDetails(item)
      }

      fun displayItemDetails(item: Item) {
        //display the data to user
      }
```

corsus.kt hosted with ♡ by GitHub view raw

Isn’t that easy and cleaner? The above example has only one API request if there are N number of requests, just imagine the code with callbacks and Rx which will be a mess and confusing. Now let’s move to understand suspend functions

Suspend functions

The suspend functions are not any special kind of functions they are just normal functions appended with the **suspend** modifier to have the superpower of suspending.

suspend — is an indication saying that the method associated with this modifier is synchronous and will not be returned immediately. The method associated with it will be suspended for a while and then return the result when available.

If there are multiple suspend functions one called from the other it's nothing but just nesting normal function calls with just the suspend attached to it that specifies that one method needs to wait until the inner method execution is done and the result is available.

Suspend functions won't block the main thread which means when you call a suspend function on the main thread that function gets suspended and performs its work on any other worker thread and once it's done it resumes with the result so we can consume the result on the main thread.

Scopes in Coroutines

A CoroutineScope is an interface in the kotlinx.coroutines package that defines the scope of any coroutine that we create using `launch` or `async` or coroutine builders. The running coroutines can be canceled by calling `scope.cancel()` at any point in time.

Each coroutine created has its own instance of **CoroutineContext** interface. A context is nothing but a set of elements and we can get the current coroutine context by using the `coroutineContext` property. The coroutine context is a set of various elements. The main elements are the **Job** of the coroutine and its dispatcher.

Job — A job can be used as a handle to coroutine where we can track the wait time and other info related to the coroutine.

Dispatchers — It is used to specify which thread a coroutine uses for its execution. The coroutine builders, accept an optional CoroutineContext parameter that can be used to explicitly specify the dispatcher for the new coroutine and other context elements.

Let's have a simple look at the following example

```kotlin
private val job = Job()
val scope = CoroutineScope(job + Dispatchers.Main)

private fun fetchItemDetails(id:Int){

    scope.launch() {
        //do network operation and fetch data
        //do the UI update as mentioned with dispatcher
        //that is confined to the Main thread operating with UI objects
    }
}

override fun onDestroy() {
    super.onDestroy()
    job.cancel()
}
```

samplcrcntxt.kt hosted with ♡ by GitHub view raw

job — we created a new job instance and in the onDestroy method we cancel the job.

scope — we created the scope object with job instance and the required dispatcher thread.

launch — is the fire & forget coroutine build which we will see below. It's used to perform our action

Dispatchers

In Android, we mainly have three dispatchers

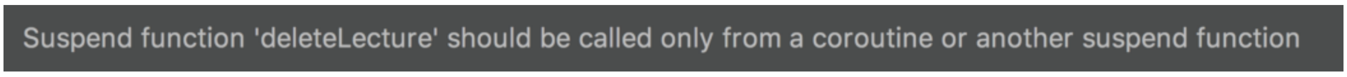
Dispatchers.Main: A coroutine dispatcher that is confined to the **Main thread** operating with UI objects. This dispatcher can be used either directly or via the MainScope factory. Usually, such dispatcher is single-threaded.
Access to this property may throw **IllegalStateException** if no main thread dispatchers are present in the classpath.

Dispatchers.Default: The default CoroutineDispatcher that is used by all coroutine builders like launch, async, etc if no dispatcher nor any other ContinuationInterceptor is specified in their context. It is Optimized for CPU intensive work off the main thread

Dispatchers.IO: The CoroutineDispatcher is designed for offloading blocking IO tasks to a shared pool of threads and networking operations. Additional threads in this pool are created and are shutdown on demand.

Coroutine builders

We can call suspend functions from a coroutine or another suspend function. When we try to call a suspend function from a non-suspend function the IDE throws an error saying:



Suspend function 'deleteLecture' should be called only from a coroutine or another suspend function

This is because the internal function is suspendable and waits until the result is available but the top-level function is a regular function that has no superpower of suspending. So here we need a connection between regular functions and suspending functions. This connection can be established using functions called **coroutine builders**.

> *Coroutine builders are simple functions that can create a coroutine and act as a bridge between the normal world and the suspending world.*

Let's explore a few coroutine builders. These coroutine builders are mainly called on scopes.

launch

The **launch{}** is a regular function in the library so we can invoke it from anywhere from the normal world. The launch function creates a coroutine and returns immediately however the work continues in the background thread pool. **It fires and forgets the coroutine**.

Let's check the syntax of the launch function.

```kotlin
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job {
    .......
    return coroutine
}
```

launchvor.kt hosted with ♡ by GitHub view raw

However, it takes a suspend functions as an argument and creates a coroutine. It also returns the object call Job. The launch is not plainly fire and forget but it doesn’t return any general result.

In Android mostly as soon as the result is available we update the UI without any checks as following

```
fun sum(){
    GlobalScope.launch {
       val item = fetchItemDetails(1)
        diaplayItemDetails(item)
    }
}
suspend fun fetchItemDetails(id:Int){
    //Hit API to fetch Data
}
```

launchcr.kt hosted with ♡ by GitHub view raw

As launch creates a coroutine that runs in the background if we update the UI it leads to a crash, we need to update the UI from the main thread. For this to happen we need to pass the context to launch to specify not just only use the background threads for execution use this when required so it takes care of dispatching the execution to the specified thread.

We need to use Dispatchers to specify the coroutine builders where to perform the task.

async /await

The **async{}** is another coroutine builder that takes a block of code and executes asynchronous tasks using suspended functions and returns the **Deferred** as a result. We can call `await` on this deferred value to wait and get the result. The async functions are concurrent functions.

The running coroutine is cancelled when the resulting deferred is cancelled by calling Job.cancel. The syntax is:

```kotlin
public fun <T> CoroutineScope.async(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
): Deferred<T> {
    return coroutine
}
```

asyncycor.kt hosted with ♡ by GitHub view raw

A simple example:

```kotlin
      scope.launch {
            val itemOne = async(Dispatchers.IO) { fetchItem1() }
            val itemTwo = async(Dispatchers.IO) { fetchItem2() }
            displayitems(itemOne.await(), itemTwo.await())
        }
```

asyncg.kt hosted with ♡ by GitHub view raw

It launches a coroutine and performs both the network calls asynchronously and waits for the result of items and then calls displayItems method.

runBlocking

The **runBlocking** is a normal function that can be invoked from any normal functions and not to be used from a coroutine. It actually blocks the main thread if the context is not specified until the coroutine finishes the execution. It runs the coroutine in the context on the thread it is invoked. Let’s check the syntax.

```kotlin
fun <T> runBlocking(
    context: CoroutineContext = EmptyCoroutineContext,
    block: suspend CoroutineScope.() -> T
): T (source)
```

runBlockingCor.kt hosted with ♡ by GitHub view raw

If this blocked thread is interrupted then the coroutine job is canceled and this `runBlocking` invocation throws **InterruptedException**.

Let's check a basic example

```
        runBlocking {
            println("runBlocking started : ${Thread.currentThread().name}")
            delay(1000)
        }

        runBlocking(CoroutineName("some-custom-name")) {
            println("runBlocking started : ${Thread.currentThread().name}")
            delay(2000)
        }
```

runBlockingsam.kt hosted with ♡ by GitHub view raw

The output of threads are

```
runBlocking started : main
runBlocking started : some-custom-name
```

> *delay is similar to Thread.sleep used blocking thread for specified amount of time*

Summary

In this post, we have understood what is a coroutine and it's basic usage with jobs, dispatchers & coroutine builders. This post wast to just provide an overview of the concept. We will learn more about these coroutine builders, scopes in my upcoming posts.

Check out more on Kotlin:

- Kotlin Guide for Beginners — Explains the basics of variable declarations & Why to learn Kotlin
- Kotlin Advanced Programming — This post is regarding basics related to functions & types of functions in Kotlin

- Kotlin Advanced Programming Part 2 — This post is regarding Functional Programming in Kotlin
- Kotlin Advanced Programming Part 3 — This post is regarding scope functions in Kotlin
- Kotlin Advanced Programming Part 4 — This post is regarding inline functions & reified types in Kotlin
- Kotlin Delegates — This post explains about inline functions & reified types in Kotlin
- Kotlin Sealed Classes — This post is regarding sealed classes which are an extension of enums.

References

KotlinConf 2017 — Introduction to Coroutines by Roman Elizarov

Coroutine basics

Please let me know your suggestions and comments.

You can find me on **Medium** and **LinkedIn** …

Thanks for reading…

Android Programming Kotlin Android App Development AndroidDev