Kotlin Coroutines(Part — 3): Coroutine Context

What does a CoroutineScope consist of? How to switch the context of a coroutine?

Aditi Katiyar · Follow
Published in DeHaat · 4 min read · Sep 6, 2021

Let us try to understand *CoroutineScope* in depth. The documentation of *CoroutineScope* class reads as follows:

> "Creates a CoroutineScope that wraps the given coroutine context.
> If the given context does not contain a Job element,
> ..."

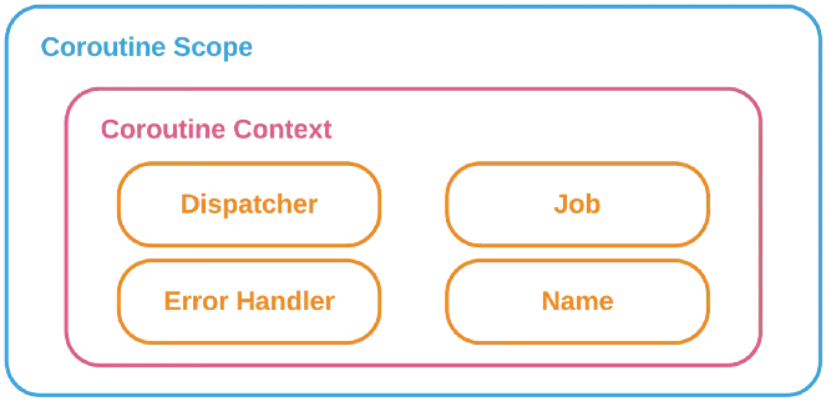
For now, we are only going to focus on the term *coroutine context*.

What is a coroutine context?

Coroutines always execute in some context represented by a value of the CoroutineContext type.

A 'CoroutineContext' is defined in the scope in which we start the coroutine. It is a collection of several 'context' elements -

1. Dispatcher(which thread to run on)
2. Job(control the lifecycle of coroutine)
3. error handler(handle errors when a coroutine throws an exception)
4. name(name of the coroutine)



Dispatcher, Job, Error Handler & Name are the context elements of a coroutine

The 'withContext()' function

It is a suspending function that is used to shift the execution of the current coroutine on a new *CoroutineContext*, in most cases, a different dispatcher.

```
14  fun fetchAndShowImage(imageView: ImageView) {
15      CoroutineScope(Dispatchers.IO).launch {  this: CoroutineScope
16          val imageFileUrl = apiService.getImageUrl()
17          withContext(Dispatchers.Main) {  this: CoroutineScope
18              loadImage(imageView, imageFileUrl)
19          }
20      }
21  }
22
23  private fun loadImage(imageView: ImageView, imageFileUrl: String)
24      // use Glide, Picasso, etc. load image in imageView
25  }
```

suspend function

We have launched a coroutine on the background thread(*Dispatchers.IO*). The function *apiService.getImage()* executes on the background thread. After we

get the image url, we want to show the image file in an ImageView. So we change the context(or the underlying thread) of our coroutine with the help of *withContext(Dispatchers.Main)*.

Note that *withContext()* is a suspending function, which means that it will make the caller coroutine **wait** until the code within *withContext()* executes.

Let's look at a few coroutine scopes and their context elements provided by the kotlinx-coroutines library.

viewModelScope

viewModelScope is a coroutine scope bound to the lifetime of a ViewModel, i.e., a coroutine started in *viewModelScope* will be cancelled when the ViewModel clears.

Dispatcher: *viewModelScope* uses *Dispatchers.Main.immediate* as the dispatcher by default. It optimises performance by immediately starting execution on the Main thread.

Job: *viewModelScope* uses *SupervisorJob* as the job element, which means that if a coroutine in *viewModelScope* throws an exception, the other coroutines in the same scope will continue to run, while the one that caused the exception will terminate.

```
9  class DogsViewModel @Inject constructor(
10     private val repository: DogsRepository
11 ) : ViewModel() {
12
13     private val doggos = MutableLiveData<List<Dog>>()
14
15     fun loadDoggos() {
16         viewModelScope.launch { this: CoroutineScope
17             doggos.value = repository.fetchFromServer()
18         }
19     }
20 }
```

lifecycleScope

lifecycleScope is a coroutine scope bound to the lifetime of a LifecycleOwner(Activity or Fragment), i.e., a coroutine started in *lifecyleScope* will be cancelled when *onDestroy()* of Activity/Fragment is called.

Dispatcher: *lifecycleScope* uses *Dispatchers.Main.immediate* as the dispatcher by default(just like *viewModelScope*).

Job: It uses *SupervisorJob* as the job element.

```
30      private fun syncDatabase(dogs: List<Dog>) =
31          lifecycleScope.launch { this: CoroutineScope
32              viewModel.syncDatabase(dogs)
33              analytics.sendSyncEvent()
34          }
```

save list in local database and send analytics event after save is complete

lifecycleScope is useful for UI operations like animations, etc. There are various functions to create coroutines in this scope:

- launch{ }
- launchWhenCreated{ }: starts when lifecycle is at least in Lifecycle.State.CREATED state
- launchWhenStarted{ } : starts when lifecycle is at least in Lifecycle.State.STARTED state
- launchWhenResumed{ }: starts when lifecycle is at least in Lifecycle.State.RESUMED state

viewLifecycleOwner.lifecycleScope

In an Activity/Fragment, we have a view's lifecycle owner — *viewLifecycleOwner*. This controls the lifecycle of a view. Its lifecycle starts when *onCreateView()* is called and destroys when *onDestroyView()* is called.

Thus, the coroutine started in v*iewLifecycleOwner.lifecycleScope* will be cancelled when *onDestroyView()* is called. It has the same context elements as *lifecycleScope*. Use this scope when you think that running your coroutine beyond *onDestroyView()* might access your views and cause exceptions.

```
18      private fun fetchData() =
19          viewLifecycleOwner.lifecycleScope.launch { this: CoroutineScope
20              viewModel
21                  .fetchData()
22                  .collectLatest { it: List<Dog>
23                      // set List<Dog> in a recyclerview adapter
24                  }
25          }
26
```

collecting a flow in Fragment/Activity

creating our own scope

While creating a scope with *CoroutineScope*, we have to pass at least one context element. We can pass a dispatcher. We can also specify the 'Job' element and exception handler.

```
CoroutineScope(Dispatchers.IO).launch {
    // scope with dispatcher as IO
}

CoroutineScope(Dispatchers.IO + SupervisorJob()).launch {
    // scope with dispatcher as IO and job as SupervisorJob
}

val exceptionHandler = CoroutineExceptionHandler { coroutineContext, throwable ->
    println("caught exception $throwable in context $coroutineContext")
}
CoroutineScope(Dispatchers.IO + SupervisorJob() + exceptionHandler).launch {
    // scope with dispatcher as IO and job as SupervisorJob & an ExceptionHandler
}
```

customcoroutinescope.kt hosted with ❤️ by GitHub view raw

The scope of a coroutine decides its lifetime. When a scope ends, the coroutine also cancels if it is running, hence preventing leaks. The scopes provided by kotlinx-coroutines library start a coroutine on the Main thread by default. Its underlying context elements(like Dispatcher) can be changed

- Kotlin Coroutines(Part — 1): The Basics
- Kotlin Coroutines(Part — 2): The 'Suspend' Function
- Kotlin Coroutines(Part — 4): Cancellation
- Kotlin Coroutines(Part — 5): Exception Handling

Thanks for reading! If you liked it, please give it a clap! Keep Learning!

Engineering Kotlin Coroutines Android Kotlin Android App Development