Kotlin Coroutines(Part — 4): Cancellation

How and when to cancel coroutines?

Aditi Katiyar · Follow
Published in DeHaat · 3 min read · Sep 8, 2021

Photo by Jeffrey Czum from Pexels

Whenever we create a coroutine, we shall always remember to cancel it. Otherwise, it can cause memory leaks or unnecessary updates to UI.

When we create a coroutine, we get a ‘job’ object. A job can be cancelled as follows:

```kotlin
class LocationManager(
    private val repository: LocationRepositiry
) {

    private var job: Job? = null

    fun saveLatLng(lat: Float, lng: Float) {
        val job = CoroutineScope(Dispatchers.IO).launch {
            repository.storeLocation(lat, lng)
        }
    }

    fun onDestroy() {
        job?.cancel()
    }
}
```

viewmodelscope.kt hosted with ❤️ by GitHub view raw

When we cancel a job, it throws a **CancellationException**. Unlike other exceptions, it does not crash the program. It is a silent exception that moves up the job hierarchy, but no special action is taken. The coroutine is cancelled normally.

```kotlin
class LocationManager(
    private val repository: LocationRepositiry
) {
    private var job: Job? = null

    fun saveLatLng(lat: Float, lng: Float) {
        val job = CoroutineScope(Dispatchers.IO).launch {
            repository.storeLocation(lat, lng)
        }
        job.invokeOnCompletion { throwable ->
            if (throwable is CancellationException)
                println("Coroutine is Cancelled!")
        }
    }

    fun onDestroy() {
        job?.cancel()
    }
}
```

cancellation.kt hosted with ❤️ by GitHub view raw

Here, if the job is cancelled before the coroutine completes, then it will throw a CancellationException and prints ‘Coroutine is Cancelled!’.

Structured Concurrency

Coroutines follow a structured concurrency paradigm. It means that coroutines have cleaner entry & exit points, ensuring that a coroutine finishes only when its children finish. Such execution allows errors to be propagated to the parent(or root) coroutine and child coroutines do not leak out(or become orphaned).

There are 5 properties of structured concurrency which coroutines follow:

1. Every coroutine starts in a logical scope

Coroutine should have a limited lifetime, or in other words, every scope should have a lifecycle. We need to track when to cancel the coroutine if it is running. It is the same as collecting observables in RxJava and disposing of them.

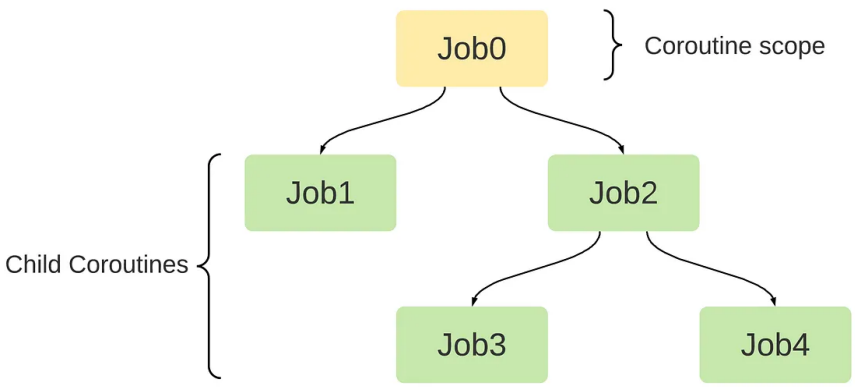
We can either create our coroutines in *viewModelScope* or *lifecycleScope*, or we can create our own scope and cancel it.

```
class LocationManager(
    private val repository: LocationRepositiry
) {

    private val scope = CoroutineScope(Dispatchers.IO)

    fun saveLatLng(lat: Float, lng: Float) {
        scope.launch {
            repository.storeLocation(lat, lng) // a suspend function
        }
    }

    fun saveName(name: String) {
        scope.launch {
            repository.saveName(name) // a suspend function
        }
    }

    fun onDestroy() {
        scope.cancel()
    }
}
```

cancelscope.kt hosted with ❤️ by GitHub view raw

Here, **cancelling the scope** will cancel all the coroutines running in this scope.

2. Coroutines started in the same scope form a job hierarchy



Jobs started in a coroutine become children of the parent's job. The child coroutines inherit all the context elements except the 'Job' object.

If *Job0* is started on the Main thread, then by default its child jobs — *Job1* & *Job2* will also start on the Main thread, and so on.

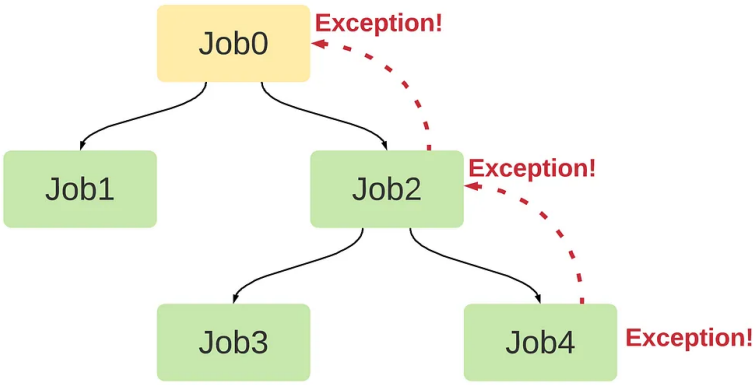
3. A parent job won't complete until all of its children have completed

In the above example, *Job0* will not be complete unless *Job1* & *Job2* are complete. And *Job2* will not be complete unless *Job3* & *Job4* are complete.

4. Cancelling a parent cancels its children recursively

If we cancel *Job0*, then it will cancel *Job1* & *Job2*. And *Job2* will further cancel *Job3* & *Job4*. However, cancelling a job does not cancel its sibling. e.g. cancelling *Job2* will not cancel *Job1*.

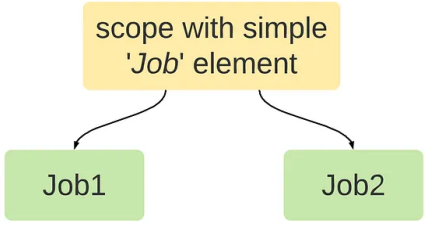
5. If a child coroutine fails, the exception propagates upwards in the job hierarchy, and depending on the parent job type, either all siblings are cancelled or none.



If an exception occurs at *Job4*, then it is propagated to its parent, *Job2*, and *Job2* further propagates the exception to *Job0*.

Supervisor Job

By default, a coroutine starts with a simple 'Job' element.



If *Job1* throws an exception, then *Job2*(which was started in the same scope) will be cancelled. But if we don't want *Job2* to get cancelled, then we have to use 'SupervisorJob' instead of the simple 'Job' element.

```
val scope = CoroutineScope(Dispatchers.Default + SupervisorJob())

scope.launch {
    // Job1
}

scope.launch {
    // Job2
}
```


them explicitly like:

```
CoroutineScope(Dispatchers.IO).launch{
    // do some work and launch a child coroutine
    CoroutineScope(Dispatchers.Main + SupervisorJob()) {
        // we can override inherited context elements in a child coroutine
    }
}
```

changingdefaultcontext.kt hosted with ❤️ by GitHub view raw

If you want to learn more about coroutines, check these out:

- Kotlin Coroutines(Part — 1): The Basics
- Kotlin Coroutines(Part — 2): The 'Suspend' Function
- Kotlin Coroutines(Part — 3): Coroutine Context
- Kotlin Coroutines(Part — 5): Exception Handling

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

Engineering Android App Development Kotlin Coroutine Android