Ken Yee

Follow 170 Followers About

Android Kotlin Coroutine Best Practices

Ken Yee Feb 5, 2019 · 6 min read

The is a continuously maintained set of best practices for using Kotlin Coroutines on Android. Please comment below if you have any suggestions on anything that should be added.

1. Handling Android Lifecycles

In a similar way that you use CompositeDisposables with RxJava, Kotlin Coroutines have to be cancelled at the right time with awareness of Android Livecycles with Activities and Fragments.

a) Using Android Viewmodels

This is the easiest way to set up coroutines so they're shut down at the right time, but it only works inside an Android ViewModel which has an onCleared function that coroutine jobs can be reliably cancelled from:

```
private val viewModelJob = Job()
private val uiScope = CoroutineScope(Dispatchers.Main + viewModelJob)

override fun onCleared() {
 super.onCleared()
 uiScope.coroutineContext.cancelChildren()
}
```

"viewModelscope.launch { }". Note that 2.x means your app will need to be on AndroidX because I'm not sure they plan on backporting this to the 1.x version of ViewModels.

b) Using Lifecycle Observers

This other technique creates a scope that you attach to an activity or fragment (or anything else that implements an Android Lifecycle):

```
/**
 * Coroutine context that automatically is cancelled when UI is
destroyed
 */
class UiLifecycleScope : CoroutineScope, LifecycleObserver {

    private lateinit var job: Job
    override val coroutineContext: CoroutineContext
        get() = job + Dispatchers.Main

    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    fun onCreate() {
        job = Job()
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    fun destroy() = job.cancel()
}

... inside Support Lib Activity or Fragment
private val uiScope = UiLifecycleScope()

override fun onCreate(savedInstanceState: bundle) {
  super.onCreate(savedInstanceState)
  lifecycle.addObserver(uiScope)
}
```

c) GlobalScope

If you use GlobalScope, it's a scope that lasts the lifetime of the app. You would use this for doing background synchronization, repo refreshes, etc. (not tied to an Activity lifecycle).

Open in app

Services can cancel their jobs in the onDestroy.

```
private val serviceJob = Job()
private val serviceScope = CoroutineScope(Dispatchers.Main +
serviceJob)

override fun onCleared() {
 super.onCleared()
 serviceJob.cancel()
}
```

2. Handling Exceptions

a) In async vs. launch vs. runBlocking

It's important to note that exceptions in a launch{} block will crash the app without an exception handler. Always set up a default exception handler to pass as a parameter to launch.

An exception within a runBlocking{} block will crash the app unless you add a try catch. Always add a try/catch if you're using runBlocking. Ideally, only use runBlocking for unit tests.

An exception thrown within an async{} block will not propagate or run until the block is awaited because it's really a Java Deferred underneath. The calling function/method should catch exceptions.

b) Catching exceptions

If you use async to run code that may throw exceptions, you have to wrap the code in a coroutineScope to catch exceptions properly (thanks to LouisC for the example):

```
try {
    coroutineScope {
        val mayFailAsync1 = async {
            mayFail1()
        }
```

Open in app

```
        useResult(mayFailAsync1.await(), mayFailAsync2.await())
    }
} catch (e: IOException) {
    // handle this
    throw MyIoException("Error doing IO", e)
} catch (e: AnotherException) {
    // handle this too
    throw MyOtherException("Error doing something", e)
}
```

When you catch the exception, wrap it in another Exception (similar to what you do for RxJava) so that you get the stacktrace line in your own code instead of seeing a stacktrace with only coroutine code.

c) Logging exceptions

If using GlobalScope.launch or an actor, always pass in an exception handler that can log exceptions. E.g.

```
val errorHandler = CoroutineExceptionHandler { _, exception ->
  // log to Crashlytics, logcat, etc.
}
val job = GlobalScope.launch(errorHandler) {
...
}
```

Almost always, you should structured scopes on Android and a handler should be used:

```
val errorHandler = CoroutineExceptionHandler { _, exception ->
  // log to Crashlytics, logcat, etc.; can be dependency injected
}
val supervisor = SupervisorJob() // cancelled w/ Activity Lifecycle

with(CoroutineScope(coroutineContext + supervisor)) {
  val something = launch (errorHandler) {
    ...
  }
}
```

d) Consider Result/Error Sealed Class

Consider using a result sealed class that can hold an error instead of throwing exceptions:

```
sealed class Result<T, E> {
  data class Success(val data:T): Result()
  data class Error(val error: E): Result()
}
```

e) Name Coroutine Context

When declaring an async lambda, you can also name it like so:

```
async(CoroutineName("MyCoroutine")) { }
```

If you’re creating your own thread to run in, you can also name it when creating this thread executor:

```
newSingleThreadContext("MyCoroutineThread")
```

3. Executor Pools and Default Pool Sizes

Coroutines is really cooperative multitasking (with compiler assistance) on a limited thread pool size. That means that if you do something blocking in your coroutine (e.g., use a blocking API), you will tie up the entire thread until the blocking operation is done. The coroutine also won’t suspend unless you do a yield or delay, so if you have a long processing loop, be sure to check if the coroutine has been cancelled (call “ensureActive()” on the scope) so you can free up the thread; this is similar to how RxJava works.

only change UI elements in this context. There is also a Dispatchers.Unconfined which can hop between UI and background threads so it isn't on a single thread; this generally should not be used except in unit tests. There's a Dispatchers.IO for IO handling (network calls that suspend often). Finally, there is a Dispatchers.Default which is the main background thread pool but this is limited to the number of CPUs.

In practice, you should use an interface for common dispatchers that are passed in via you class' constructor so that you can swap different ones for testing. E.g.:

```
interface CoroutineDispatchers {
  val UI: Dispatcher
  val IO: Dispatcher
  val Computation: Dispatcher
  fun newThread(val name: String): Dispatcher
}
```

4. Avoiding Data Corruption

Do not have suspending functions modify data outside the function. For example, this can have unintended data modification if the two methods are run from different threads:

```
val list = mutableListOf(1, 2)
suspend fun updateList1() {
  list[0] = list[0] + 1
}
suspend fun updateList2() {
  list.clear()
}
```

You can avoid this type of issue by:
- having your coroutines return an immutable object instead of reaching out and changing one
- run all these coroutines in a single threaded context that's created via: newSingleThreadContext("contextname")

These should rules need to be added for release builds of your app:

```
-keepnames class kotlinx.coroutines.internal.MainDispatcherFactory {}
-keepnames class kotlinx.coroutines.CoroutineExceptionHandler {}
-keepnames class
kotlinx.coroutines.android.AndroidExceptionPreHandler {}
-keepnames class kotlinx.coroutines.android.AndroidDispatcherFactory
{}

-keep class kotlinx.coroutines.internal.MainDispatcherFactory {}
-keep class kotlinx.coroutines.CoroutineExceptionHandler {}
-keep class kotlinx.coroutines.android.AndroidExceptionPreHandler {}
-keep class kotlinx.coroutines.android.AndroidDispatcherFactory {}
-keepclassmembernames class kotlinx.** { volatile <fields>; }
```

6. Interop with Java

If you're working on a legacy app, you'll no doubt have a significant chunk of Java code. You can call coroutines from Java by returning a CompletableFuture (be sure to include the kotlinx-coroutines-jdk8 artifact):

```
doSomethingAsync(): CompletableFuture<List<MyClass>> =
   GlobalScope.future { doSomething() }
```

7. Retrofit Don't Need withContext

If you're using the Retrofit coroutines adapter, you get a Deferred which uses okhttp's async call under the hood. So you don't need to add withContext(Dispatchers.IO) like you'd have to do with RxJava to make sure the code runs on an IO thread; if you don't use the Retrofit coroutines adapter and call a Retrofit Call directly, you do need withContext.

The Android Arch Components Room DB also automatically does work on a non-UI context, so you don't need withContext.

8. Turn on Debug Mode For Better Stacktraces

Open in app

```
test {
    systemProperty 'kotlinx.coroutines.debug', 'on'
}
```

For Android, in your application startup, set the same property depending on whether you're in debug mode:

```
System.setProperty("kotlinx.coroutines.debug", if (BuildConfig.DEBUG)
"on" else "off")
```

And include the Debug Agent in your app so you get improved stacktraces by adding it as a dependency in your build.gradle:

```
dependencies {
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-
debug:$coroutine_version"
}
```

Once you have the debug agent installed in your application via DebugProbes.install(), you can then do DebugProbes.dumpCoroutines() or DebugProbes.printJob to dump the stack of running coroutines when you get an exception or when you're trying to debug a deadlock.

References:

- https://medium.com/capital-one-tech/kotlin-coroutines-on-android-things-i-wish-i-knew-at-the-beginning-c2f0b1f16cff
- https://speakerdeck.com/elizarov/fresh-async-with-kotlin
- https://medium.com/@michaelbukachi/coroutines-and-idling-resources-c1866bfa5b5d

linkId=63267803

- https://proandroiddev.com/managing-exceptions-in-nested-coroutine-scopes-9f23fd85e61

Android App Development Kotlin Coroutine Android AndroidDev