Constructing a coroutine scope

> This is a chapter from the book Kotlin Coroutines. You can find it on LeanPub or Amazon.

In previous chapters, we've learned about the tools needed to construct a proper scope. Now it is time to summarize this knowledge and see how it is typically used. We will see two common examples: one for Android, and one for backend development.

CoroutineScope factory function

`CoroutineScope` is an interface with a single property `coroutineContext`.

PREVIOUS

```
interface CoroutineScope {
    val coroutineContext: CoroutineContext
```

NEXT

```
}
```

Therefore, we can make a class implement this interface and just directly call coroutine builders in it.

```kotlin
class SomeClass : CoroutineScope {
   override val coroutineContext: CoroutineContext = Job()

   fun onStart() {
       launch {
           // ...
       }
   }
}
```

However, this approach is not very popular. On one hand, it is convenient; on the other, it is problematic that in such a class we can directly call other `CoroutineScope` methods like `cancel` or `ensureActive`. Even accidentally, someone might cancel the whole scope, and coroutines will not start anymore. Instead, we generally prefer to hold a coroutine scope as an object in a property and use it to call coroutine builders.

```kotlin
class SomeClass {
   val scope: CoroutineScope = ...

   fun onStart() {
       scope.launch {
           // ...
       }
   }
}
```

The easiest way to create a coroutine scope object is by using the `CoroutineScope` factory function[1]. It creates a scope with provided context (and an additional `Job` for structured concurrency if no job is already part of the context).

```kotlin
public fun CoroutineScope(
   context: CoroutineContext
): CoroutineScope =
   ContextScope(
       if (context[Job] != null) context
       else context + Job()
   )

internal class ContextScope(
   context: CoroutineContext
) : CoroutineScope {
   override val coroutineContext: CoroutineContext = context
```

```
    override fun toString(): String =
        "CoroutineScope(coroutineContext=$coroutineContext)"
}
```

Constructing a scope on Android

In most Android applications, we use an architecture that is a descendant of MVC: currently mainly MVVM or MVP. In these architectures, we extract presentation logic into objects called ViewModels or Presenters. This is where coroutines are generally started. In other layers, like in Use Cases or Repositories, we generally just use suspending functions. Coroutines might also be started in Fragments or Activities. Regardless of where coroutines are started on Android, how they are constructed will most likely be the same. Let's take a `MainViewModel` as an example: let's say it needs to fetch some data in `onCreate` (which is called when a user enters the screen). This data fetching needs to happen in a coroutine which needs to be called on some scope object. We will construct a scope in the `BaseViewModel` so it is defined just once for all view models. So, in the `MainViewModel`, we can just use the `scope` property from `BaseViewModel`.

```
abstract class BaseViewModel : ViewModel() {
    protected val scope = CoroutineScope(TODO())
}

class MainViewModel(
    private val userRepo: UserRepository,
    private val newsRepo: NewsRepository,
) : BaseViewModel {

    fun onCreate() {
        scope.launch {
            val user = userRepo.getUser()
            view.showUserData(user)
        }
        scope.launch {
            val news = newsRepo.getNews()
                .sortedByDescending { it.date }
            view.showNews(news)
        }
    }
}
```

Time to define a context for this scope. Given that many functions in Android need to be called on the Main thread, `Dispatchers.Main` is considered the best option as the default dispatcher. We will use it as a part of our default context on Android.

Do you want to be the first to know about new articles?
Follow Marcin Moskala on Twitter!
@marcinmoskala
Kt. Academy

```kotlin
abstract class BaseViewModel : ViewModel() {
   protected val scope = CoroutineScope(Dispatchers.Main)
}
```

Second, we need to make our scope cancellable. It is a common feature to cancel all unfinished processes once a user exits a screen and `onDestroy` (or `onCleared` in case of ViewModels) is called. To make our scope cancellable, we need it to have some `Job` (we do not really need to add it, because if we don't it will be added by the `CoroutineScope` function anyway, but it is more explicit this way). Then, we can cancel it in `onCleared`.

```kotlin
abstract class BaseViewModel : ViewModel() {
   protected val scope =
       CoroutineScope(Dispatchers.Main + Job())

   override fun onCleared() {
       scope.cancel()
   }
}
```

Even better, it is a common practice to not cancel the whole scope but only its children. Thanks to that, as long as this view model is active, new coroutines can start on its `scope` property.

```kotlin
abstract class BaseViewModel : ViewModel() {
   protected val scope =
       CoroutineScope(Dispatchers.Main + Job())

   override fun onCleared() {
       scope.coroutineContext.cancelChildren()
   }
}
```

We also want different coroutines started on this scope to be independent. When we use `Job`, if any of the children is cancelled due to an error, the parent and all its other children are cancelled as well. Even if there was an exception when loading user data, it should not stop us from seeing the news. To have such independence, we should use `SupervisorJob` instead of `Job`.

```
abstract class BaseViewModel : ViewModel() {
   protected val scope =
       CoroutineScope(Dispatchers.Main + SupervisorJob())

   override fun onCleared() {
       scope.coroutineContext.cancelChildren()
   }
}
```

The last important functionality is the default way of handling uncaught exceptions. On Android, we often define what should happen in the case of different kinds of exceptions. If we receive a `401 Unauthorized` from an HTTP call, we might open the login screen. For a `503 Service Unavailable` API error, we might show a server problem message. In other cases, we might show dialogs, snackbars, or toasts. We often define these exception handlers once, for instance in some `BaseActivity`, and then pass them to view models (often via constructor). Then, we can use `CoroutineExceptionHandler` to call this function if there is an unhandled exception.

```
abstract class BaseViewModel(
   private val onError: (Throwable) -> Unit
) : ViewModel() {
   private val exceptionHandler =
       CoroutineExceptionHandler { _, throwable ->
           onError(throwable)
       }

   private val context =
       Dispatchers.Main + SupervisorJob() + exceptionHandler

   protected val scope = CoroutineScope(context)

   override fun onCleared() {
       context.cancelChildren()
   }
}
```

An alternative would be to hold exceptions as a live data property, which is observed in the `BaseActivity` or another view element.

```
abstract class BaseViewModel : ViewModel() {
   private val _failure: MutableLiveData<Throwable> =
       MutableLiveData()
   val failure: LiveData<Throwable> = _failure
```

```
    private val exceptionHandler =
        CoroutineExceptionHandler { _, throwable ->
            _failure.value = throwable
        }

    private val context =
        Dispatchers.Main + SupervisorJob() + exceptionHandler

    protected val scope = CoroutineScope(context)

    override fun onCleared() {
        context.cancelChildren()
    }
}
```

`viewModelScope` and `lifecycleScope`

In modern Android applications, instead of defining your own scope, you can also use `viewModelScope` (needs `androidx.lifecycle:lifecycle-viewmodel-ktx` version `2.2.0` or higher) or `lifecycleScope` (needs `androidx.lifecycle:lifecycle-runtime-ktx` version `2.2.0` or higher). How they work is nearly identical to what we've just constructed: they use `Dispatchers.Main` and `SupervisorJob`, and they cancel the job when the view model or lifecycle owner gets destroyed.

```
// Implementation from lifecycle-viewmodel-ktx version 2.4.0
public val ViewModel.viewModelScope: CoroutineScope
    get() {
        val scope: CoroutineScope? = this.getTag(JOB_KEY)
        if (scope != null) {
            return scope
        }
        return setTagIfAbsent(
            JOB_KEY,
            CloseableCoroutineScope(
                SupervisorJob() +
                    Dispatchers.Main.immediate
            )
        )
    }

internal class CloseableCoroutineScope(
    context: CoroutineContext
) : Closeable, CoroutineScope {
    override val coroutineContext: CoroutineContext = context

    override fun close() {
        coroutineContext.cancel()
    }
}
```

Using `viewModelScope` and `lifecycleScope` is convenient and recommended if we do not need any special context as a part of our scope (like `CoroutineExceptionHandler`). This is why this approach is chosen by many (maybe most) Android applications.

```
class ArticlesListViewModel(
    private val produceArticles: ProduceArticlesUseCase,
) : ViewModel() {

    private val _progressBarVisible =
        MutableStateFlow(false)
    val progressBarVisible: StateFlow<Boolean> =
        _progressBarVisible

    private val _articlesListState =
        MutableStateFlow<ArticlesListState>(Initial)
    val articlesListState: StateFlow<ArticlesListState> =
        _articlesListState

    fun onCreate() {
        viewModelScope.launch {
            _progressBarVisible.value = true
            val articles = produceArticles.produce()
            _articlesListState.value =
                ArticlesLoaded(articles)
            _progressBarVisible.value = false
        }
    }
}
```

Constructing a coroutine on the backend

Many backend frameworks have built-in support for suspending functions. Spring Boot allows controller functions to be suspended. In Ktor, all handlers are suspending functions by default. Thanks to that, we rarely need to create a scope ourselves. However, assuming that we do (maybe because we need to start a task or work with an older version of Spring), what we most likely need is:

- a custom dispatcher with a pool of threads (or `Dispatchers.Default`);
- `SupervisorJob` to make different coroutines independent;
- probably some `CoroutineExceptionHandler` to respond with proper error codes, send dead letters[2], or log problems.

```
@Configuration
public class CoroutineScopeConfiguration {

   @Bean
   fun coroutineDispatcher(): CoroutineDispatcher =
       Dispatchers.IO.limitedParallelism(5)

   @Bean
   fun coroutineExceptionHandler() =
       CoroutineExceptionHandler { _, throwable ->
           FirebaseCrashlytics.getInstance()
               .recordException(throwable)
       }

   @Bean
   fun coroutineScope(
       coroutineDispatcher: CoroutineDispatcher,
       coroutineExceptionHandler: CoroutineExceptionHandler,
   ) = CoroutineScope(
       SupervisorJob() +
           coroutineDispatcher +
           coroutineExceptionHandler
   )
}
```

Such a scope is most often injected into classes via the constructor. Thanks to that, the scope can be defined once to be used by many classes, and it can be easily replaced with a different scope for testing purposes.

Constructing a scope for additional calls

As explained in the Additional operations section of the Coroutine scope functions chapter, we often make scopes for starting additional operations. These scopes are then typically injected via arguments to functions or the constructor. If we only plan to use these scopes to suspend calls, it is enough if they just have a `SupervisorScope`.

WORKSHOP
start using coroutines like a master with this hands-on training.

```
val analyticsScope = CoroutineScope(SupervisorJob())
```

All their exceptions will only be shown in logs; so, if you want to send them to a monitoring system, use `CoroutineExceptionHandler`.

```
private val exceptionHandler =
   CoroutineExceptionHandler { _, throwable ->
       FirebaseCrashlytics.getInstance()
           .recordException(throwable)
   }

val analyticsScope = CoroutineScope(
   SupervisorJob() + exceptionHandler
)
```

Another common customization is setting a different dispatcher. For instance, use `Dispatchers.IO` if you might have blocking calls on this scope, or use `Dispatchers.Main` if you might want to modify the main view on Android (if we set `Dispatchers.Main`, testing on Android is easier).

```
val analyticsScope = CoroutineScope(
   SupervisorJob() + Dispatchers.IO
)
```

Summary

I hope that after this chapter you will know how to construct scopes in most typical situations. This is important when using coroutines in real-life projects. This is enough for many small and simple applications, but for those that are more serious we still need to cover two more topics: proper synchronization and testing.

1: A function that looks like a constructor is known as a fake constructor. This pattern is explained in Effective Kotlin Item 33: Consider factory functions instead of constructors.

2: This is a popular microservices pattern that is used when we use a software bus, like in Apache Kafka.

The author:

Marcin Moskała

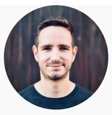
Marcin Moskala is an experienced developer and Kotlin trainer. He is the founder of the Kt. Academy, an official JetBrains partner for Kotlin training, author of the books Effective Kotlin, Kotlin Coroutines, Functional Kotlin and Android Development with Kotlin. He is also the main author of the biggest medium publication about Kotlin and a speaker invited to many programming conferences.

Reviewers:

Nicola Corti

Nicola Corti is a Google Developer Expert for Kotlin. He has been working with the language since before version 1.0 and he is the maintainer of several open-source libraries and tools.

He's currently working as Android Infrastructure Engineer at Spotify in Stockholm, Sweden.



Furthermore, he is an active member of the developer community. His involvement goes from speaking at international conferences about Mobile development to leading communities across Europe (GDG Pisa, KUG Hamburg, GDG Sthlm Android).

In his free time, he also loves baking, photography, and running.

Add a comment

Write here...

Submit

Comments

Thanh Dang 2023-04-17T18:43:23.487Z

Thank you so much for great explanation. And I have some questions:

1. Is the order matter? E.g. SupervisorJob() + coroutineDispatcher + coroutineExceptionHandler is different from coroutineDispatcher + SupervisorJob() + coroutineExceptionHandler.
2. Could you please explain what is the difference between SupervisorJob vs. SupervisorScope?

Twitter
@ktdotacademy

Email
contact@kt.academy

Newsletter
Sign up

We have a community of more than 3000 followers and we only post programming-related content.

We are happy to talk about our workshops and adjust them to your needs. Contact us if you have any questions.

Stay updated with our articles and workshops. We only send programming-related content.

© Marcin Moskała 2023 Privacy policy Sitemap