Coroutine against Goroutine: comparison between ***Kotlin*** and ***GoLang*** concurrency

Activity Project in *Operating Systems M*

Luca Marchegiani

February 24, 2025

Contents

Abstract

In this paper we are going to make a comparison between `Kotlin` and `Go` concurrency that is the main focus of the activity project in *Operating Systems M* course of the master degree in *Computer Science Engineering* at the *Alma Mater Studiorum* University of Bologna.

After a description of the concurrency management in these two languages, we will try to go into an experimental comparison using also a previously made project of the author in the courses of *Software System Engineering* and *Mobile Systems M*.

1 Introduction

`Kotlin` is a modern, multi-platform and blended programming language developed by Jetbrains that works on JVM such as `Scala`. `Kotlin` is completely interoperable with `Java`[1] and it is *object-oriented* with strong elements of functional programming that make it more powerful than his father `Java`. As specified in the main page of the official website,

[1] All classes written in `Kotlin` are callable from `Java` code and vice-versa.

`Kotlin` has also the advantages to be *concise*, *safe in nullability*, *expressive*, *asynchronous*, *interoperable* and *multiplatform*.

Furthermore, since 2019, Google has declared `Kotlin` as the preferred language for developing `Android` applications, establishing it as the *de facto* official language. As anticipated, `Kotlin` supports also multiplatform allowing the developer to write `Kotlin` code that can be compiled for native platforms (including `Android` and `iOS`), `JVM` and `JavaScript`.

`Go` is an open source programming language developed and supported by `Google`. It's an *imperative* and *object-oriented* language that is strongly designed for concurrency thanks to its very easy way to launch process and its efficiency. The idea of this language is to maintain the run-time efficiency of `C` but with more readability and usability. Differently from `C`, `Go` has *memory safety*, *garbage collection* and *structural typing* as said by Wikipedia.

In the last years, `Go` also supports mobile platform (`Android` and `iOS`) as described in the official wiki, by writing *all-Go native mobile applications* or *SDK applications* with bindings for `Java` or `Objective-C`. There is also a toolkit called Fyne that is free and open source that makes easy to build graphical application also for mobile using `Go` .

From the concurrency point of view, **both of this language supports coroutines** as concurrent units of execution. *Coroutines* are lightweight processes that can run over multiple OS threads, allowing to save on thread management costs.

Coroutines and threads are very similar, but the main difference is that the firsts are *non-preemptive* (or *cooperatives*) differently from the seconds that are typically *preemptive* and scheduled by the OS. Indeed, the execution of a coroutine can be suspended and resumed by the developer, calling some operations, and not by the OS.

We will go in the details for both of this language.

2 Overview of the concurrency in `Kotlin` and `Go`

As specified in the introduction, `Kotlin` and `Go` exposes concurrency thanks to **coroutines** and other tools that let the developer manage their synchronization. To be precise, while `Go` has only coroutines to implement concurrency, `Kotlin` has a more sophisticated and complete *framework* for that: indeed, lots of `Kotlin` application (including `Android` apps) run over a `JVM` (or on the `ART`), so all the standard `Java` threading packages are available.

Anyway, as we already said, **coroutines** are **lightweight processes for cooperation that execute over OS threads** and that can suspend at a certain point and resume later at the same point, but with the possibility to execute on a different thread. The main advantage of using them instead threads is that **switching between coroutines does not require any *system call***, ensuring lower management costs. This introduces great advantages, especially for *asynchronous* computation.

To conclude this general introduction, coroutines can use *shared memory* or *message passing*, based on what developer choose to use. Indeed, both `Kotlin` and `Go` provides supports for the two mechanisms: *semaphore* and *mutex* for shared memory and *channels* for *message passing*.

2.1 `Kotlin` concurrency overview

We said that `Kotlin` is based on the `JVM` (but can also compile `JavaScript` or native using LLVM) and is interoperable with `Java`. The main implementation of `Kotlin` is done in its compiler: for `Kotlin` on `JVM`, all classes are compiled as normal `Java` classes. This means that **`Kotlin` can access to all `threading`** packages exposed by `Java` (and this is also valid for `Android`). So, in `Kotlin` **it is possible to use the standard threads** that are provided by `Java`.

Even if there is the possibility to use the standard `Java` threads, as anticipated, `Kotlin` introduces the new `kotlinx.coroutines` library to realize concurrency by adopting *coroutines*. Coroutines are *instances of suspendable computation* that let the developer to easily write **asynchronous and non-blocking code** that can run concurrently, without using *callback* or *promises*. The main mechanism that turns around `Kotlin` coroutines is the concept of **suspending function**: a special type of `Kotlin` method that can suspend the execution of the current coroutine without blocking the current thread. A function can be marked as `suspend` simply by adding this modifier to its signature.

2.1.1 Realization of coroutines in `Kotlin`

To go into the details of coroutines in `Kotlin` , we have to introduce some basic concepts[2]:

- Job:
  The object that represents the *background job* of one coroutine. When a coroutine is launched, the `launch` method immediately returns the reference to the `Job` associated to the coroutine. **A job represents the lifecycle of a coroutine** and can be used to *cancel* its execution. Then, it can have six possible states, each coded by a combination of the three properties of the `Job` class: `isActive`, `isCompleted` and `isCancelled`. The following table summarizes the possible states of a `Job` and the value of the three properties for each state:

State	Type	isActive	isCompleted	isCancelled
New	initial	`false`	`false`	`false`
Active	initial	`true`	`false`	`false`
Completing	transient	`true`	`false`	`false`
Cancelling	transient	`false`	`false`	`true`
Cancelled	final	`false`	`true`	`true`
Completed	final	`false`	`true`	`false`

  Table 1: States of a `Job`

  The graphs shown in the 1 represents the entire lifecycle of a `Job`, so it also represents the lifecycle of a `Kotlin` coroutine.

- CoroutineDispatcher:
  As we already said, in their lifecycle coroutine can run in different threads. For example, suppose to have a coroutine C_1 that is started on the thread T_1 that executes its code:

[2]See medium.com for additional details.

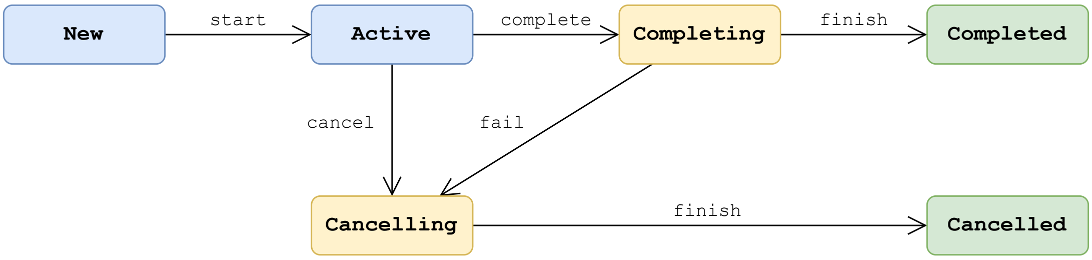

Figure 1: Lifecycle of `Kotlin` coroutine in `Job`

1. C_1 starts its execution on thread T_1;
2. during its execution, C_1 encounter an instruction I_1 that suspends itself waiting for something (but the instruction does not suspend the thread);
3. C_1 is suspended by I_1 and another coroutine C_2 starts to execute on T_1;
4. C_2 is executing on T_1 while I_1 returns resuming C_1 from its suspension, but now T_1 is not available because it is executing the code of C_2;
5. C_1 may execute on another available thread T_2 while C_2 continue to run in parallel on T_1 (if the configuration allows it).

`CoroutineDispatcher` is the object that *dispatchs* coroutines between the different available threads. The `CoroutineDispatcher` is important because it determines in which thread a couroutine can run: for example, in `Android` using `Dispatchers.Main` means that the coroutine will be executed confined to the `Main` thread[3].

By default, when a coroutine is created, it is used the `Dispatchers.Default` that uses *worker* threads, a shared pool of threads on the `JVM` in which coroutines can execute in parallel.

- CoroutineContext:
  Each coroutine in `Kotlin` has a *context* that is *immutable*. A context is simply a set of *elements* that realizes the concept of *context* in which the coroutine executes. The main elements in a context are:
  - the `Job` that represents the coroutine;
  - the `CoroutineDispatcher` that dispatches the execution of coroutine over the threads;
  - the `CoroutineName` that is the name associated to the coroutine (useful for debugging);
  - the `CoroutineExceptionHandler` that is an handler for all the exception thrown during the execution of the coroutine;
  - the `ContinuationInterceptor` that allows to define *how* the coroutine should continue after a resume (a sort of *callback* that is invoked on coroutine resume).

[3] In this case, the coroutine can update the `UI`. There are also dispatchers for `JavaFX` or `Swing` for `Kotlin JVM` to force coroutines to be executed on the thread that can update the user interface.

Notice that `CoroutineContext` **is immutable, but it is possible to add elements using the plus operator** that produces a new context instance. In addition, **all of these elements extends `CoroutineContext`** so, using `plus` operator let to easily create a context that is a *join* of others. For example:

```
val newContext = CoroutineName("MyCoroutine") + Dispatchers.Main
```

creates a new context named *MyCoroutine* in which coroutine will be executed using `Disparchers.Main`.

A context can be passed to the coroutine builder before launching it or, if the context has to be changed while the coroutine is running, it is possible to use the withContext suspend function. `Kotlin` has also a default context for builders that is `EmptyCoroutineContext` which can be also used with `plus` operator to create new contexts.

- CoroutineScope:
  Each coroutine in `Kotlin` must have a *scope* which delimits the lifetime of the coroutine. The `CoroutineScope` consists in only one property: `coroutineContext`, an instance of `CoroutineContext`. In addition to this, the `CoroutineScope` has also some *extension functions* such as `launch` that is a builder for coroutines.

  Then, when `launch` is invoked using a `CoroutineScope`, it launches a new coroutine and its context is *inherited* from those of the scope. In this way, all the elements of the parents and its cancellation are propagated to the child; then, if a scope is cancelled, all the coroutine launched starting from it will be cancelled.

In `Kotlin` **the concept of *coroutine* can be summarized by the formula**:

Coroutine = `CoroutineContext` + `Job`

In order to launch a coroutine, the developer has to:

1. create an instance of `CoroutineScope`, for example using the `runBlocking` scope builder;
2. call a coroutine builder starting from the created scope, such as `launch`, that returns the `Job` associated to the coroutine.

Here there is an example of the creation of a simple coroutine taken from the official documentation on kotlinlang.org:

```
fun main() = runBlocking { // this: CoroutineScope
 launch { // launch a new coroutine and continue
  delay(1000L) // non-blocking delay for 1 second
  println("World!") // print after delay
 }
 println("Hello") // main coroutine continues while a previous one is
    ↪delayed
}
```

that produces this result on the console:

```
Hello
World!
```

To fully understand this snippet, the reader should know something about *higher-order functions* and *receivers* which are concepts that came from *functional programming* available in `Kotlin` .

Notice that `runBlocking` has also an optional `CoroutineContext` argument that can be used to pass elements that will be added to the context of the scope. All of these elements are inherited by the child except for the `Job` that is created by the coroutine builder.

For example:

```
runBlocking(CoroutineName("MyCoroutine")) {
 val parentScope = this
 println("parent  : $coroutineContext")
 val job1 = launch {
  println("launch1 : $coroutineContext," +
  " childScope == parentScope : ${this == parentScope}")
 }
 val job2 = launch {
  println("launch2 : $coroutineContext, " +
  "childScope == parentScope : ${this == parentScope}")
 }
 joinAll(job1, job2)
}
```

produces an output similar to:

```
parent  : [CoroutineName(MyCoroutine),
          BlockingCoroutine{Active}@68f7aae2, BlockingEventLoop@4f47d241]
launch1 : [CoroutineName(MyCoroutine),
          StandaloneCoroutine{Active}@d70c109, BlockingEventLoop@4f47d241],
          childScope == parentScope : false
launch2 : [CoroutineName(MyCoroutine),
          StandaloneCoroutine{Active}@1bc6a36e, BlockingEventLoop@4f47d241],
          childScope == parentScope : false
```

As you can see, both child scopes are different from parent even if they are in relationship: cancelling parent scope cancels those of the children, but the reverse is not true. About the context, it's clear that child contexts are completely inherited from the parent except for the `Job` instances[4] that are different.

2.1.2 Synchronization between coroutines in `Kotlin`

We highlight that **coroutines in `Kotlin` can use shared memory or *messages*** to synchronize themselves. In particular:

- The package `kotlinx.coroutines.sync` exposes the classical tools for synchronization in a shared memory environment (*mutex* and *semaphore*).

  Notice that this type of synchronization is very basic if compared with the standard `Java` tools for concurrency such as `Lock` and `Condition`; at this moment, `Kotlin` does not define any mechanism similar to `Java` condition, but, however, it's very easy to implement it (for example, we have an implementation made by the author called `CoroutineCondition` that uses the `Continuation` object of a coroutine).

[4] `BlockingCoroutine` and `StandaloneCoroutine` are `Job` extensions.

[See `MutexPiCalculation.kt` for a basic example]

- The package `kotlinx.coroutines.channels` exposes classical tools for synchronization by exchanging messages in no shared memory environment (*channels*). The main entity of this package is `Channel`, that is very similar to `BlockingQueue` but with suspending operations instead of the `Java` blocking methods.

  Two coroutines can use a channel in order to transfer a single value that came from the *producer* (the coroutine that invoke the `send` operation) and goes to the *consumer* (the coroutine that invoke the `receive` operation); originally, in `Kotlin` channels were **bidirectional** and **symmetric** (one-to-one), but in the last updates of the language it is possible to have some *asymmetric* behavior thanks to *Fan-Out* and *Fan-In* mechanisms. Nevertheless, `Kotlin` has more sophisticated tools to make multiple coroutines able to have an asymmetric communication and we will see them below.

  The semantic of *send/receive* operations depends on the nature of the channel that is determinated by its capacity, but the communication can be **synchronous** or **asynchronous** with also some little variations of these (for example, **rendez-vous**). Notice that, as anticipated, a channel can safely be shared between coroutines, but the developer has to pay attention because the `receive` operation can easily lead to competition problems if invoked parallel from two or more coroutines.

  [See `ChannelPiCalculation.kt` for a basic example]

- The package `kotlinx.coroutines.flow` exposes the tools for using *flows* that are defined by the documentation as *asynchronous cold stream of elements* that can safely be used to synchronize multiple coroutine at the same time.

  Flow can be more formally defined as **mono-directional, one-to-many and asynchronous** channels with the possibility to be **buffered** for replay strategies. In the latest versions of `Kotlin` , flows replaced `BroadcastChannel`.

2.1.3 Suspending functions

Suspending functions are normal `Kotlin` methods but with the feature that **they can *suspend* the execution of a coroutine**. The main example of suspending function it's `delay` that suspends the execution of the coroutine which calls it for a specified time.

Let's make an example (`SuspendingFunctionExample.kt`):

```kotlin
suspend fun sleep(who : String, timeMillis : Long) {
  println("$who: I'm going to sleep for $timeMillis milliseconds...")
  delay(timeMillis)
  println("$who: Good morning, I wake up!")
}

suspend fun pollAlive(who : String, pollingTime : Long) {
  while (true) {
    delay(pollingTime)
    println("$who: i'm alive [thread=${Thread.currentThread()}]")
  }
}

```

```
suspend fun sayHello(who : String) {
  println("$who : Hello... I'm a coroutine " +
    "[thread=${Thread.currentThread()}]")
  println("$who : My context: $coroutineContext")
}

fun main() {
  @OptIn(DelicateCoroutinesApi::class)
  val ctx = newSingleThreadContext("CoroutineSingleThread")

  runBlocking(ctx) {
    println("parent: [thread=${Thread.currentThread()}]")
    val job1 = launch {
      val who = "job1"
      sayHello(who)
      sleep(who, 3000)
      sayHello(who)
    }
    val job2 = launch {
      val who = "job2"
      sayHello(who)
      pollAlive("job2", 500)
      sayHello(who)
    }
    job1.join()
    println("parent: job1 = $job1, job2 = $job2")
    job2.cancelAndJoin()
    println("parent: job1 = $job1, job2 = $job2")
  }
}
```

it produces:

```
parent: [thread=Thread[CoroutineSingleThread,5,main]]
job1 : Hello... I'm a coroutine [
                thread=Thread[CoroutineSingleThread,5,main]]
job1 : My context: [StandaloneCoroutine{Active}@739c17c3,
                java.util.concurrent.ScheduledThreadPoolExecutor@4289a013
                [Running, pool size = 1, active threads = 1,
                        queued tasks = 1, completed tasks = 1]]
job1: I'm going to sleep for 3000 milliseconds...
job2 : Hello... I'm a coroutine [
                thread=Thread[CoroutineSingleThread,5,main]]
job2 : My context: [StandaloneCoroutine{Active}@7ce21fc2,
                java.util.concurrent.ScheduledThreadPoolExecutor@4289a013
                [Running, pool size = 1, active threads = 1,
                        queued tasks = 1, completed tasks = 2]]
job2: i'm alive [thread=Thread[CoroutineSingleThread,5,main]]
job2: i'm alive [thread=Thread[CoroutineSingleThread,5,main]]
job2: i'm alive [thread=Thread[CoroutineSingleThread,5,main]]
job2: i'm alive [thread=Thread[CoroutineSingleThread,5,main]]
job2: i'm alive [thread=Thread[CoroutineSingleThread,5,main]]
job1: Good morning, I wake up!
job1 : Hello... I'm a coroutine [
```

```
                 thread=Thread[CoroutineSingleThread,5,main]]
job1 : My context: [StandaloneCoroutine{Active}@739c17c3,
                 java.util.concurrent.ScheduledThreadPoolExecutor@4289a013
                 [Running, pool size = 1, active threads = 1,
                         queued tasks = 1, completed tasks = 8]]
parent: job1 = StandaloneCoroutine{Completed}@739c17c3,
                 job2 = StandaloneCoroutine{Active}@7ce21fc2
parent: job1 = StandaloneCoroutine{Completed}@739c17c3,
                 job2 = StandaloneCoroutine{Cancelled}@7ce21fc2
```

In this significant example we have used the `newSingleThreadContext` in order to create a context with a single thread T_x dedicated for the execution of the coroutine: all coroutines that inherits this context executes on T_x.

The example shows some important characteristic of `Kotlin` coroutines and suspending function:

1. When `job1` calls the *suspend* function `sleep(who, 3000)` and encounters the `delay` instruction at the line 3, coroutine goes into suspension for 3 seconds but, once resumed, the execution restarts exactly by the end of `delay` at line 3;
2. Since we have forced a single thread for the two coroutines, this snippet shows that even if `job1` is suspended on the `delay` (line 3), the thread is however active (not paused or suspended) and it continues to run the `job2`. This is demonstrated by all the *alive* prints of `job2` in the resulting command windows.
3. The instruction `cancelAndJoin()` let the developer easy to cancel a coroutine, waiting for its end.

To conclude, a **suspending function** is a `Kotlin` **method able to suspend the coroutine that calls it without blocking the executing thread**. From the implementation point of view, a suspending function is a normal method with an *hidden* parameter of type `Continuation` that is automatically added when the code is compiled. The implementation of this class that is provided built-in, is used by the coroutines to save their state before a suspension point.

To understand, suppose to compile the example `SuspendingFunctionExample.kt` that has been shown before on a desktop machine with `JVM`. After the normal `Kotlin` compilation we will have the executable `SuspendingFunctionExampleKt.class`, and if we decompile[5] we will see that all the suspending functions have this `Java` signature:

```java
public static final Object sleep(@NotNull String who, long timeMillis,
 @NotNull Continuation<? super Unit> paramContinuation)

public static final Object pollAlive(@NotNull String who,
 long pollingTime,
 @NotNull Continuation<? super Unit> paramContinuation)

public static final Object sayHello(@NotNull String who,
 @NotNull Continuation $completion)
```

[5]For example using *JD-GUI*.

So, a suspending function is simply compiled into a `Java` method with a `Continuation` as last parameter that can be used to suspend the coroutine that calls the function.

2.1.4 Fast overview on flows

As anticipated, even if it is possible to use `Kotlin` channels with an asymmetric communication, invoking the `receive` on the same channel in parallel from multiple coroutines can cause leaks or unexpected situations. In addition, **the information that is sent on a channel is not replicated**, so if there are multiple coroutines waiting on the same channel, only one of them will receive the information that is sent on the channel, without direct control of the developer[6]. This problem occurs only at the *receive* size: if more coroutines send messages over the same channel but there is only one *consumer* at the other side, there is no problem. Then, in `Kotlin` **it is possible to use channels to realize *many-to-one* communication**.

Nevertheless, in many situations it is useful to realize cooperation using a *one-to-many* that, as said, is not *safe* directly using channel. For this reason, `Kotlin` originally provided the `BroadcastChannel` as the official and safe way to make a *sender* able to interact with multiple *receivers*.

Anyway, as you can read in the official documentation, the `BroadcastChannel` *api* is deprecated since the `1.5.0` version of `Kotlin` and replaced with `SharedFlow` that is an implementation of `Flow` interface.

2.2 `Go` concurrency overview

As said in the official page, `Go` has a *built-in concurrency and a robust standard library* which is one of the central features of the language.

[6] In the sense that the developer can not directly choose which coroutine has to read the message.