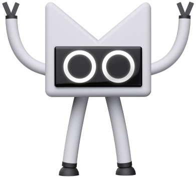
Deep dive into coroutines

on the example of Kotlin implementation

Maciej Procyk



March 27, 2023

Table of contents

Kotlin

Introduction

General description

- described as "functions whose execution you can pause"

General description

- described as “functions whose execution you can pause”
- computation can be paused and resumed without blocking a thread

General description

- described as “functions whose execution you can pause”
- computation can be paused and resumed without blocking a thread
- generalization of subroutines for cooperative multitasking

General description

- described as "functions whose execution you can pause"
- computation can be paused and resumed without blocking a thread
- generalization of subroutines for cooperative multitasking
- fast context switching

General description

- described as “functions whose execution you can pause”
- computation can be paused and resumed without blocking a thread
- generalization of subroutines for cooperative multitasking
- fast context switching
- perfect tool for implementing iterators, infinite lists, pipes

General description

- described as “functions whose execution you can pause”
- computation can be paused and resumed without blocking a thread
- generalization of subroutines for cooperative multitasking
- fast context switching
- perfect tool for implementing iterators, infinite lists, pipes
- asynchronous code in synchronous manner

Fundamental characteristics

1. the values of data local to a coroutine persist between successive calls

Fundamental characteristics

1. the values of data local to a coroutine persist between successive calls
2. the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage

Asynchronous programming

- nowadays, we write the programs that wait for a lot of time

Kotlin

Asynchronous programming

- nowadays, we write the programs that wait for a lot of time
- we write them using threads, but there are a few limitations

Asynchronous programming

- nowadays, we write the programs that wait for a lot of time
- we write them using threads, but there are a few limitations
- the number of threads cannot be too high, as they consume a lot of memory

Asynchronous programming

- nowadays, we write the programs that wait for a lot of time
- we write them using threads, but there are a few limitations
- the number of threads cannot be too high, as they consume a lot of memory
- e.g. on Linux x64 default stack size (configured with `-XX:ThreadStackSize` or `-Xss` option) is 256 KB

Kotlin

Asynchronous programming

- nowadays, we write the programs that wait for a lot of time
- we write them using threads, but there are a few limitations
- the number of threads cannot be too high, as they consume a lot of memory
- e.g. on Linux x64 default stack size (configured with `-XX:ThreadStackSize` or `-Xss` option) is 256 KB
- which means that 100K threads would consume about 24 GB of memory

Asynchronous programming

- nowadays, we write the programs that wait for a lot of time
- we write them using threads, but there are a few limitations
- the number of threads cannot be too high, as they consume a lot of memory
- e.g. on Linux x64 default stack size (configured with `-XX:ThreadStackSize` or `-Xss` option) is 256 KB
- which means that 100K threads would consume about 24 GB of memory
- that's a lot and what if we think about 1M threads

Coroutines vs threads

- the difference is that threads are scheduled on cores preemptively, while coroutines are scheduled onto threads cooperatively

Coroutines vs threads

- the difference is that threads are scheduled on cores preemptively, while coroutines are scheduled onto threads cooperatively
- starting a thread on JVM requires starting native thread, which consumes memory for its stack

Coroutines vs threads

- the difference is that threads are scheduled on cores preemptively, while coroutines are scheduled onto threads cooperatively
- starting a thread on JVM requires starting native thread, which consumes memory for its stack
- switching between threads requires going through system's kernel

Kotlin

Coroutines vs threads

- the difference is that threads are scheduled on cores preemptively, while coroutines are scheduled onto threads cooperatively
- starting a thread on JVM requires starting native thread, which consumes memory for its stack
- switching between threads requires going through system's kernel
- and is expensive in terms of CPU cycles consumed during that operation

Kotlin

Coroutines vs threads

- coroutine is user-level abstraction (implemented by the language compiler in case of Kotlin)

Coroutines vs threads

- coroutine is user-level abstraction (implemented by the language compiler in case of Kotlin)
- in the simplest case it's a single reference object in JVM heap memory

Kotlin

Coroutines vs threads

- coroutine is user-level abstraction (implemented by the language compiler in case of Kotlin)
- in the simplest case it's a single reference object in JVM heap memory
- switching between coroutines is as simple (and cheap) as invoking a regular function

Sample code starting 100 000 threads

```kotlin
fun main() {
    val counter = AtomicInteger(0)
    List(100_000) {
        thread {
            sleep(1_000)
            counter.incrementAndGet()
        }
    }.forEach { it.join() }
    println("counter = ${counter.get()}")
}
```

Kotlin

Sample code starting 100 000 threads

```kotlin
fun main() {
    val counter = AtomicInteger(0)
    List(100_000) {
        thread {
            sleep(1_000)
            counter.incrementAndGet()
        }
    }.forEach { it.join() }
    println("counter = ${counter.get()}")
}
```

- possible, but takes a lot of time (and actually they don't exist in the same time)

Kotlin

Sample code starting 100 000 threads

```kotlin
fun main() {
    val counter = AtomicInteger(0)
    List(100_000) {
        thread {
            sleep(1_000)
            counter.incrementAndGet()
        }
    }.forEach { it.join() }
    println("counter = ${counter.get()}")
}
```

- possible, but takes a lot of time (and actually they don't exist in the same time)
- if we increase the sleep time the number of threads existing in the same time would increase

Kotlin

Sample code starting 100 000 threads

```kotlin
fun main() {
    val counter = AtomicInteger(0)
    List(100_000) {
        thread {
            sleep(1_000)
            counter.incrementAndGet()
        }
    }.forEach { it.join() }
    println("counter = ${counter.get()}")
}
```

- possible, but takes a lot of time (and actually they don't exist in the same time)
- if we increase the sleep time the number of threads existing in the same time would increase
- and we would end up with `Exception in thread "main" java.lang.OutOfMemoryError ...`

Kotlin

Sample code starting 100 000 jobs

```kotlin
fun main() = runBlocking {
    val counter = AtomicInteger(0)
    List(100_000) {
        launch {
            delay(1_000)
            counter.incrementAndGet()
        }
    }.forEach { it.join() }
    println("counter = ${counter.get()}")
}
```

Sample code starting 100 000 jobs

```
fun main() = runBlocking {
    val counter = AtomicInteger(0)
    List(100_000) {
        launch {
            delay(1_000)
            counter.incrementAndGet()
        }
    }.forEach { it.join() }
    println("counter = ${counter.get()}")
}
```

- explicitly defined context is needed to start the whole process

Kotlin

Sample code starting 100 000 jobs

```kotlin
fun main() = runBlocking {
    val counter = AtomicInteger(0)
    List(100_000) {
        launch {
            delay(1_000)
            counter.incrementAndGet()
        }
    }.forEach { it.join() }
    println("counter = ${counter.get()}")
}
```

- explicitly defined context is needed to start the whole process
- code runs over 10 times faster than the threads implementation

Kotlin

Sample code starting 100 000 jobs

```
fun main() = runBlocking {
    val counter = AtomicInteger(0)
    List(100_000) {
        launch {
            delay(1_000)
            counter.incrementAndGet()
        }
    }.forEach { it.join() }
    println("counter = ${counter.get()}")
}
```

- explicitly defined context is needed to start the whole process
- code runs over 10 times faster than the threads implementation
- because it uses computer resources more thoughtful

Kotlin

Toy problem wasting computer resources on sleeping

```kotlin
fun processRequest() {
    val request = receiveRequest()
    val processed = handleRequest(request)
    sendResponse(processed)
}
```

Kotlin

Toy problem wasting computer resources on sleeping

```
fun processRequest() {
    val request = receiveRequest()
    val processed = handleRequest(request)
    sendResponse(processed)
}
```

- potentially, the application may have thousands of these tasks run concurrently

Toy problem wasting computer resources on sleeping

```kotlin
fun processRequest() {
    val request = receiveRequest()
    val processed = handleRequest(request)
    sendResponse(processed)
}
```

- potentially, the application may have thousands of these tasks run concurrently
- but every of them would consume a single thread to be executed

Toy problem wasting computer resources on sleeping

```kotlin
fun processRequest() {
    val request = receiveRequest()
    val processed = handleRequest(request)
    sendResponse(processed)
}
```

- potentially, the application may have thousands of these tasks run concurrently
- but every of them would consume a single thread to be executed
- the problem are the blocking functions which are related to IO operations and expensive processing

Kotlin

The “old” JVM solution for this problem are callbacks

```kotlin
fun processRequestCallbacks() {
    receiveRequest { request ->
        handleRequest(request) { processed ->
            sendResponse(processed)
        }
    }
}
```

The "old" JVM solution for this problem are callbacks

```kotlin
fun processRequestCallbacks() {
    receiveRequest { request ->
        handleRequest(request) { processed ->
            sendResponse(processed)
        }
    }
}
```

- functions returns immediately and run the callbacks later to get the result

Kotlin

The “old” JVM solution for this problem are callbacks

```
fun processRequestCallbacks() {
    receiveRequest { request ->
        handleRequest(request) { processed ->
            sendResponse(processed)
        }
    }
}
```

- functions returns immediately and run the callbacks later to get the result
- this introduces “callback hell” problem

The “old” JVM solution for this problem are callbacks

```kotlin
fun processRequestCallbacks() {
    receiveRequest { request ->
        handleRequest(request) { processed ->
            sendResponse(processed)
        }
    }
}
```

- functions returns immediately and run the callbacks later to get the result
- this introduces “callback hell” problem
- this code has multiple limitations - the structure of code prevents imperative coding (how to write loops?)

Writing imperative code with callbacks

```kotlin
fun imperative() {
    var number = 0
    for (i in 1..1_000)
        number = inc(number)
    println(number)
}
```

Kotlin

Writing imperative code with callbacks

```kotlin
fun imperativeCallback() {
    class LateInitCallbackWrapper<T : Any> {
        lateinit var callback: (T) -> Unit
    }

    val calls = Array(1_000) { LateInitCallbackWrapper<Int>() }
    for (i in 0..998)
        calls[i].callback = { incWithCallback(it, calls[i + 1].callback::invoke) }
    calls[999].callback = { incWithCallback(it, ::println) }
    calls[0].callback(0)
}
```

Writing imperative code with callbacks

```kotlin
fun imperativeCallback() {
    class LateInitCallbackWrapper<T : Any> {
        lateinit var callback: (T) -> Unit
    }

    val calls = Array(1_000) { LateInitCallbackWrapper<Int>() }
    for (i in 0..998)
        calls[i].callback = { incWithCallback(it, calls[i + 1].callback::invoke) }
    calls[999].callback = { incWithCallback(it, ::println) }
    calls[0].callback(0)
}
```

- hard to understand intention of programmer using callbacks

Writing imperative code with callbacks

```kotlin
fun imperativeCallback() {
    class LateInitCallbackWrapper<T : Any> {
        lateinit var callback: (T) -> Unit
    }

    val calls = Array(1_000) { LateInitCallbackWrapper<Int>() }
    for (i in 0..998)
        calls[i].callback = { incWithCallback(it, calls[i + 1].callback::invoke) }
    calls[999].callback = { incWithCallback(it, ::println) }
    calls[0].callback(0)
}
```

- hard to understand intention of programmer using callbacks
- additionally, there's extra cost related to working with the references for callbacks

Writing imperative code with callbacks

```kotlin
fun imperativeCallback() {
    class LateInitCallbackWrapper<T : Any> {
        lateinit var callback: (T) -> Unit
    }

    val calls = Array(1_000) { LateInitCallbackWrapper<Int>() }
    for (i in 0..998)
        calls[i].callback = { incWithCallback(it, calls[i + 1].callback::invoke) }
    calls[999].callback = { incWithCallback(it, ::println) }
    calls[0].callback(0)
}
```

- hard to understand intention of programmer using callbacks
- additionally, there's extra cost related to working with the references for callbacks
- this solution doesn't scale well so other approach needs to be found

Kotlin

Futures/Promises/Rx being a rescue?

```kotlin
fun processRequestPromise() {
    receiveRequestPromise()
        .flatMap { handleRequestPromise(it) }
        .subscribeBy(
            onSuccess = { sendResponse(it) },
            onError = { println("error: $it") },
        )
}
```

Futures/Promises/Rx being a rescue?

```kotlin
fun processRequestPromise() {
    receiveRequestPromise()
        .flatMap { handleRequestPromise(it) }
        .subscribeBy(
            onSuccess = { sendResponse(it) },
            onError = { println("error: $it") },
        )
}
```

- we got nicer, composable code, with no extra indentation

Kotlin

Futures/Promises/Rx being a rescue?

```kotlin
fun processRequestPromise() {
    receiveRequestPromise()
        .flatMap { handleRequestPromise(it) }
        .subscribeBy(
            onSuccess = { sendResponse(it) },
            onError = { println("error: $it") },
        )
}
```

- we got nicer, composable code, with no extra indentation
- we're able to propagate our exceptions

Kotlin

Futures/Promises/Rx being a rescue?

```kotlin
fun processRequestPromise() {
    receiveRequestPromise()
        .flatMap { handleRequestPromise(it) }
        .subscribeBy(
            onSuccess = { sendResponse(it) },
            onError = { println("error: $it") },
        )
}
```

- we got nicer, composable code, with no extra indentation
- we're able to propagate our exceptions
- but the combinators are the problem - it might be hard to learn them all

Kotlin

Futures/Promises/Rx being a rescue?

```kotlin
fun processRequestPromise() {
    receiveRequestPromise()
        .flatMap { handleRequestPromise(it) }
        .subscribeBy(
            onSuccess = { sendResponse(it) },
            onError = { println("error: $it") },
        )
}
```

- we got nicer, composable code, with no extra indentation
- we're able to propagate our exceptions
- but the combinators are the problem - it might be hard to learn them all
- while there are different libraries with other names

Kotlin solution are coroutines

```kotlin
suspend fun processRequestCoroutines() {
    val request = receiveRequestCoroutines()
    val processed = handleRequestCoroutines(request)
    sendResponseCoroutines(processed)
}
```

Kotlin solution are coroutines

```kotlin
suspend fun processRequestCoroutines() {
    val request = receiveRequestCoroutines()
    val processed = handleRequestCoroutines(request)
    sendResponseCoroutines(processed)
}
```

- need to mark functions with `suspend` modifier

Kotlin

Kotlin solution are coroutines

```kotlin
suspend fun processRequestCoroutines() {
    val request = receiveRequestCoroutines()
    val processed = handleRequestCoroutines(request)
    sendResponseCoroutines(processed)
}
```

- need to mark functions with `suspend` modifier
- but the structure of the code looks like regular code (loops, exceptions, extension functions)

Kotlin

Kotlin solution are coroutines

```
suspend fun processRequestCoroutines() {
    val request = receiveRequestCoroutines()
    val processed = handleRequestCoroutines(request)
    sendResponseCoroutines(processed)
}
```

- need to mark functions with `suspend` modifier
- but the structure of the code looks like regular code (loops, exceptions, extension functions)
- we got "not obvious" suspension points

Kotlin solution are coroutines

```kotlin
suspend fun processRequestCoroutines() {
    val request = receiveRequestCoroutines()
    val processed = handleRequestCoroutines(request)
    sendResponseCoroutines(processed)
}
```

- need to mark functions with `suspend` modifier
- but the structure of the code looks like regular code (loops, exceptions, extension functions)
- we got "not obvious" suspension points
- but the good IDE helps here a lot if we're interested in this

Kotlin

Coroutines in details

Kotlin

What's under the hood?

- suspending functions are implemented via Continuation-Passing-Style (CPS)

What's under the hood?

- suspending functions are implemented via Continuation-Passing-Style (CPS)
- every suspending function has an additional `Continuation` parameter passed implicitly when function is invoked

What's under the hood?

- suspending functions are implemented via Continuation-Passing-Style (CPS)
- every suspending function has an additional `Continuation` parameter passed implicitly when function is invoked
- consider the following signature of `suspend` function

What's under the hood?

- suspending functions are implemented via Continuation-Passing-Style (CPS)
- every suspending function has an additional `Continuation` parameter passed implicitly when function is invoked
- consider the following signature of `suspend` function

```
suspend fun <T> CompletableFuture<T>.await(): T =
```

What's under the hood?

- suspending functions are implemented via Continuation-Passing-Style (CPS)
- every suspending function has an additional `Continuation` parameter passed implicitly when function is invoked
- consider the following signature of `suspend` function

```
suspend fun <T> CompletableFuture<T>.await(): T =
```

- its actual implementation after compiler transformation has the following form

Kotlin

What's under the hood?

- suspending functions are implemented via Continuation-Passing-Style (CPS)
- every suspending function has an additional `Continuation` parameter passed implicitly when function is invoked
- consider the following signature of `suspend` function

```
suspend fun <T> CompletableFuture<T>.await(): T =
```

- its actual implementation after compiler transformation has the following form

```
fun <T> CompletableFuture<T>.await(continuation: Continuation<T>): Any? =
```

Continuation interface

- `context` represents an arbitrary user-defined context that is associated with the coroutine

```
interface Continuation<in T> {
    val context: CoroutineContext
    fun resumeWith(result: Result<T>)
}
```

Continuation interface

- `context` represents an arbitrary user-defined context that is associated with the coroutine
- `resumeWith` function is a completion callback that is used to report on coroutine completion:

```
interface Continuation<in T> {
    val context: CoroutineContext
    fun resumeWith(result: Result<T>)
}
```

Kotlin

Continuation interface

- `context` represents an arbitrary user-defined context that is associated with the coroutine
- `resumeWith` function is a completion callback that is used to report on coroutine completion:
  - a success - with a value

```kotlin
interface Continuation<in T> {
    val context: CoroutineContext
    fun resumeWith(result: Result<T>)
}
```

Continuation interface

- `context` represents an arbitrary user-defined context that is associated with the coroutine
- `resumeWith` function is a completion callback that is used to report on coroutine completion:
  - a success - with a value
  - a failure - with an exception

```kotlin
interface Continuation<in T> {
    val context: CoroutineContext
    fun resumeWith(result: Result<T>)
}
```

Continuation interface

- `context` represents an arbitrary user-defined context that is associated with the coroutine
- `resumeWith` function is a completion callback that is used to report on coroutine completion:
  - a success - with a value
  - a failure - with an exception
- there are two extension functions defined for convenience

```kotlin
interface Continuation<in T> {
    val context: CoroutineContext
    fun resumeWith(result: Result<T>)
}
```

```kotlin
fun <T> Continuation<T>.resume(value: T) { .. }

fun <T> Continuation<T>.resumeWithException(exception: Throwable) { .. }
```

Kotlin

How the typical implementation looks like?

```
suspend fun <T> CompletableFuture<T>.await(): T =
    suspendCoroutine { cont: Continuation<T> ->
        whenComplete { result, exception ->
            if (exception == null) cont.resume(result)
            else cont.resumeWithException(exception)
        }
    }
```

How the typical implementation looks like?

```
suspend fun <T> CompletableFuture<T>.await(): T =
    suspendCoroutine { cont: Continuation<T> ->
        (this as CompletableFuture<T>).whenComplete(
            object : BiConsumer<T, Throwable?> {
                override fun accept(result: T, exception: Throwable?) {
                    if (exception == null) cont.resume(result)
                    else cont.resumeWithException(exception)
                }
            }
        )
    }
```

Kotlin

How the typical implementation looks like?

```kotlin
suspend fun <T> CompletableFuture<T>.await(): T =
    suspendCoroutine { cont: Continuation<T> ->
        (this as CompletableFuture<T>).whenComplete(
            object : BiConsumer<T, Throwable?> {
                override fun accept(result: T, exception: Throwable?) {
                    if (exception == null) cont.resume(result)
                    else cont.resumeWithException(exception)
                }
            }
        )
    }
```

- we use the callback interface to resume the continuation

How the typical implementation looks like?

```kotlin
suspend fun <T> CompletableFuture<T>.await(): T =
    suspendCoroutine { cont: Continuation<T> ->
        (this as CompletableFuture<T>).whenComplete(
            object : BiConsumer<T, Throwable?> {
                override fun accept(result: T, exception: Throwable?) {
                    if (exception == null) cont.resume(result)
                    else cont.resumeWithException(exception)
                }
            }
        )
    }
```

- we use the callback interface to resume the continuation
- to actually suspend execution function must invoke other suspending function - `await` implementation invokes a suspending function `suspendCoroutine`

State machines as perfect solution

- it's crucial to implement coroutines efficiently, i.e. create as few classes and objects as possible

Kotlin

State machines as perfect solution

- it's crucial to implement coroutines efficiently, i.e. create as few classes and objects as possible
- compiler creates a single instance of class that may have any number of suspension points

Kotlin

State machines as perfect solution

- it's crucial to implement coroutines efficiently, i.e. create as few classes and objects as possible
- compiler creates a single instance of class that may have any number of suspension points
- general idea - suspending function is a state machine

Kotlin

State machines as perfect solution

- it's crucial to implement coroutines efficiently, i.e. create as few classes and objects as possible
- compiler creates a single instance of class that may have any number of suspension points
- general idea - suspending function is a state machine
- states corresponds to suspension points

Kotlin

State machines as perfect solution

- it's crucial to implement coroutines efficiently, i.e. create as few classes and objects as possible
- compiler creates a single instance of class that may have any number of suspension points
- general idea - suspending function is a state machine
- states corresponds to suspension points

```
suspend fun stateMachine() {
    val a = a()
    val y = foo(a).await() // 1st suspension point
    b()
    val z = bar(a, y).await() // 2nd suspension point
    c(z)
}
```

State machines as perfect solution

- three states for the given block of code:

State machines as perfect solution

- three states for the given block of code:
  1. initial (before any suspension point)

Kotlin

State machines as perfect solution

- three states for the given block of code:
  1. initial (before any suspension point)
  2. after the first suspension point

State machines as perfect solution

- three states for the given block of code:
  1. initial (before any suspension point)
  2. after the first suspension point
  3. after the second suspension point

State machines as perfect solution

- three states for the given block of code:
  1. initial (before any suspension point)
  2. after the first suspension point
  3. after the second suspension point
- each of them being an entry point to continuation of this block

State machines as perfect solution

- three states for the given block of code:
  1. initial (before any suspension point)
  2. after the first suspension point
  3. after the second suspension point
- each of them being an entry point to continuation of this block
- which can be seen as `switch` statement with label to each suspension point

Kotlin

State machines as perfect solution

- three states for the given block of code:
  1. initial (before any suspension point)
  2. after the first suspension point
  3. after the second suspension point
- each of them being an entry point to continuation of this block
- which can be seen as `switch` statement with label to each suspension point
- the execution is continued with `resumeWith` method of coroutine which selects the current branch

Kotlin

State machines as perfect solution

- three states for the given block of code:
  1. initial (before any suspension point)
  2. after the first suspension point
  3. after the second suspension point
- each of them being an entry point to continuation of this block
- which can be seen as `switch` statement with label to each suspension point
- the execution is continued with `resumeWith` method of coroutine which selects the current branch
- local variables are generated as class fields of anonymous class

Kotlin

State machines implementation

Pseudo-code corresponding to JVM bytecode created for suspending function:

```
class <anonymous_for_state_machine> extends SuspendLambda<...> {
    // current state of the state machine
    int label = 0

    // local variables of the coroutine
    A a = null
    Y y = null

    void resumeWith(Object result) {
        if (label == 0) goto L0
        if (label == 1) goto L1
        if (label == 2) goto L2
        else throw IllegalStateException()
        // each label code
```

Kotlin

State machines implementation

```
        // local variables and state switch
      L0:
        // result is expected to be `null` at this invocation
        a = a()
        label = 1
        result = foo(a).await(this) // 'this' is passed as a continuation
        if (result == COROUTINE_SUSPENDED) return // return if await had suspended
      L1:
        // external code has resumed this coroutine passing the result of .await()
        y = (Y) result
        b()
        label = 2
        result = bar(a, y).await(this) // 'this' is passed as a continuation
        if (result == COROUTINE_SUSPENDED) return // return if await had suspended
      L2:
        // external code has resumed this coroutine passing the result of .await()
        Z z = (Z) result
        c(z)
        label = -1 // No more steps are allowed
        return
    }
}
```

Generators

- generators with `yield` can be easily implemented using `suspend` functions

Generators

- generators with `yield` can be easily implemented using `suspend` functions
- and that's how the Kotlin std-lib does, providing `Sequence<T>` abstraction with `yield` as a function of `SequenceScope<T>`

Generators

- generators with `yield` can be easily implemented using `suspend` functions
- and that's how the Kotlin std-lib does, providing `Sequence<T>` abstraction with `yield` as a function of `SequenceScope<T>`
- they can represent potentially infinite lists (like in Haskell)

Kotlin

Generators

- generators with `yield` can be easily implemented using `suspend` functions
- and that's how the Kotlin std-lib does, providing `Sequence<T>` abstraction with `yield` as a function of `SequenceScope<T>`
- they can represent potentially infinite lists (like in Haskell)
- or just in general lazily computed sequences

Generators

- generators with `yield` can be easily implemented using `suspend` functions
- and that's how the Kotlin std-lib does, providing `Sequence<T>` abstraction with `yield` as a function of `SequenceScope<T>`
- they can represent potentially infinite lists (like in Haskell)
- or just in general lazily computed sequences
- the strength of them is supporting arbitrary control flow, with `for`, `when`, `try`/`catch` etc.

Generators - example Fibonacci sequence

```
fun fibonacci() = sequence {
    yield(1)
    var curr = 1
    var next = 1
    while (true) {
        yield(next)
        val tmp = curr + next
        curr = next
        next = tmp
    }
}
```

Generators - example Fibonacci sequence

```
fun fibonacci() = sequence {
    yield(1)
    var curr = 1
    var next = 1
    while (true) {
        yield(next)
        val tmp = curr + next
        curr = next
        next = tmp
    }
}

fun main() = fibonacci().take(42).forEach(::println)
```

Initial idea - extra DSL

```
fun processRequest() = async {
    val request = await(receiveRequestPromise())
    val processed = await(handleRequestPromise(request))
    sendResponse(processed)
}
```

Kotlin

Initial idea - extra DSL

```
fun processRequest() = async {
    val request = await(receiveRequestPromise())
    val processed = await(handleRequestPromise(request))
    sendResponse(processed)
}
```

- original prototype defined the library functions, similar to keywords from e.g. C#, Typescript, Dart, Python

Kotlin

Initial idea - extra DSL

```kotlin
fun processRequest() = async {
    val request = await(receiveRequestPromise())
    val processed = await(handleRequestPromise(request))
    sendResponse(processed)
}
```

- original prototype defined the library functions, similar to keywords from e.g. C#, Typescript, Dart, Python
- but the compiler needed to know that `await` can be suspended, so it was marked with `suspend` modifier

Kotlin

Initial idea - extra DSL

```kotlin
fun processRequest() = async {
    val request = await(receiveRequestPromise())
    val processed = await(handleRequestPromise(request))
    sendResponse(processed)
}
```

- original prototype defined the library functions, similar to keywords from e.g. C#, Typescript, Dart, Python
- but the compiler needed to know that `await` can be suspended, so it was marked with `suspend` modifier
- so the decision was to remove the need for `await` call

Initial idea - extra DSL

```kotlin
fun processRequest() = async {
    val request = await(receiveRequestPromise())
    val processed = await(handleRequestPromise(request))
    sendResponse(processed)
}
```

- original prototype defined the library functions, similar to keywords from e.g. C#, Typescript, Dart, Python
- but the compiler needed to know that `await` can be suspended, so it was marked with `suspend` modifier
- so the decision was to remove the need for `await` call
- which led to adding `suspend` modifier to functions instead of using `async` builder

Initial idea - extra DSL

- there was some fear as no other language resigned from `async`/`await`

Initial idea - extra DSL

- there was some fear as no other language resigned from `async/await`
- actually in Go language nobody is worried about the not visible suspension points

Initial idea - extra DSL

- there was some fear as no other language resigned from `async/await`
- actually in Go language nobody is worried about the not visible suspension points

```
suspend fun go() = coroutineScope {
    go { printDelayed("go: fizz") }
    printDelayed("go: buzz")
}
```

Initial idea - extra DSL

- there was some fear as no other language resigned from `async/await`
- actually in Go language nobody is worried about the not visible suspension points

```
suspend fun go() = coroutineScope {
    go { printDelayed("go: fizz") }
    printDelayed("go: buzz")
}
```

```
suspend fun kt() = coroutineScope {
    launch { printDelayed("kt: fizz") }
    printDelayed("kt: buzz")
}
```

Kotlin

Initial idea - extra DSL

- there was some fear as no other language resigned from `async/await`
- actually in Go language nobody is worried about the not visible suspension points

```kotlin
suspend fun go() = coroutineScope {
    go { printDelayed("go: fizz") }
    printDelayed("go: buzz")
}
```

```kotlin
suspend fun kt() = coroutineScope {
    launch { printDelayed("kt: fizz") }
    printDelayed("kt: buzz")
}
```

```kotlin
suspend fun printDelayed(msg: Any, times: Int = 5) {
    for (i in 1..times) {
        delay(1_000)
        println(msg)
    }
}
```

Kotlin

Context and Scope of execution

```kotlin
suspend fun implicitScope(): Unit = coroutineScope {
    launch(context = Default) { say("hello") }
    launch(context = IO) { say("world") }
}
```

Context and Scope of execution

```
suspend fun implicitScope(): Unit = coroutineScope {
    launch(context = Default) { say("hello") }
    launch(context = IO) { say("world") }
}
```

- coroutines execute in some context represented by the `CoroutineContext` type

Kotlin

Context and Scope of execution

```
suspend fun implicitScope(): Unit = coroutineScope {
    launch(context = Default) { say("hello") }
    launch(context = IO) { say("world") }
}
```

- coroutines execute in some context represented by the `CoroutineContext` type
- it contains the `CoroutineDispatcher` which delegates the coroutine to some thread of execution

Kotlin

Context and Scope of execution

```kotlin
suspend fun implicitScope(): Unit = coroutineScope {
    launch(context = Default) { say("hello") }
    launch(context = IO) { say("world") }
}
```

- coroutines execute in some context represented by the `CoroutineContext` type
- it contains the `CoroutineDispatcher` which delegates the coroutine to some thread of execution
- it's inherited from the parent `CoroutineScope` by default

Kotlin

Context and Scope of execution

```kotlin
suspend fun implicitScope(): Unit = coroutineScope {
    launch(context = Default) { say("hello") }
    launch(context = IO) { say("world") }
}
```

- coroutines execute in some context represented by the `CoroutineContext` type
- it contains the `CoroutineDispatcher` which delegates the coroutine to some thread of execution
- it's inherited from the parent `CoroutineScope` by default
- and all coroutine builders like `launch` and `async` accept optional `CoroutineContext` to explicitly specify the dispatcher for the new coroutine context

Context and Scope of execution

```
suspend fun explicitScope(): Unit = coroutineScope {
    val scope: CoroutineScope = this
    scope.launch(context = Default) { say("hello") }
    scope.launch(context = IO) { say("world") }
}
```

- coroutines execute in some context represented by the `CoroutineContext` type
- it contains the `CoroutineDispatcher` which delegates the coroutine to some thread of execution
- it's inherited from the parent `CoroutineScope` by default
- and all coroutine builders like `launch` and `async` accept optional `CoroutineContext` to explicitly specify the dispatcher for the new coroutine context

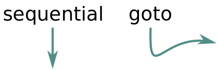
Kotlin

What is structured concurrency?

- compare understanding concurrency to understanding program control flow

What is structured concurrency?

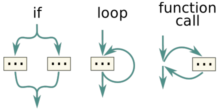
- compare understanding concurrency to understanding program control flow
- with `goto` instruction it can start being hard to understand



source: https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/

What is structured concurrency?

- compare understanding concurrency to understanding program control flow
- with `goto` instruction it can start being hard to understand
- but when we limit ourselves to known structures, it's much more obvious and not that limited



source: https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/

Kotlin

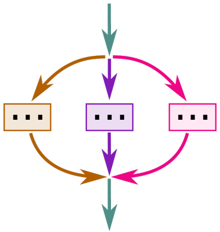
What is structured concurrency?

- compare understanding concurrency to understanding program control flow
- with `goto` instruction it can start being hard to understand
- but when we limit ourselves to known structures, it's much more obvious and not that limited
- in concurrent programming model we have the same problem with tasks

Kotlin

What is structured concurrency?

- compare understanding concurrency to understanding program control flow
- with `goto` instruction it can start being hard to understand
- but when we limit ourselves to known structures, it's much more obvious and not that limited
- in concurrent programming model we have the same problem with tasks
- introducing some "structures" to keep everything in consistent state helps in understanding what may happen



source: https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/

Kotlin

`launch` and `async` coroutine builders

- `launch` produces a `Job`, while `async` produces a `Deferred<T>`

launch and async coroutine builders

- `launch` produces a `Job`, while `async` produces a `Deferred<T>`
- they start a new coroutine without blocking the current one

`launch` and `async` coroutine builders

- `launch` produces a `Job`, while `async` produces a `Deferred<T>`
- they start a new coroutine without blocking the current one
- they can be called only in context of some `CoroutineScope` which is "responsible" for its execution

`launch` and `async` coroutine builders

- `launch` produces a `Job`, while `async` produces a `Deferred<T>`
- they start a new coroutine without blocking the current one
- they can be called only in context of some `CoroutineScope` which is "responsible" for its execution
- coroutine is cancelled when the returned `Job` is cancelled

launch and async coroutine builders

- launch produces a Job, while async produces a Deferred<T>
- they start a new coroutine without blocking the current one
- they can be called only in context of some CoroutineScope which is “responsible” for its execution
- coroutine is cancelled when the returned Job is cancelled
- but still we need to manually check for cancellation in long-running tasks

Structured concurrency in Kotlin

- `CoroutineScope` can be seen a structure in which the control flows of concurrent tasks are merged

Structured concurrency in Kotlin

- `CoroutineScope` can be seen a structure in which the control flows of concurrent tasks are merged
- `launch` and `async` corresponds to creating new execution branches

Structured concurrency in Kotlin

- `CoroutineScope` can be seen a structure in which the control flows of concurrent tasks are merged
- `launch` and `async` corresponds to creating new execution branches
- parent always waits for children completion

Structured concurrency in Kotlin

- `CoroutineScope` can be seen a structure in which the control flows of concurrent tasks are merged
- `launch` and `async` corresponds to creating new execution branches
- parent always waits for children completion
- there's no place for loosing some resource

Structured concurrency in Kotlin

- `CoroutineScope` can be seen a structure in which the control flows of concurrent tasks are merged
- `launch` and `async` corresponds to creating new execution branches
- parent always waits for children completion
- there's no place for loosing some resource
- there's no exception lost - they're propagated

launch running examples

```kotlin
suspend fun justSuspensionPoints(): Unit = coroutineScope {
    sayA()
    sayB()
}
```

Kotlin

`launch` running examples

```kotlin
suspend fun justSuspensionPoints(): Unit = coroutineScope {
    sayA()
    sayB()
}
```

- executes `sayA` and `sayB` sequentially

launch running examples

```kotlin
suspend fun justSuspensionPoints(): Unit = coroutineScope {
    sayA()
    sayB()
}
```

- executes `sayA` and `sayB` sequentially
- finishes `justSuspensionPoints` when both jobs are finished

launch running examples

```
suspend fun launchFirstInScope(): Unit = coroutineScope {
    launch { sayA() }
    sayB()
}
```

launch running examples

```kotlin
suspend fun launchFirstInScope(): Unit = coroutineScope {
    launch { sayA() }
    sayB()
}
```

- executes sayA and sayB concurrently

Kotlin

launch running examples

```
suspend fun launchFirstInScope(): Unit = coroutineScope {
    launch { sayA() }
    sayB()
}
```

- executes `sayA` and `sayB` concurrently
- finishes `launchFirstInScope` when both jobs are finished

launch running examples

```kotlin
suspend fun launchBothInScope(): Unit = coroutineScope {
    launch { sayA() }
    launch { sayB() }
}
```

launch running examples

```
suspend fun launchBothInScope(): Unit = coroutineScope {
    launch { sayA() }
    launch { sayB() }
}
```

- executes `sayA` and `sayB` concurrently

launch running examples

```
suspend fun launchBothInScope(): Unit = coroutineScope {
    launch { sayA() }
    launch { sayB() }
}
```

- executes `sayA` and `sayB` concurrently
- finishes `launchBothInScope` when both jobs are finished

Kotlin

launch running examples

```kotlin
suspend fun launchWithException(): Unit = coroutineScope {
    launch { error("illegal to sayA") }
    launch { sayB() }
}
```

launch running examples

```kotlin
suspend fun launchWithException(): Unit = coroutineScope {
    launch { error("illegal to sayA") }
    launch { sayB() }
}
```

- executes sayA and sayB concurrently

launch running examples

```kotlin
suspend fun launchWithException(): Unit = coroutineScope {
    launch { error("illegal to sayA") }
    launch { sayB() }
}
```

- executes `sayA` and `sayB` concurrently
- when `sayA` fails, `sayB` is cancelled as well

launch running examples

```kotlin
suspend fun launchInSupervisorWithException(): Unit = supervisorScope {
    launch { error("illegal to sayA") }
    launch { sayB() }
}
```

launch running examples

```kotlin
suspend fun launchInSupervisorWithException(): Unit = supervisorScope {
    launch { error("illegal to sayA") }
    launch { sayB() }
}
```

- executes sayA and sayB concurrently

Kotlin

launch running examples

```kotlin
suspend fun launchInSupervisorWithException(): Unit = supervisorScope {
    launch { error("illegal to sayA") }
    launch { sayB() }
}
```

- executes sayA and sayB concurrently
- when sayA fails, sayB is executed until it finishes

Kotlin

Higher-order function

- introducing coroutine brings advantage of no extra need for higher-order combinators like `flatMap`

Higher-order function

- introducing coroutine brings advantage of no extra need for higher-order combinators like `flatMap`
- it's enough to use the `suspend` modifier for lambda parameter

Higher-order function

- introducing coroutine brings advantage of no extra need for higher-order combinators like `flatMap`
- it's enough to use the `suspend` modifier for lambda parameter

```kotlin
suspend fun main() = retry(afterMillis = 10) { say("hello") }

suspend fun retry(afterMillis: Long, action: suspend () -> Unit) {
    do {
        try {
            return action()
        } catch (e: Exception) {
            println(e.stackTrace)
        }
        delay(afterMillis)
    } while (coroutineContext.isActive)
}
```

Higher-level APIs

`Channel<T>` in Kotlin

- is implemented in Kotlin as a library, conceptually very similar to `BlockingQueue`

Kotlin

Channel<T> in Kotlin

- is implemented in Kotlin as a library, conceptually very similar to BlockingQueue
- implements SendChannel<T> and ReceiveChannel<T>

```kotlin
interface SendChannel<T> {
    suspend fun send(value: T)
    fun close()
}

interface ReceiveChannel<T> {
    suspend fun receive(): T
    suspend operator fun iterator(): ReceiveIterator<T>
}
```

Channel<T> in Kotlin

- is implemented in Kotlin as a library, conceptually very similar to `BlockingQueue`
- implements `SendChannel<T>` and `ReceiveChannel<T>`
- `send` suspends when the channel buffer is full, while `receive` suspends when the buffer is empty

Channel<T> in Kotlin

- is implemented in Kotlin as a library, conceptually very similar to `BlockingQueue`
- implements `SendChannel<T>` and `ReceiveChannel<T>`
- `send` suspends when the channel buffer is full, while `receive` suspends when the buffer is empty
- channels are hot - there's a coroutine on the other side of the channel that produces values, so we cannot just drop a reference to the ReceiveChannel, because the producer is going to be suspended forever waiting for a consumer, wasting memory resources, open network connections, etc.

Kotlin

`Flow<T>` in Kotlin

- represents the stream of values that are being computed asynchronously

Kotlin

`Flow<T>` in Kotlin

- represents the stream of values that are being computed asynchronously
- can be used just like a Sequence$<$T$>$ type for synchronously computed values

Kotlin

Flow<T> in Kotlin

- represents the stream of values that are being computed asynchronously
- can be used just like a Sequence<T> type for synchronously computed values
- flows are cold streams similar to sequences

Flow<T> in Kotlin

- represents the stream of values that are being computed asynchronously
- can be used just like a Sequence$<$T$>$ type for synchronously computed values
- flows are cold streams similar to sequences
- the code inside a flow builder does not run until the flow is collected

Kotlin

Images downloader - goals

- have single abstraction responsible for caching the results

Kotlin

Images downloader - goals

- have single abstraction responsible for caching the results
- have multiple workers responsible for doing the job

Images downloader - goals

- have single abstraction responsible for caching the results
- have multiple workers responsible for doing the job
- communicate between these abstractions in safely way

Kotlin

Images downloader - goals

- have single abstraction responsible for caching the results
- have multiple workers responsible for doing the job
- communicate between these abstractions in safely way
- structure our concurrency model to be sure what's going on

Kotlin

Images downloader - downloader

```kotlin
fun CoroutineScope.downloader(
    receiveStringUrlChannel: ReceiveChannel<String>,
    sendResultByteArrayChannel: SendChannel<ByteArray>,
    sendToWorkerChannel: SendChannel<Url>,
    receiveFromWorkerChannel: ReceiveChannel<DownloadedData>,
) = launch {
    val cached = mutableMapOf<Url, ByteArray>()
    while (isActive) {
        select {
            receiveFromWorkerChannel.onReceive { (url, data) ->
                cached[url] = data
                sendResultByteArrayChannel.send(data)
            }
            receiveStringUrlChannel.onReceive {
                val url = Url(it)
                val data = cached[url]
                if (data == null) sendToWorkerChannel.send(url)
                else sendResultByteArrayChannel.send(data)
            }
        }
    }
}
```

Images downloader - worker

```
fun CoroutineScope.downloadWorker(
    receive: ReceiveChannel<Url>,
    sendData: SendChannel<DownloadedData>,
) = launch {
    val client = HttpClient(CIO)
    for (url in receive) {
        val data = client.get(url)
        val bytes = data.readBytes()
        val downloaded = DownloadedData(url, bytes)
        sendData.send(downloaded)
    }
}
```

Kotlin

Images downloader - common scope

```kotlin
fun CoroutineScope.processUrls(
    receiveStringUrlChannel: ReceiveChannel<String>,
    sendResultByteArrayChannel: SendChannel<ByteArray>,
) {
    val urls = Channel<Url>(capacity = 1)
    val data = Channel<DownloadedData>(capacity = 1)
    repeat(N_WORKERS) { downloadWorker(urls, data) }
    downloader(receiveStringUrlChannel, sendResultByteArrayChannel, urls, data)
}
```

Kotlin

Images downloader - Jetpack Compose demo

```kotlin
fun main() = application {
    val scope = rememberCoroutineScope()
    val urlChannel = Channel<String>(capacity = 1)
    val imageDataChannel = Channel<ByteArray>(capacity = 1)
    LaunchedEffect(Unit) { processUrls(urlChannel, imageDataChannel) }
    // application ui
```

Still confused? Check out these places

1. Roman Elizarov - Structured concurrency ▸ Watch
2. KotlinConf 2017 - Introduction to Coroutines by Roman Elizarov ▸ Watch
3. KotlinConf 2017 - Deep Dive into Coroutines on JVM by Roman Elizarov ▸ Watch
4. KotlinConf 2018 - Kotlin Coroutines in Practice by Roman Elizarov ▸ Watch
5. KotlinConf 2019 - Asynchronous Data Streams with Kotlin Flow by Roman Elizarov ▸ Watch
6. Notes on structured concurrency, or: Go statement considered harmful ▸ Read
7. Revisiting Coroutines, Ana Moura & Roberto Ierusalimschy ▸ Read

Time to `suspend`

Thank you for your attention!