Threading lightly in Kotlin

EXPLORING COROUTINES

Your host today

Vasco Veloso

Working in software development since 1996.

Currently interested in designing software systems and giving back to the community.

Author and speaker.

Portuguese living in the Netherlands.

linkedin.com/in/vascoveloso

@vveloso

Threads enjoy running

Preemptive multitasking.

Suspension of execution.

Stateful instances.

Coroutines enjoy sharing

Cooperative multitasking.

Suspension of execution.

Stateful instances.

Threads enjoy running

```
1 thread Producer
2  while queue is not full
3       add new item to queue
```

```
1 thread Consumer
2  while queue is not empty
3       remove item from queue
```

Coroutines enjoy sharing

```
1 coroutine Producer
2   while queue is not full
3     add new item to queue
4   yield to Consumer
```

```
1 coroutine Consumer
2   while queue is not empty
3     remove item from queue
4   yield to Producer
```

Subroutines dream of coroutines

```
1 subroutine Producer
2  while queue is not full
3    add new item to queue
4  call Consumer
```

```
1 subroutine Consumer
2  while queue is not empty
3    remove item from queue
4  call Producer
```

What's on the repository?

Let's take a look at

https://github.com/vveloso/kotlin-concurrency

Declaring coroutines in Kotlin

coroutines/workshop/exercise1/first-coroutine.kt

```
fun main() = runBlocking {
    launch {
        delay(1000) // busy busy work
    }
    busyBee()
}

suspend fun busyBee() {
    delay(1000) // also busy
}
```

Cost overview

THREADS

Consume memory.

Hold up kernel resources.

Require context switching operations.

COROUTINES

Consume memory.

Context switching is not mandatory.

Cost overview

coroutines/workshop/exercise2/cost-overview.kt

Try creating 100 000 threads.

Try creating 100 000 coroutines.

Which one worked better and why?

Cooperation is important

Coroutines can allow other coroutines to run only at suspension points:

- `yield()`
- `delay()`
- Call to any other coroutine.

Cooperation is important

coroutines/workshop/exercise3/cooperation.kt

- Run 10 threads that are greedy and never yield.
- Run 10 coroutines that are greedy and never yield.
- Run 10 coroutines that yield to one another (tip: yield).
- What happened?

Giving up on a coroutine

`launch` returns an object implementing the `Job` interface.

Interesting methods:

- `cancel()` - asks the job to stop ahead of time
- `join()` - suspends the current coroutine until the job completes

Cancellation is **cooperative**, it only happens at suspension points.

- `yield()`
- `isActive`

Giving up on a coroutine

coroutines/workshop/exercise4/cancellation.kt

Measure how long the calculation takes on your machine.

Cancel the coroutine after half that time and wait until the job completes.

Replace the explicit yield with a check of the property that tells if the coroutine is active. Repeat.

Remove both the explicit yield and the property check. Repeat. Did the coroutine cancel?

Giving up on a coroutine w/ timeout

When the caller needs to wait no longer than a specific amount of time, use:

- `withTimeout()` - throws a TimeoutCancellationException on timeout
- `withTimeoutOrNull()` - returns null on timeout

Going explicitly concurrent

One coroutine can launch other coroutines on a concurrent manner but it must do so explicitly.

Using the async function...

```
val one = async { doSomethingUsefulOne() }
```

… returns an instance of `Deferred`, which is also a `Job` so it is cancellable.

The most useful method from `Deferred` is:

- `await()` - awaits for completion without blocking and resumes when the result is available.

Going explicitly concurrent

coroutines/workshop/exercise5/explicit-async.kt

- Measure the time it takes to get one translation of Hello in one language.
- Get the translation for at least four languages and measure the time.
- Issue the requests concurrently, still measuring the time.
- Confirm they were faster.

Going explicitly concurrent

coroutines/workshop/example/structured-concurrency.kt

Problem:

- How do we cancel all concurrent tasks we launched with async if one throws an exception?

Solution:

- Structured concurrency.

Use `coroutineScope` to aggregate related concurrent coroutines in one single logical unit of work.

If something goes wrong in one of the coroutines and it throws an exception, all coroutines in the same scope are cancelled.

Thread affinity

Conceptually:

- A program running coroutines is single threaded.

In practice with Kotlin:

- It depends on the dispatcher.
- Better to think of it as free-threaded.

Thread affinity

coroutines/workshop/exercise6/thread-affinity.kt

- Try without specifying a dispatcher.
- Try specifying Dispatchers.**Default** in launch().
- Try specifying Dispatchers.Unconfined in launch().
- Conclusions?

Thread affinity

What are the consequences of a free-threaded model?

Can we rely on traditional JRE thread synchronization mechanisms?

Thread affinity

Solutions to handle shared state or resources:

- Don't share.
- Use thread-safe data structures.
- Use channels.
- Run all coroutines that share state or resources in the same single-threaded dispatcher.
  - Use newSingleThreadContext() and withContext().
- Use kotlinx.coroutines.sync.Mutex.

Communicating between coroutines

Communication between coroutines is possible by using Channels:

- One coroutine sends objects through the channel.
- Another consumes objects from the channel.
- A Channel can be seen as a “special” queue.

We can fan-out or fan-in using channels: sending and receiving operations are fair.

The channel interface is `kotlinx.coroutines.channels.Channel<E>`.

Communicating between coroutines

There are two important operations in the Channel interface:

- `send()`
- `receive()`

A Channel also supports `close()`. This signals that no more items will be sent.

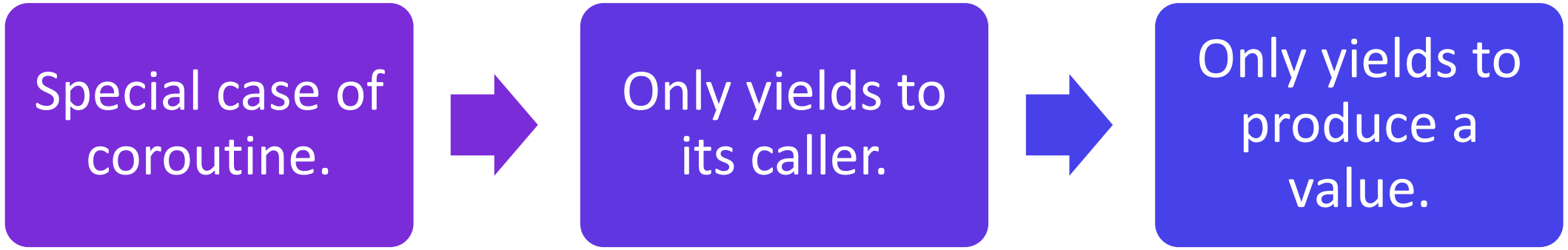
Communicating between coroutines

coroutines/workshop/exercise7/channels.kt

Have two coroutines communicating through a channel.

Try the experimental producer method.

Generator


Special case of coroutine.
Only yields to its caller.
Only yields to produce a value.

Flows are generators

```
fun foo(): Flow<Int> = flow { // flow builder
    for (i in 1..3) {
        emit(i) // emit next value
    }
}

foo().collect { value -> println(value) }
```

Flows are generators

Flows are data streams.

Flows are cold: collection does not start until the terminal operation is executed.

- collect, reduce, ...

Flows only suspend when the generator coroutine suspends.

Flows support intermediate operators.

- map, transform, take, zip, …

Flows are transparent to exceptions.

Flows complete when the data ends or with an exception.

Flows

coroutines/workshop/exercise8/flows.kt

Build a flow that provides some data items retrieved from one API.

Suggestions:

- Quotes (https://github.com/lukePeavey/quotable)
- Evil Insult Generator (https://evilinsult.com/)

Testing coroutines

`runBlocking()` can be used in unit tests.

`runBlockingTest()` is better suited for testing coroutines that contain delays: it advances time automatically.

- Available from the `org.jetbrains.kotlinx:kotlinx-coroutines-test` dependency.
- Can also provide a Main dispatcher for testing purposes.

Real-world usage: Spring WebFlux

Spring WebFlux allows us to leverage coroutines to build reactive Web applications: a reactive Web application needs a non-blocking server infrastructure with support for backpressure.

The problem with this kind of applications is that *nothing* is allowed to block.

So non-blocking versions of HTTP clients, database clients, other I/O, etc. are a must.

Coroutines have the same requirements! :)

Real-world usage: reference application

Build a Spring Boot Web MVC application based on Tomcat with one endpoint `/name` that returns three names.

The method handling the GET verb should thus return a list of strings.

Tip: use https://start.spring.io to get started.

Real-world usage: reactive application

Build a Spring Boot WebFlux application with one endpoint `/name` that also returns three names.

Being a WebFlux application, you may use a coroutine as the GET verb handler or it may return a `Flux` of strings.

Tip: use https://start.spring.io to continue.

Real-world usage: so what?

Use the provided Gatling script to measure performance.

- Script is in the master branch.

Compare with the reference application:

- How many threads does each application need? (tip: check with jconsole)
- What happens if you add a delay to the response in the reference application?
- And in the reactive application?

Free exercise

Build an application that...

- Lists the whole cast of any movie.
  - See https://developers.themoviedb.org (registration is required)
- Works asynchronously with a database.
  - See https://r2dbc.io/ for existing non-blocking database driver implementations.
- Does everything or something else but non-blocking. :-)