Kotlin Coroutines

Introduction

Coroutines are code the execution of which can be suspended and then resumed later.

The main purpose of coroutines is to run asynchronous code without callbacks, in direct code style.

A coroutine may be resumed by the same thread that was running it before suspension or by the other threads.

A coroutine cannot be suspended in any point during execution, only in certain points. These points are called ‘suspension points’.

Coroutines are not lightweight threads!

Suspending functions

Functions with special modifier ‘suspend’. These functions are functions the execution of which can be suspended/resumed. The Kotlin compiler needs the ‘suspend’ modified because these are special functions now and the compiler changes the code so the functions can be suspended/resumed.

Continuation interface

Continuation<T> interface is used to resume the suspended function from the point where it was suspended.

State machine

Kotlin compiler transforms each function with a ‘suspended’ modifier. After the transformation, the functions’ execution can be suspended and then resumed in certain points not any points during execution.

For each suspend function the kotlin compiler creates a state machine

Before:

```
suspend fun loginUser(userId: String, password: String): User {
  val user = userRemoteDataSource.logUserIn(userId, password)
  val userDb = userLocalDataSource.logUserIn(user)
  return userDb
}
```

After:

```
fun loginUser(userId: String?, password: String?, completion:
Continuation<Any?>) {

    class LoginUserStateMachine(
        // completion parameter is the callback to the function that
called loginUser
        completion: Continuation<Any?>
    ): CoroutineImpl(completion) {
        // objects to store across the suspend function
        var user: User? = null
        var userDb: UserDb? = null

        // Common objects for all CoroutineImpl
        var result: Any? = null
        var label: Int = 0

        // this function calls the loginUser again to trigger the
        // state machine (label will be already in the next state) and
        // result will be the result of the previous state's
computation
        override fun invokeSuspend(result: Any?) {
            this.result = result
            loginUser(null, null, this)
        }
    }

    val continuation = completion as? LoginUserStateMachine ?:
LoginUserStateMachine(completion)

    when(continuation.label) {
        0 -> {
            // Checks for failures
            throwOnFailure(continuation.result)
            // Next time this continuation is called, it should go to
state 1
            continuation.label = 1
            // The continuation object is passed to logUserIn to resume
            // this state machine's execution when it finishes
            userRemoteDataSource.logUserIn(userId!!, password!!,
continuation)
        }
        1 -> {
            // Checks for failures
            throwOnFailure(continuation.result)
            // Gets the result of the previous state
            continuation.user = continuation.result as User
            // Next time this continuation is called, it should go to
state 2
            continuation.label = 2
            // The continuation object is passed to logUserIn to resume
            // this state machine's execution when it finishes
```

```
            userLocalDataSource.logUserIn(continuation.user,
continuation)
        }
        2 -> {
            // Checks for failures
            throwOnFailure(continuation.result)
            // Gets the result of the previous state
            continuation.userDb = continuation.result as UserDb
            // Resumes the execution of the function that called this
one
            continuation.cont.resume(continuation.userDb)
        }
        else -> throw IllegalStateException(/* ... */)
    }
}
```

References:

- https://habr.com/ru/articles/659699/
- https://www.youtube.com/watch?v=rB5Q3y73FTo