Introduction to Kotlin Coroutines

Michael Joshua

2024

Introduction to Kotlin Coroutines

Coroutines are a feature in Kotlin that simplifies asynchronous programming. They allow you to write code that is sequential and easy to read but runs asynchronously and efficiently.

Coroutine Basics: Launch, Async, and RunBlocking

1. **Launch:** This function starts a new coroutine without blocking the current thread. It's used when you don't need a result from the coroutine.

```
import kotlinx.coroutines.*

fun main() {
    GlobalScope.launch {
        delay(1000L)  // Non-blocking delay for 1 second
        println("Hello from Coroutine!")
    }
    println("Hello from Main Thread!")
    Thread.sleep(2000L)  // Blocking delay for 2 seconds to keep JVM alive
}
```

In this example, "Hello from Coroutine!" is printed after "Hello from Main Thread!" because the coroutine is delayed.

2. Async: This function starts a new coroutine and returns a Deferred object, which represents a future result.

```
import kotlinx.coroutines.*

fun main() {
    GlobalScope.launch {
        val result = async { computeResult() }
        println("Computed result: ${result.await()}")
    }
    Thread.sleep(2000L)
}

suspend fun computeResult(): Int {
    delay(1000L)
    return 42
}
```

Here, computeResult is a suspending function that returns 42 after a delay, and await waits for the result.

3. RunBlocking: Bridges non-coroutine and coroutine worlds, starting a top-level main coroutine.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(1000L)
        println("Hello from Coroutine!")
    }
    println("Hello from Main Thread!")
}
```

runBlocking ensures that "Hello from Coroutine!" is printed before the program exits.

Coroutine Context and Dispatchers

Coroutines run in a specific context which includes a dispatcher to determine the thread on which the coroutine runs.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
   launch(Dispatchers.IO) { println("IO: ${Thread.currentThread().name}") }
   launch(Dispatchers.Default) { println("Default: ${Thread.currentThread().name}") }
   launch(Dispatchers.Main) { println("Main: ${Thread.currentThread().name}") }
}
```

Different dispatchers (IO, Default, Main) are used to specify the appropriate thread for different types of tasks.

Exception Handling in Coroutines

Coroutines handle exceptions similarly to regular code but can also propagate them through asynchronous boundaries.

```
import kotlinx.coroutines.*
fun main() = runBlocking {
   val job = GlobalScope.launch {
      println("Throwing exception from coroutine")
      throw IllegalArgumentException()
   }
   job.join()
   println("Joined failed job")
   val deferred = GlobalScope.async {
      println("Throwing exception from async")
      throw ArithmeticException()
      42
   }
   try {
      deferred.await()
   } catch (e: ArithmeticException) {
      println("Caught ArithmeticException")
   }
}
```

This example shows how to handle exceptions in launch and async coroutines.

Structured Concurrency

Structured concurrency ensures that coroutines are bound to a specific scope and are cancelled when the scope is cancelled.

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    launch {
        delay(1000L)
        println("Task from runBlocking")
    }

    coroutineScope {
        launch {
            delay(2000L)
            println("Task from nested launch")
        }

        delay(500L)
        println("Task from coroutine scope")
    }

    println("Coroutine scope is over")
}
```

coroutineScope creates a scope that ensures all its child coroutines complete before it ends.

Suspending Functions

Suspending functions can be paused and resumed, making them ideal for long-running operations.

```
import kotlinx.coroutines.*

suspend fun doSomething() {
    delay(1000L)
    println("Doing something")
}
fun main() = runBlocking {
    launch { doSomething() }
}
```

Here, doSomething is a suspending function that pauses for a second before printing.

Non-blocking Delays

delay is used in coroutines to pause without blocking the thread.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(1000L)
        println("Hello from Coroutine!")
    }
    println("Hello from Main Thread!")
}
```

Job Hierarchy

Jobs form a hierarchy, and cancelling a parent job cancels all its children.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val parentJob = launch {
        val childJob = launch {
            while (true) {
                println("Child is running")
                delay(500L)
            }
        }
        delay(2000L)
        println("Cancelling child job")
        childJob.cancel()
    }
    parentJob.join()
}
```

This example shows how a parent job can cancel its child job.

Channels and Flows

Channels allow for sending and receiving a stream of values between coroutines.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
    val channel = Channel<Int>()
    launch {
        for (x in 1..5) channel.send(x * x)
        channel.close()
    }
    repeat(5) { println(channel.receive()) }
    println("Done!")
}
```

Here, the channel is used to send and receive squared numbers.

Flow is a Kotlin feature for handling a stream of values sequentially.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
fun numbers(): Flow<Int> = flow {
    for (i in 1..5) {
        delay(1000L)
        emit(i)
    }
}
fun main() = runBlocking {
    launch {
        for (k in 1..5) {
            println("I'm not blocked $k")
            delay(1000L)
        }
    }
    numbers().collect { value -> println(value) }
}
```

Advanced Flow Operators

Flow operators like debounce help transform and manage data.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

suspend fun performRequest(request: Int): String {
    delay(1000L)
    return "response $request"
}

fun main() = runBlocking {
    val flow = (1..5).asFlow().onEach { delay(300L) }
    flow.debounce(500L)
        .map { request -> performRequest(request) }
        .collect { response -> println(response) }
}
```

This example demonstrates how debounce can control the rate of emitting values in a flow.

Combining Multiple Coroutines

You can combine results of multiple coroutines using operators like zip.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

suspend fun performRequest(request: Int): String {
    delay(1000L)
    return "response $request"
}

fun main() = runBlocking {
    val nums = (1..5).asFlow()
    val strs = nums.map { performRequest(it) }
    nums.zip(strs) { a, b -> "$a -> $b" }
        .collect { println(it) }
}
```

zip combines two flows into one, emitting pairs of values.

Callbacks and Coroutines

Convert callback-based APIs to coroutines using suspendCancellableCoroutine.

```
suspend fun fetchUser(id: String): User = suspendCancellableCoroutine {
continuation ->
    api.getUser(id, object : Callback<User> {
        override fun onResponse(call: Call<User>, response: Response<User>) {
            continuation.resume(response.body())
        }

        override fun onFailure(call: Call<User>, t: Throwable) {
            continuation.resumeWithException(t)
        }
    })
}
```

This example shows how to convert a callback API to a suspending function.

Coroutine Timeouts

```
import kotlinx.coroutines.*

suspend fun doSomething() {
    delay(3000L)
}

fun main() = runBlocking {
    try {
        withTimeout(2000L) {
            doSomething()
        }
    } catch (e: TimeoutCancellationException) {
        println("The task exceeded the timeout limit.")
    }
}
```

withTimeout cancels the task if it exceeds the specified time.

Coroutine Scopes and Supervision

Use SupervisorJob or supervisorScope for independent child coroutine cancellation.

```
import kotlinx.coroutines.*

suspend fun doSomething() {
    delay(1000L)
    throw Exception("Something went wrong.")
}

fun main() = runBlocking {
    val supervisor = SupervisorJob()
    with(CoroutineScope(coroutineContext + supervisor)) {
        val child1 = launch { doSomething() }
        val child2 = launch { delay(2000L); println("Coroutine 2 completed.") }
    }
    delay(3000L)
}
```

Shared Mutable State and Concurrency

Use Mutex or Atomic classes for safe access to shared mutable state.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.sync.*
var counter = 0
val mutex = Mutex()

suspend fun doWork() {
    mutex.withLock {
        counter++
    }
}
fun main() = runBlocking {
    val jobs = List(100) { launch { doWork() } }
    jobs.joinAll()
    println("Counter: $counter")
}
```

Mutex ensures that only one coroutine accesses counter at a time.

Cancellation and Timeouts

Handle coroutine cancellation and timeouts effectively using try-catch or finally.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job = launch {
        try {
            repeat(1000) { i ->
                println("Job: I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            println("Job: I'm running finally.")
        }
    }
    delay(1300L)
    println("Main: I'm tired of waiting!")
    job.cancelAndJoin()
    println("Main: Now I can quit.")
}
```

Kotlin Coroutines

P. Michael Joshua

2024