Scala asynchronous programming

Scala asynchronous programming enables you to write non-blocking code that can run operations concurrently, improving performance especially in I/O-bound or long-running tasks. The most common tool in Scala (including Scala 2) for async programming is the `Future` API combined with the `scala-async` library that introduces `async`/`await` syntax for clearer asynchronous code.

Key Concepts:

- Future: Represents a computation that will complete with a value or failure at some point.
- Promise: A writable, single-assignment container that completes its associated Future.
- ExecutionContext: A thread pool or executor where Futures run asynchronously.
- scala-async: A DSL allowing you to write asynchronous code in a direct style using `async` blocks and `await` expressions, making async code look like synchronous code.

https://www.youtube.com/watch?v=I7-hxTbpscU

Simple Scala Future Example

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Success, Failure}

object FutureExample extends App {
  val futureComputation = Future {
    Thread.sleep(1000)  // Simulate long computation
    42
  }

  futureComputation.onComplete {
    case Success(value) => println(s"Computation finished with
result: $value")
```

```
    case Failure(e) => println(s"Computation failed:
${e.getMessage}")
  }

  println("This prints immediately, not waiting for future.")
  Thread.sleep(1500) // Keep JVM alive to see async result
}
```

This snippet runs a task asynchronously and continues the program without blocking. The printed result comes when the `Future` completes.

Using `scala-async` for Cleaner Async Code

To use, add this dependency in your `build.sbt`:

```
libraryDependencies += "org.scala-lang.modules" %% "scala-async"
% "1.0.1"
```

Then:

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.async.Async.{async, await}

object AsyncAwaitExample extends App {
  def slowComputation: Future[Int] = Future {
    Thread.sleep(1000)
    10
  }

  def anotherSlowComputation(s: String): Future[Int] = Future {
    Thread.sleep(1500)
    s.length
  }
```

```
  val combined: Future[Int] = async {
    val first = await(slowComputation)              // waits
non-blockingly
    val second = await(anotherSlowComputation("Scala"))
    first + second
  }

  combined.onComplete {
    case scala.util.Success(result) => println(s"Combined
result: $result")
    case scala.util.Failure(e) => println(s"Failed with
${e.getMessage}")
  }

  Thread.sleep(3000) // Keep JVM alive to see result
}
```

Note

- `async { ... }` marks an asynchronous block.
- `await(future)` suspends the block until the future completes, but without blocking the thread.
- This style makes asynchronous code look sequential and easier to read.

Parallel Asynchronous Execution

You can run Futures concurrently by starting them before `await` calls:

```
val parallelCombined: Future[Int] = async {
  val f1 = slowComputation      // Trigger future, does not wait
  val f2 = anotherSlowComputation("Scala")
  await(f1) + await(f2)         // Await both results after
they've started
}
```

This runs both computations in parallel, improving overall performance compared to sequential `await` usage.

Summary:

- Use `Future` for basic asynchronous tasks with callbacks or combinators (`map`, `flatMap`).
- Use `scala-async` to write asynchronous logic with `async`/`await`, improving code clarity.
- Control concurrency by carefully ordering your `Future` creations and `await` calls.

If you want, I can provide a complete runnable Scala program that demonstrates these concepts end-to-end. Let me know!