EPFL

Execution Context

Effective Programming in Scala

Where Are Asynchronous Computations Executed?

So far, we have seen how to control *when* to execute asynchronous computations (e.g., we can chain them with `flatMap`).

However, when we call the method `bcrypt`, in which thread is it executed? Also, when we use `map` to transform the result of fetching a resource from a remote service, *where* is this code executed?

Execution Threads

Scala provides a thread-pool to execute asynchronous computations.

By default, it contains as many threads as the number of physical processors.

This means that operations that combine `Future` values in parallel (`zip` and `traverse`) have a parallelism level equal to the number of processors.

Third parties can provide custom execution contexts (e.g., with just one thread, or with as many threads as possible).

How Are Asynchronous Computations Scheduled?

In the standard library, all the operations that return `Future` values also take an implicit parameter of type `ExecutionContext`.

```
trait Future[A]:
  def map[B](f: A => B)(using ExecutionContext): Future[B]
  def zip[B](that: Future[B])(using ExecutionContext): Future[(A, B)]
  ...
```

The implementation of `map` schedules the execution of calling the function `f` in the given execution context.

Using the Scala `global` Execution Context

The simplest way to provide such a given `ExecutionContext` is to add the following import:

```
import scala.concurrent.ExecutionContext.Implicits.given
```

Blocking

The default `ExecutionContext` is designed to make all the computing power of a machine (i.e., its processors) available to your program.

It tries to fairly distribute the tasks that need to be computed to the threads.

However, sometimes a task occupies a thread while doing nothing! Example:

```
Future {
  Thread.sleep(10_000)
}
```

This program creates a task that will use a thread to do nothing but waiting 10 seconds. We say that such programs *block* threads.

Blocking (2)

In practice, blocking happens when you read a file, or communicate with a remote database with JDBC.

In such a case, you should create more threads than the number of physical processors, to not starve your thread-pool.

You achieve that by wrapping your “blocking” code in `concurrent.blocking`:

```
Future {
  concurrent.blocking {
    Thread.sleep(10_000)
  }
}
```

Creating `Future` Values

In the previous code examples, we have seen several ways to construct `Future` values.

1. Immediately create a successful `Future` value by using the constructor `Future.successful` with a value that you already have on hand.
2. Immediately create a failed `Future` value by using the constructor `Future.failed` with an exception.
3. Use a `Future { }` block to schedule a computation in the available `ExecutionContext`.

Combining `Future` Values vs Combining Asynchronous Computations

One pitfall with using `Future` is that you manipulate the *results* of computations, but not the computations themselves.

When you use operations on `Future` (such as `flatMap`), you don't manipulate the computations that produce the `Future` values, but only their results.

As a consequence, you may, by mistake, run two independent computations sequentially (although running them in parallel would be more efficient).

Let us illustrate this with an example.

Comparing Two User's Age

Consider a method `getUser`, which returns a `Future[User]`:

```
def getUser(id: UserID): Future[User]
```

Here is a program that computes the age difference between two users:

```
getUser(id1).flatMap { user1 =>
  getUser(id2).map { user2 =>
    user1.age - user2.age
  }
}
```

Can you tell whether the calls to `getUser` can happen in parallel or if they will necessarily happen sequentially?

Comparing Two User's Age (Possibly In Parallel)

Here are two ways to perform the calls to `getUser` in parallel. First, with `zip`:

```
getUser(id1).zip(getUser(id2)).map { (user1, user2) =>
  user1.age - user2.age
}
```

Or with `flatMap`, by moving the second call outside of the function passed to `flatMap`:

```
val eventualUser2 = getUser(id2)
getUser(id1).flatMap { user1 =>
  eventualUser2.map { user2 =>
    user1.age - user2.age
  }
}
```

Summary

Asynchronous computations are executed in an `ExecutionContext`, which handles the underlying threads.

The default execution context is optimized for non-blocking code. In case you use blocking code, remember to wrap it in `concurrent.blocking`.

Most of the time, you don't need to think much about the execution context. Just import it at the beginning of your source files, or take it as a context parameter.

Rule of thumb: prefer combining `Future` values of independent computations with `zip` instead of `flatMap`, to maximize parallelism.