Lecture 12 – Concurrent Programming

SWS121: Secure Programming

Jihyeok Park

PLRG

2024 Spring

Recall

- **Metaprogramming**
- **Inline**
  - Inline Constants
  - Inline Methods
  - Inline Parameters
  - Inline Matches
  - Transparent Inline Methods
- **Macros**

Contents

Contents

Futures

The following code immediately **runs** `task` and bounds to `x`:

```
// Sleep for 5 seconds and then return 42
def task: Int = { Thread.sleep(5_000); 42 }
val x = task // Blocks for 5 seconds and then x = 42
```

Can we **run** `task` in a **non-blocking** way? Yes with **Futures**!

Futures provide a way to reason about performing many operations in **parallel** – in a **non-blocking** way.

A `Future` represents a value which may or may not be currently available, but will be available at some point, or an exception if not.

To utilize a `Future`, we need to import the following:

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

Future

If we wrapping it into the `Future`, it has **not been completed** yet:

```
val eventualInt: Future[Int] = Future(task) // Future(<not completed>)
```

But if we check again **after 5 seconds**, it is **completed successfully**:

```
eventualInt // Future(<not completed>) before 5 seconds
eventualInt // Future(Success(42)) after 5 seconds
```

The value in a `Future` is always an instance of `Try` types:

- `Success` if the computation is successful
- `Failure` if the computation throws an exception

Therefore, we need to handle the `Try` type to get the result.

Callbacks – `onComplete`

We can use **callbacks** with futures to handle the result.

There are three common callbacks:

- `onComplete`
- `foreach`
- `andThen`

```
// Sleep for 3 seconds and then return a list of names
def namesTask: List[String] =
  Thread.sleep(3_000)
  List("Park", "Lee", "Ryu", "Hong")
```

The `onComplete` callback takes a **function** that handles the `Try` type:

```
// After 3 seconds, prints "Park", "Lee", "Ryu", "Hong"
Future(namesTask).onComplete {
  case Success(names) => for (name <- names) println(name)
  case Failure(e) => e.printStackTrace
}
```

Callbacks – `foreach`

If we want to **only** handle the **successful case**, we can use `foreach`:

```
Future(namesTask).foreach(names => for (name <- names) println(name))
```

It is equivalent to the following code using `for`-comprehension:

```
for {
  names <- Future(namesTask)
  name <- names
} println(name)
```

because the `for`-comprehension without `yield` will be **desugared** into the sequence of `foreach` method calls:

```
Future(namesTask).foreach(names => names.foreach(name => println(name)))
```

Callbacks – andThen

The andThen callback is used purely for **side-effecting** purposes.

While onComplete and foreach return a Unit, andThen returns the original Future without any transformation.

```
var firstChars: Set[Char] = Set.empty
Future {
  namesTask
}.andThen {
  case Success(names) =>
    println("Assigning first characters...")
    Thread.sleep(2_000)
    for (name <- names) firstChars += name.head
}.andThen {
  case _ =>
    println("Printing first characters...")
    Thread.sleep(2_000)
    for (c <- firstChars) println(c)  // 'P', 'L', 'R', 'H'
}
```

Combinators – map

We can use **combinators** to transform the value inside the Future.

There are three common combinators:

- map – maps the value inside the Future
- flatMap – maps and flattens the value inside the Future
- filter – filters the value inside the Future

```
// Sleep for 3 seconds and then return a list of names
def namesTask: List[String] =
  Thread.sleep(3_000)
  List("Park", "Lee", "Ryu", "Hong")
```

The map combinator takes a **function** transforming the value in Future:

```
val lengths = Future(namesTask).map(names => names.map(_.length))

lengths // Future(Success(List(4, 3, 3, 4))) after 3 seconds
```

Combinators – `flatMap` and `filter`

The `flatMap` combinator is used when the transformation returns a `Future`; it **flattens** the nested `Future`:

```
val nestedLengths: Future[Future[List[Int]]] =
  Future(namesTask).map(names => Future(names.map(_.length)))

val lengths: Future[List[Int]] =
  Future(namesTask).flatMap(names => Future(names.map(_.length)))
```

The `filter` combinator creates a new `Future` with the value satisfying the **predicate**:

```
val namesTrue = Future(namesTask).filter(_.length > 3)
val namesFalse = Future(namesTask).filter(_.length > 7)
```

After 3 seconds, two `Future` objects will be:

```
namesTrue   // Future(Success(List("Park", "Lee", "Ryu", "Hong")))
namesFalse  // Future(Failure(... predicate is not satisfied)
```

Combinators – For-Comprehension

PLRG

We can also use **for-comprehension** to use `map`, `flatMap`, and `filter` (more precisely, `withFilter`) combinators for `Future` objects.

```
val lengths: Future[List[Int]] = for {
  names <- Future(namesTask)
  if names.length > 3
  lengths <- Future(names.map(_.length))
} yield lengths
```

It will be **desugared** into the following code:

```
val lengths: Future[List[Int]] = Future(namesTask)
  .withFilter(names => names.length > 3)
  .flatMap(names => {
    Future(names.map(_.length))
  })
```

Multiple Futures

To run multiple computations in **parallel** and **combine** the results, we need to use **for-comprehension**.

For example, we can combine three futures `f1`, `f2`, and `f3`:

```
val f1 = Future { Thread.sleep(1_000); 5 }
val f2 = Future { Thread.sleep(2_000); 6 }
val f3 = Future { Thread.sleep(3_000); 7 }

val result = for {
  r1 <- f1
  r2 <- f2
  r3 <- f3
} yield r1 + r2 + r3

// Prints "The result is 18." after 3 seconds
result.foreach { r => println(s"The result is $r.") }

println("The main thread waits for the result.")
Thread.sleep(10_000)
```

Multiple Futures

Note that if the computations were run within the for-comprehension, they would be executed **sequentially**.

```
val result = for {
  r1 <- Future { Thread.sleep(1_000); 5 }
  r2 <- Future { Thread.sleep(2_000); 6 }
  r3 <- Future { Thread.sleep(3_000); 7 }
} yield r1 + r2 + r3

// Prints "The result is 18." after 6 seconds
result.foreach { r => println(s"The result is $r.") }
```

So, we need to remember to run the computations **outside** the for-comprehension to run them in **parallel**.

Futures

To summarize, a few **key points** about futures are:

- Futures are intended for **one-shot** computations by creating a temporary pocket of concurrency.
- A future starts running as soon as it is **created**.
- We don't have to concern ourselves with the **low-level details** of thread management.
- We can combine **multiple** futures using **for-comprehension**.

Contents

Promise

So far, we have only considered `Future` objects directly created by the constructor of the `Future` class.

We can also create a `Future` object using a **promise**.

Futures are defined as a type of read-only placeholder object created for a result which does not yet exist.

Promises are defined as a writable, single-assignment container, which completes a future with a value.

We need to import the following to use `Promise`:

```scala
import scala.concurrent.Promise
```

We can complete a future `p.future` of a promise `p` with:

- `success` – completes with a value to represent success
- `failure` – completes with an exception to represent failure

Promise

```scala
val p: Promise[Int] = Promise()
val f: Future[Int] = p.future

val producer = Future {
  println("Producing...")
  val x: Int = { Thread.sleep(2_000); 42 }
  println("Done producing.")
  p.success(x)
  println("Producer do something else...")
}

val consumer = Future {
  println("Consumer set up a callback...")
  f.foreach { r =>
    println(s"Consuming... $r")
    Thread.sleep(3_000)
    println("Done consuming.")
  }
  println("Consumer do something else...")
}
```

Promise

```scala
val producer = Future {
  println("Producing...")
  val x: Int = { Thread.sleep(2_000); 42 }
  println("Done producing.")
  p.success(x)
  println("Producer do something else...")
}
```

The `producer` future produces a value `x = 42` after 2 seconds.

Then, it **completes** the future `f` of the promise `p` with the value of `x` (i.e., 42) using `p.success(x)`.

Finally, without waiting for the completion of the future `f`, the `producer` future continues to do something else.

Promise

```scala
val consumer = Future {
  println("Consumer set up a callback...")
  f.foreach { r =>
    println(s"Consuming... $r")
    Thread.sleep(3_000)
    println("Done consuming.")
  }
  println("Consumer do something else...")
}
```

The `consumer` future sets up a callback to consume the value of the future `f` of the promise `p`.

Without waiting for the completion of the future `f`, the `consumer` future continues to do something else.

After 2 seconds, the future `f` of the promise `p` is completed with the value 42, and the callback (`foreach`) is executed.

After 3 seconds, the callback is done consuming the value 42.

Promise

The method `completeWith` can be used to complete a promise `p` with another future `f` (i.e., `p.completeWith(f)`).

```
val producer = Future {
  println("Producing...")
  val intFuture: Future[Int] = Future {
    Thread.sleep(2_000)
    println("Done producing.")
    42
  }
  p.completeWith(intFuture)
  println("Producer do something else...")
}
```

The above code is almost equivalent to the previous code.

However, the only difference is that the `producer` future does something else **before** producing the value 42.

Contents

Parallel Collection

We can use **parallel collections**[1] to perform operations in **parallel**.

However, since it is an external library, we need to install it in `build.sbt`:

```
libraryDependencies +=
  "org.scala-lang.modules" %% "scala-parallel-collections" % "<version>"
```

And, we need to import the following to use parallel collections:

```
import scala.collection.parallel.CollectionConverters.*
```

Then, we can freely **convert** a collection to the corresponding parallel collection using the `par` method:

```
List(1, 2, 3, 4, 5).par // A parallel collection of List(1, 2, 3, 4, 5)
```

[1] https://github.com/scala/scala-parallel-collections

Parallel Collection

For example, consider the following code:

```
def slowInc(x: Int): Int = { Thread.sleep(1_000); x + 1 }
val list = List(1, 2, 3, 4, 5)
list.map(slowInc) // List(2, 3, 4, 5, 6) after 5 seconds
```

It will take **5 seconds** to complete.

However, we can convert the list to a parallel collection and perform the `slowInc` operation in parallel:

```
list.par.map(slowInc).toList // List(2, 3, 4, 5, 6) after 1 second
```

It will take **1 second** to complete.

Parallel Collection

Similarly, we can use other methods such as `reduce` or `filter`:

For example, we can compute the sum of the first 1,000,000 numbers in parallel using `reduce` method:

```
(1L to 1_000_000L).toArray.par.reduce(_ + _) // 500000500000
```

Or, we can filter numbers divisible by 3 in parallel using `filter` method:

```
(1L to 1_000_000L).toArray.par.filter(_ % 3L == 0L).length // 333333
```

Parallel Collection

However, we need to be careful the **out-of-order** behavior when using parallel collections.

The **out-of-order** semantics of parallel collections can lead to the following implications:

- **Side-effecting** operations can lead to **non-determinism**.
- **Non-associative** operations can lead to **non-determinism**.

For example, the following code is **non-deterministic** because of the **side-effect** operation `sum += i`:

```
var sum = 0
(1 to 1_000).toArray.par.foreach { i => sum += i }
sum
```

The following code is also **non-deterministic** because of the **non-associative** operation `-`:

```
(1 to 1_000).toArray.par.reduce(_ - _)
```

Summary

1. Futures
    - Callbacks
    - Combinators
    - Multiple Futures
2. Promise
3. Parallel Collection

Next Lecture

- Course Review

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr