\>>> Channeling the Inner Complexity

\>>> or, lightweight threads and channels for Scala

Name: Jakob Odersky

Date: 2018-11-15

>>> Overview

* Basic concurrency models
* Futures and Promises
* Channels and lightweight threads

>>> Definitions

* **parallelism**: the simultaneous execution on multiple processors of different parts of a program[1]
* **concurrency**: the ability of different parts of a program to be executed out-of-order or in partial order, without affecting the final outcome[2]

[1] https://en.wikipedia.org/wiki/Parallelism
[2] https://en.wikipedia.org/wiki/Concurrency_(computer_science)

>>> Premise

* scalable programs need a good concurrency model
* "good":
    * increased efficiency (take advantage of parallelism)
    * reduced complexity

>>> Concurrency - Threads

* single entry point, sequence of instructions
* traditional way to decompose programs for parallel execution
* own stack and kernel resources (fairly expensive)
* context switches (fairly expensive)
* runnable on a physical processor

>>> Single Thread

```scala
def mkmeme(imageUrl: String, text: String): Image = {
  val layer1: Image = fetchUrl(imageUrl) // network call
  val layer2: Image = textToImage(text) // slow
  superimpose(layer1, layer2) // need both results
}
```

>>> Single Thread

* concurrency unit is the whole program

>>> Many Threads

```
def mkmeme(imageUrl: String, text: String): Image = {
  var layer1: Image = null
  var layer2: Image = null
  thread {
    layer1 = fetchUrl(imageUrl)
  }
  thread {
    layer2 = textToImage(text)
  }
  while(layer1 == null || layer2 == null) {
    // wait somehow
  }
  superimpose(layer1, layer2)
}
```

>>> Many Threads

* synchronization between threads at some point
* rendezvous through memory barriers (CMPXCHG)
* logic flow much more complex
* threads, blocked and running
    * consume memory
    * memory is cheap! create more threads? context switches
* threads are a low-level building block, using them efficiently is complex
* not available on all platforms (i.e. browser)

>>> Multiple Threads, Queue-based

```scala
def mkmeme(imageUrl: String, text: String): Image = {
  val q1 = Queue[Image]
  val q2 = Queue[Image]
  thread {
    q1.put(fetchUrl(imageUrl))
  }
  thread {
    q2.put(textToImage(text))
  }
  superimpose(q1.take(), q2.take())
}
```

>>> Multiple Threads, Queue-based

* simpler logic flow
* same resource usage as plain threads

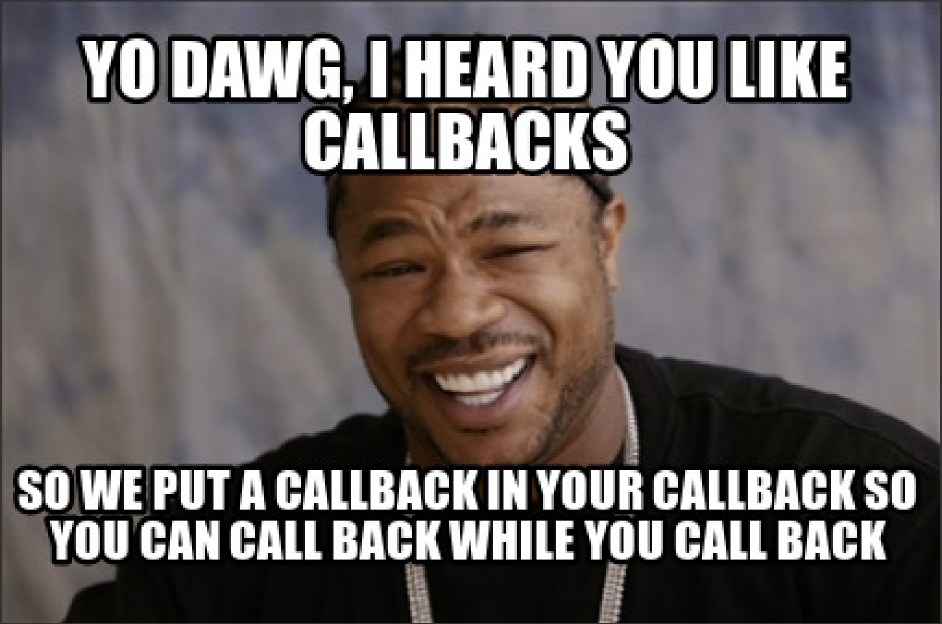
>>> Concurrency - Callbacks

* "reactive"
* many entrypoints
* register operation on event
* "call back" when event has happened, operation is run
* examples:
    * JavaScript
    * libuv
    * event loops
* in a sense, a more fundamental construct

_

>>> Callbacks

```
def mkmeme(imageUrl: String, text: String,
    callback: Image => Unit): Unit = {
  var layer1 = null
  var layer2 = null
  def combine() = callback(superimpose(layer1, layer2))
  fetchUrl(imageUrl, img => {
    layer1 = img
    if (layer2 != null) { //!\\ danger if parallelism > 1
      combine()
    }
  })
  textToImage(text, img => {
    layer2 = img
    if (layer1 != null) {
      combine()
    }
  })
}
```


YO DAWG, I HEARD YOU LIKE CALLBACKS
SO WE PUT A CALLBACK IN YOUR CALLBACK SO YOU CAN CALL BACK WHILE YOU CALL BACK

[~]$ _

>>> Callbacks

* advantages:
    * little resource overhead
    * available on all platforms
    * runnable on many processors
* disadvantage:
    * program logic quickly becomes extremely complex and scattered: *callback hell*

* can we wrap callbacks in a more functional way?
    * reduce complexity
    * keep efficiency, and run it on ideal number of processors

>>> Concurrency - Futures

scala.concurrent.Future[A]

  * contains an operation of result type A
  * transformable with map and flatMap
  * when operation is run, future completes with a result
    (success or failure)

>>> Future

```scala
def mkmeme(imageUrl: String, text: String): Future[Image] = {
  val layer1: Future[Image] = fetchUrl(imageUrl)
  val layer2: Future[Image] = textToImage(text)
  for {
    l1 <- layer1
    l2 <- layer2
  } yield {
    superimpose(l1, l2)
  }
}
```

>>> Promises

scala.concurren.Promise[A]

* used to create and complete futures at the edge of the callback graph

```scala
// ScalaJS, env: browser

def url: Future[String] = {
  val promise = Promise[String] // create promise
  input.onsubmit(_ => promise.success(input.value))
  promise.future
}

// single callback at the edge
url.map(fetch).onComplete{
  case Success(site) => webview.value = site
  case Failure(error) =>
    textbox.value = "oh no!"
    textbox.color = red
}
```

>>> Execution Contexts

Who runs a future?

* one process traverses all callbacks? no!
* operation "chunks" on an execution context

ExecutionContext

* contains graph of callbacks as chunks

```
future1.flatMap(f1 => op1(f1).map(op2(_))(ec))(ec)
```

* chunks are run on a *ThreadPool*

ThreadPool

* (limited) group of threads
* every thread runs a chunk, when done takes a next chunk
  * aside: *when done* ← this is why blocking in futures is not recomended

>>> Futures - Composition

```scala
def lookupUser(id: String): Future[Option[User]]
def authorize(user: User, capabilities: Set[Cap]):
  Future[Option[User]]

def authorizeduser(userId: String): Future[Option[User]] = {
  lookupUser(userId).flatMap{
    case None => Future.successful(None)
    case Some(user) => authorize(user, Set("see_meme"))
  }
}
```

>>> Futures - Shortcomings

1. composition can be messy[3]
2. one-shot; it is not simple to model recurrent events

[3] monad transformers may help

>>> Solution to 1 - Scala Async

* Can we write a program that looks synchronous (single-threaded), but is split into chunks and run on a thread pool?

* yes, with macros!

* two constructs:

    * async(a: => A): Future[A] // macro
    * await(f: Future[A]): A // usable only in await

* installs handlers on futures to run a state machine

* official project of the Scala Center

* https://github.com/scala/scala-async

* see also python async

```scala
import scala.concurrent.ExecutionContext.Implicits.global
import scala.async.Async._

// looks like single-threaded code
def mkmeme(imageUrl: String, text: String): Future[Image] =
  async {
    val layer1 = await(fetchUrl(imageUrl))
    val layer2 = await(textToImage(text))
    superimpose(layer1, layer2)
  }
```

>>> Solution to 2 - Channels

* futures are one-shot value
* queues are general useful construct for scalable programs
    * separation of concerns
* as shown previously, traditional thread-based queues block
* can we avoid blocking, yet keep the programming model?

>>> Solution to 2 - Channels

* project "escale" (fr. stop, as in bus stop)

* inspired from Clojure's core.async library

* watch Rich Hickey's talk about it
  https://www.infoq.com/presentations/core-async-clojure

* constructs:

    * go {block}: Future[A] ~ lightweight thread
    * Channel[A] ~ queue
    * ch.put(value: A): Future[A] ~ write operation
    * ch.take(): Future[A] ~ read operation
    * select(ch: Channel[_]*)

* syntax sugar

* form of communicating sequential processes (CSP) [1]

    * there is a formal mathematical model

* since runtime is abstracted, runs on JVM, JS and Native

```
>>> escale
import scala.concurrent.ExecutionContext.Implicits.global
import escale.syntax._

val ch = chan[Int]() // create a channel

go {
  ch !< 1 // write to channel, "block" if no room
  println("wrote 1")
}
go {
  ch !< 2
  println("wrote 2")
}

go {
  val r: Int = !<(ch) // read from channel
  println(r)
  println(!<(ch))
}
```

[~]$ _

>>> escale

```scala
import escale.syntax._

go {
  val Ch1 = chan[Int]() // create a channel
  val Ch2 = chan[Int]()

  go { Ch1 !< 1 } // write to channel
  go { Ch2 !< 1 }

  // "await" one and only one value
  select(Ch1, Ch2) match {
    case (Ch1, value) => "ch1 was first"
    case (Ch2, value) => "ch2 was first"
  }
}
```

\>>> escale - Implementation

* proof-of-concept
* https://github.com/jodersky/escale (soon)
* channels take care of buffering and efficient locking operations
* put and take return futures (select slightly more complex, but also returns a future)
* rely on scala-async to transform future into state machine
* provide syntax sugar to hide calls to await and alias async

>>> escale - Roadmap

* channel closing and error handling
* deeper integration with scala async
    * explore working with the state machine directly, rather than relying on double macro transformations
* select on puts
* buffer policies (drop first, sliding window)
* API improvements:
    * consider replacing symbols
    * remove wilcard import escale.sytntax._
    * directionality type refinements

>>> Summary: what have we done?

* replaced queues and threads with conceptually lightweight queues and threads
* same programming model, better concurrency
* in a library!

All problems in computer science can be solved by another level of indirection.

>>> Other Approaches

Actors

* actors and CSP can be considered duals
* actors are named, processes are anonymous
* message path is anonymous, channels are named
* sending messages is fundamentally non-blocking, whereas (unbuffered) channels can serve as rendezvous points

Reactive Streams

* builds a protocol on top of actors to achieve rendezvous capabilities and backpressure

>>> Guidelines

Keep programs *simple*, it will make it *easier* for others to understand.

1. write synchronous logic
2. use futures and promises with scala-async
3. escale and other concurrency libraries
4. ...
5. ...
6. ...
7. ...
8. ...
9. ...
10. consider callbacks

>>> Thank You!

* slides: https://jakob.odersky.com/talks
* project: https://github.com/jodersky/escale
* author: @jodersky

>>> References

[1] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM. 21 (8)*, pp. 666-667, 1978.