Table of Contents for Programming Scala, 2nd Edition

The *multicore problem* was a growing concern in the early 2000s as we hit the end of Moore's Law for single-core CPUs. We've continued to scale performance through increasing numbers of cores and servers, trading vertical scaling for horizontal scaling.

That's created a challenge for developers, because it has meant writing concurrent software. Concurrency isn't easy because it has traditionally required coordinated access to shared, mutable state, which means difficult multithreaded programming with tools like locks, mutexes, and semaphores. Failure to coordinate access correctly can result in the *spooky action at a distance* we mentioned in Chapter 2, where some mutable state you're using suddenly and unexpectedly changes, due to activity on another thread. Or it can mean race conditions and lock contention.

Functional programming started going mainstream when we learned that embracing immutability and purity largely bypasses these problems. We also saw a renaissance of older approaches to concurrency, like the actor model.

This chapter explores concurrency tools for Scala. You can certainly use any mechanism you've used with Java, including the multithreading APIs, message queues, etc. We'll just discuss Scala-specific tools, starting with an API for a very old approach: single-threaded processes that work together.

The scala.sys.process Package

In some cases, we can use small, synchronous processes that coordinate state through database transactions, message queues, or piping data from one process to another.

The `scala.sys.process` package provides a DSL for running and managing operating system processes, including I/O. Here's a REPL session that demonstrates some of the features. Note that a Unix shell like `bash` is used for the commands:

```
// src/main/scala/progscala2/concurrency/process/processes.sc
scala> import scala.sys.process._
scala> import scala.language.postfixOps
scala> import java.net.URL
scala> import java.io.File

// Run the command, write to
stdout.
       "ls -l
scala> src"       .!
total 0
drwxr-xr-x  4 deanwampler  staff  136 Dec 19  2013 main
drwxr-xr-x  4 deanwampler  staff  136 Dec 19  2013 test
res33: Int = 0

// Pass command tokens as a Seq, return a single string of the
output.
scala> Seq("ls", "-l", "src").!!
res34: String =
"total
0
drwxr-xr-x  4 deanwampler  staff  136 Dec 19  2013
main
drwxr-xr-x  4 deanwampler  staff  136 Dec 19  2013
test
"
```

We can also connect processes. Consider these methods:

```
// Build a process to open a URL, redirect the output to "grep
$filter",
// and append the output to file (not overwrite
it).
def findURL(url: String, filter: String) =
                   "grep
  new URL(url) #> s$filter"       #>> new File(s"$filter.txt")

// Run ls -l on the file. If it exists, then count the
lines.
                                     "ls -l                "wc -l
def countLines(fileName: String) = s$fileName"        #&& s$fileName"
```

The `#>` method in the DSL overwrites a file or pipes into `stdin` for a second process. The `#>>` method can only be used to overwrite a file. The `#&&` method only runs the process to its right if the process to its left succeeds, which means that it returns exit code zero. Both methods return a `scala.sys.process.ProcessBuilder`. They don't actually run the commands. For that we need to invoke their `!` method:

```
scala> findURL("http://scala-lang.org", "scala") !
res0: Int = 0

scala> countLines("scala.txt") !
-rw-r--r--  1 deanwampler  staff  4111 Jul 31 22:35
scala.txt
      43 scala.txt
res1: Int = 0

scala> findURL("http://scala-lang.org", "scala") !
res2: Int = 0

scala> countLines("scala.txt") !
-rw-r--r--  1 deanwampler  staff  8222 Jul 31 22:35
scala.txt
      86 scala.txt
res3: Int = 0
```

Note that the file size doubled, because we appended new text for each run.

When it's an appropriate design solution, small, synchronous processes can be implemented in Scala or any other language, then *glued* together using the `process` package API.

Futures

For many needs, process boundaries are too course-grained. We need easy to use concurrency primitives within a single process. That is, we need a higher-level API than the traditional multithreading APIs, one that exposes reasonably intuitive building blocks.

Suppose you have units of work that you want to run asynchronously, so you don't block while they're running. They might need to do I/O, for example. The simplest mechanism is `scala.concurrent.Future`.

When you construct a `Future`, control returns immediately to the caller, but the value is not guaranteed to be available yet. The `Future` instance is a handle to an eventually available result. You can continue doing other work until the future completes, either successfully or unsuccessfully. There are different ways to handle this completion.

We saw a simple example in A Taste of Futures, which we used to discuss implicit arguments, the `scala.concurrent.ExecutionContext` required to manage and run futures. We used `ExecutionContext.global`, which manages a thread pool using `java.util.concurrent.ForkJoinPool`, which it uses to perform the work encapsulated in the `Future`s. As users, we don't need to care about how our asynchronous processing is executed, except for special circumstances, such as performance tuning. (While `ForkJoinPool` is part of JDK 7, because Scala currently supports JDK 6, it actually ships the original implementation by Doug Lea that was subsequently added to JDK 7.)

To explore `Future`s, first consider the case where we need to do ten things in parallel, then combine the results:

```
// src/main/scala/progscala2/concurrency/futures/future-fold.sc
import scala.concurrent.{Await, Future}
import scala.concurrent.duration.Duration
import scala.concurrent.ExecutionContext.Implicits.global

val futures = (0 to 9) map {
// ❶
  i => Future {
    val s = i.toString
// ❷
    print(s)
    s
  }
}

val f = Future.reduce(futures)((s1, s2) => s1 + s2)                    //
❸

val n = Await.result(f, Duration.Inf)
// ❹
```

❶

Create ten asynchronous futures, each performing some work.

❷

`Future.apply` takes two argument lists. The first has a single, by-name `body` to execute asynchronously. The second list has the implicit `ExecutionContext`. We're allowing the `global` implicit value to be used. Our `body` converts the integer to a string, prints it, then returns it. The type of `futures` is `IndexedSeq[Future[String]]`. In this contrived example, the `Future`s complete immediately.

❸

Reduce the sequence of `Future` instances into a single `Future[String]`. In this case, concatenate the strings.

❹

To block until the `Future` `f` completes, we use `scala.concurrent.Await`. The `Duration` argument says to wait for *infinity*, if necessary. Using `Await` is the preferred way to block the current thread when you need to wait for a `Future` to complete.

Crucially, when the `print` statements in the `Future` body are called, the outputs will be out of order, e.g., `0214679538` and `0123467985` are the outputs of two of my runs. However, because `fold` walks through the `Future`s in the same order in which they were constructed, the string it produces always has the digits in strict numerical order, `0123456789`.

`Future.fold` and similar methods execute asynchronously themselves; they return a new `Future`. Our example only blocks when we called `Await.result`.

Often, we don't want to block on a result. Instead, we want a bit of code to be invoked when the `Future` completes. Registering a callback does the trick. For example, a simple web server might construct a `Future` to handle a request and use a callback to send the result back to the caller. The following example demonstrates the logic:

```
// src/main/scala/progscala2/concurrency/futures/future-callbacks.sc
import scala.concurrent.Future
import scala.concurrent.duration.Duration
import scala.concurrent.ExecutionContext.Implicits.global

case class ThatsOdd(i: Int) extends RuntimeException(                //
❶
   "odd $i
  sreceived!"        )

import scala.util.{Try, Success, Failure}
// ❷

val doComplete: PartialFunction[Try[String],Unit] = {                //
❸
  case s @ Success(_) => println(s)
// ❹
  case f @ Failure(_) => println(f)
}

val futures = (0 to 9) map {
// ❺
  case i if i % 2 == 0 => Future.successful(i.toString)
  case i => Future.failed(ThatsOdd(i))
}
futures map (_ onComplete doComplete)
// ❻
```

❶

An exception we'll throw for odd integers.

❷

Import `scala.util.Try` and its descendants, `Success` and `Failure`.

❸

Define a callback handler for both successful and failed results. Its type must be `PartialFunction[Try[String],Unit]`, because the callback will be passed a `Try[A]`, encapsulating success or failure. `A` will be `String`. The function's return type is `Unit`, because nothing can be returned from the handler, since it runs asynchronously. For a web server, it would send a response to the caller here.

❹

If the `Future` succeeds, the `Success` clause will match. Otherwise the `Failure` will match. We just print either result.

❺

Create the `Future`s where odd integers are "failures." We use two methods on the `Future` companion object for immediately completing the `Future` as a success or failure.

❻

Traverse over the `futures` to attach the callback, which will be called immediately since our `Future`s have already completed by this point.

Running the script produces output like the following, where the order varies from run to run:

```
Success(0)
Success(2)
Failure($line137.$read$$iw$$iw$ThatsOdd: odd 1 received!)       //
❶
Success(4)
Failure($line137.$read$$iw$$iw$ThatsOdd: odd 3 received!)
Success(6)
Success(8)
Failure($line137.$read$$iw$$iw$ThatsOdd: odd 5 received!)
Failure($line137.$read$$iw$$iw$ThatsOdd: odd 9 received!)
Failure($line137.$read$$iw$$iw$ThatsOdd: odd 7 received!)
```

❶

Ugly, synthesized name created when compiling `ThatsOdd` in the script.

We'll also see more examples of `Future` idioms in the next section.

`Future` is "monadic" like `Option`, `Try`, `Either`, and the collections. We can use them in `for` comprehensions and manipulate the results with our combinator friends, `map`, `flatMap`, `filter`, and so forth.

Async

When working with `Future`s, excessive use of callbacks can get complicated quickly. So can composition of `Future`s to implement interdependent sequencing of tasks. A new `scala.async.Async` module is designed to make it easier to build such computations. It is described in SIP-22, implemented for both Scala 2.10 and 2.11 on GitHub, and distributed as an "optional module" (see Table 21-11 for the list of such modules for 2.11).

There are two fundamental methods used with an asynchronous block:

```
def async[T](body: => T): Future[T]              //
❶
def await[T](future: Future[T]): T               //
❷
```

❶

Start an asynchronous computation and return a `Future` for it immediately.

❷

Wait until a `Future` completes.

The following example simulates a sequence of asynchronous calls, first to determine if a "record" exists for an `id` and if so, to get the record. Otherwise, it returns an error record:

```
//
src/main/scala/progscala2/concurrency/async/async.sc
import scala.concurrent.{Await, Future}
import scala.concurrent.duration.Duration
import scala.async.Async.{async, await}
import scala.concurrent.ExecutionContext.Implicits.global

object AsyncExample {
  def recordExists(id: Long): Boolean = {                              //
❶
    println(s"recordExists($id)...")
    Thread.sleep(1)
    id > 0
  }

  def getRecord(id: Long): (Long, String) = {                          // ❷
    println(s"getRecord($id)...")
    Thread.sleep(1)
          "record:
    (id, s$id"          )
  }

  def asyncGetRecord(id: Long): Future[(Long, String)] = async {       // ❸
    val exists = async { val b = recordExists(id); println(b); b }
    if (await(exists)) await(async { val r = getRecord(id); println(r); r
})
              "Record not
    else (id, found!"             )
  }
}

(-1 to 1) foreach { id =>                                              //
❹
  val fut = AsyncExample.asyncGetRecord(id)
  println(Await.result(fut, Duration.Inf))
}
```

❶

An expensive predicate to test for the existence of a record. It returns true if the `id` is greater than zero.

❷

Another expensive method, which retrieves the record for an `id`.

❸

A method that sequences asynchronous computations together. It first invokes `recordExists` asynchronously. It waits on the result and if true, it fetches the record asynchronously. Otherwise, it returns an error record.

❹

Try it with three indices.

It produces the following results (after a few seconds):

```
recordExists(-1)...
false
(-1,Record not
found!)
recordExists(0)...
false
(0,Record not found!)
recordExists(1)...
true
getRecord(1)...
(1,record: 1)
(1,record: 1)
```

Note that `getRecord` is only called once, for the "valid" index `1`.

`Async` code is cleaner than sequencing `Future`s; it's still not as transparent as truly synchronous code, but you get the benefits of asynchronous execution.

Using `Future`s, with or without `Async`, is a *tactic* for concurrency, but not a *strategy*. It doesn't provide large-scale facilities for managing asynchronous processes, including error handling, on an application-wide scale. For those needs, we have the actor model.

Robust, Scalable Concurrency with Actors

Actors were originally designed for use in Artificial Intelligence research. Carl Hewitt and collaborators described them in a 1973 paper (the 2014 update is available at *arxiv.org*), and Gul Agha described the actor model theory in his 1987 book *Actors* (MIT Press). Actors are a core concept baked into the Erlang language and its virtual machine. In other languages, like Scala, actors are implemented as a library on top of other concurrency abstractions.

Fundamentally, an *actor* is an object that receives messages and takes action on those messages, one at a time and without preemption. The order in which messages arrive is unimportant in some actor systems, but not all. An actor might process a message internally, or it might forward the message or send a new message to another actor. An actor might create new actors as part of handling a message. An actor might change how it handles messages, in effect implementing a state transition in a state machine.

Unlike traditional object systems that use method calls, actor message sending is usually asynchronous, so the global order of actions is nondeterministic. Like traditional objects, an actor may control some state that it evolves in response to messages. A well-designed actor system will prevent any other code from accessing and mutating this state directly, or it will at least strongly discourage this practice.

These features allow actors to run in parallel, even across a cluster. They provide a *principled* approach to managing global state, largely (but not completely) avoiding the problems of traditional multithreaded concurrency.

Akka: Actors for Scala

In 2009 when the first edition of this book was written, Scala came with an actor library, which we used for the examples. However, a new, independent actor project called Akka had just started.

Today, the original actor library has been dropped from Scala and Akka is now the official library for actor-based concurrency in Scala. It remains a separate project. Both Scala and Akka are developed and supported by Typesafe. Akka provides a comprehensive Java API, too.

In A Taste of Concurrency, we worked through a simple example using Akka. Now let's work through a more realistic example. You might find the Akka Scaladocs useful as we go.

The two most important, production-ready implementations of the actor model are the Erlang implementation and Akka, which drew its inspiration from Erlang's implementation. Both implement an important innovation, a robust model of error handling and recovery.

Not only are actors created to do the routine work of the system, *supervisors* are created to watch the life cycle of one or more actors. Should an actor fail, perhaps because an exception is thrown, the supervisor follows a strategy for recovery that can include restarting, shutting down, ignoring the error, or delegating to its superior for handling.

When restarting, an *all-for-one* strategy is used when the failed actor is working closely with other actors, all under the same supervisor, and it's best to restart all of them. A *one-for-one* strategy is used when the managed actors are independent workers, where the failure of one has no impact on the others. Only the failed actor requires restarting.

This architecture cleanly separates error-handling logic from normal processing. It enables an architecture-wide strategy for error handling. Most importantly, it promotes a principle of *let it crash*.

Its common to mix error-handling logic with normal processing code, resulting in a complicated mess, which often fails to implement a complete, comprehensive strategy. Inevitably, some production scenarios will trigger a failed recovery that leaves the system in an inconsistent state. When the inevitable crash happens, service is compromised and diagnosing the real source of the problem proves difficult.

The example we'll use simulates a client interface invoking a service, which delegates work to workers. The client interface (and location of the `main` method) is called `AkkaClient`. It passes user commands to a single `ServerActor`, which in turn delegates work to several `WorkerActor`s, so that it never blocks. Each worker simulates a *sharded* data store. It maintains a map of keys (`Long`s) and values (`String`s), and it supports CRUD (create, read, update, and delete) semantics.

`AkkaClient` constructs the `akka.actor.ActorSystem` that controls everything. You'll have one or at most a few of those in any application. Then it constructs an instance of `ServerActor` and sends it a message to start processing. Finally, `AkkaClient` provides a simple command-line interface to the user.

Before walking through `AkkaClient`, let's look at `Messages`, which defines all the messages exchanged between our actors:

```
//
src/main/scala/progscala2/concurrency/akka/Messages.scala
package progscala2.concurrency.akka
import scala.util.Try

object Messages {
// ❶
  sealed trait Request {
// ❷
    val key: Long
  }
  case class  Create(key: Long, value: String) extends Request       //
❸
  case class  Read(key: Long) extends Request
// ❹
  case class  Update(key: Long, value: String) extends Request       //
❺
  case class  Delete(key: Long) extends Request                       //
❻

  case class  Response(result: Try[String])
// ❼

  case class  Start(numberOfWorkers: Int = 1)                         //
❽
  case class  Crash(whichOne: Int)
// ❾
  case class  Dump(whichOne: Int)
// ❿
  case object DumpAll
}
```

❶

Use a `Messages` object to hold all the message types.

❷

Parent `trait` for all CRUD requests, all of which use a `Long` key.

❸

Create a new “record” with the specified key and value.

❹

Read the record for the given key.

❺

Update a record (or create it if it doesn’t exist) with the new value for the given key.

❻

Delete a record for the given key or do nothing if it doesn’t exist.

❼

Wrap responses in a common message. A `scala.util.Try` wraps the result, indicating either success or failure.

❽

Start processing. This message is sent to the `ServerActor` and it tells it how many workers to create.

❾

Send a message to a worker to simulate a "crash."

❿

Send a message to "dump" the state of a single worker or all of them.

Now let's walk through `AkkaClient`:

```
// src/main/scala/progscala2/concurrency/akka/AkkaClient.scala
package progscala2.concurrency.akka
import akka.actor.{ActorRef, ActorSystem, Props}
import java.lang.{NumberFormatException => NFE}

object AkkaClient {
// ❶
  import Messages._

  private var system: Option[ActorSystem] = None
// ❷

  def main(args: Array[String]) = {
// ❸
    processArgs(args)
    val sys = ActorSystem("AkkaClient")
// ❹
    system = Some(sys)
    val server = ServerActor.make(sys)
// ❺
    val numberOfWorkers =
// ❻
      sys.settings.config.getInt("server.number-workers")
    server ! Start(numberOfWorkers)
// ❼
    processInput(server)
// ❽
  }

  private def processArgs(args: Seq[String]): Unit = args match {
    case Nil =>
    case ("-h" | "--help") +: tail => exit(help, 0)
                                "Unknown input
    case head +: tail => exit(s$head!\n"               +help, 1)
  }
  ...
```

❶

The client is an object so we can define `main` here.

❷

The single `ActorSystem` is saved in an `Option`. We'll use it in the shutdown logic that we'll discuss in the following text. Note that the variable is private and mutable. Concurrent access to it won't be a concern, however, because the actor has total control over it.

❸

The `main` routine starts by processing the command-line arguments. The only one actually supported in `processArgs` is a help option.

❹

Create the `ActorSystem` and update the `system` option.

❺

Call a method on `ServerActor`'s companion to construct an instance of it.

❻

Determine from the configuration how many workers to use.

❼

Send the `Start` message to the `ServerActor` to begin processing.

❽

Process command-line input from the user.

Akka uses Typesafe's Config library for configuration values defined in files or programmatically. The configuration file we're using is the following:

```
// src/main/resources/application.conf
akka {
// ❶
  loggers  = [akka.event.slf4j.Slf4jLogger]                          //
❷
  loglevel = debug

  actor {
// ❸
    debug {
// ❹
      unhandled = on
      lifecycle = on
    }
  }
}

server {
// ❺
  number-workers = 5
}
```

❻

❶

Configure properties for the Akka system as a whole.

❷

Configure the logging module to use. The SBT build includes the `akka-slf4j` module that supports this interface. There is a corresponding `logback.xml` in the same directory that configures the logging (not shown). By default, all debug and higher messages are logged.

❸

Configure properties for every actor.

❹

Enable debug logging of occurrences when an actor receives a message it doesn't handle and any life cycle events.

❺

The `ServerActor` instance will be named `server`. This block configures settings for it. We have one custom setting, the number of workers to use.

Back to `AkkaClient`, processing user input is one long method:

```
  ...
  private def processInput(server: ActorRef): Unit = {                //
❶
    val blankRE = """^\s*#?\s*$""".r
    val badCrashRE = """^\s*[Cc][Rr][Aa][Ss][Hh]\s*$""".r
    val crashRE = """^\s*[Cc][Rr][Aa][Ss][Hh]\s+(\d+)\s*$""".r
    val dumpRE = """^\s*[Dd][Uu][Mm][Pp](\s+\d+)?\s*$""".r
    val charNumberRE = """^\s*(\w)\s+(\d+)\s*$""".r
    val charNumberStringRE = """^\s*(\w)\s+(\d+)\s+(.*)$""".r

                          ">>
    def prompt() = print("    )
//  ❷
    def missingActorNumber() =
              "Crash command requires an actor
      println(number."                                    )
    def invalidInput(s: String) =
               "Unrecognized command:
      println(s$s"                          )
    def invalidCommand(c: String): Unit =
               "Expected 'c', 'r', 'u', or 'd'. Got
      println(s$c"                                        )
    def invalidNumber(s: String): Unit =
               "Expected a number. Got
      println(s$s"                           )
    def expectedString(): Unit =
              "Expected a string after the command and
      println(number"                                           )
    def unexpectedString(c: String, n: Int): Unit =
               "Extra arguments after command and number '$c
      println(s$n'"                                                 )
    def finished(): Nothing = exit("Goodbye!", 0)
```

```
    val handleLine: PartialFunction[String,Unit] = {                    //
❸
                          /* do nothing
      case blankRE() =>    */
      case "h" | "help" => println(help)
      case dumpRE(n) =>
//  ❹
        server ! (if (n == null) DumpAll else Dump(n.trim.toInt))
      case badCrashRE() => missingActorNumber()
//  ❺
      case crashRE(n) => server ! Crash(n.toInt)
      case charNumberStringRE(c, n, s) => c match {                      //
❻
        case "c" | "C" => server ! Create(n.toInt, s)
        case "u" | "U" => server ! Update(n.toInt, s)
        case "r" | "R" => unexpectedString(c, n.toInt)
        case "d" | "D" => unexpectedString(c, n.toInt)
        case _ => invalidCommand(c)
      }
      case charNumberRE(c, n) => c match {
//  ❼
        case "r" | "R" => server ! Read(n.toInt)
        case "d" | "D" => server ! Delete(n.toInt)
        case "c" | "C" => expectedString
        case "u" | "U" => expectedString
        case _ => invalidCommand(c)
      }
      case "q" | "quit" | "exit" => finished()
//  ❽
      case string => invalidInput(string)
//  ❾
    }

    while (true) {
      prompt()
//  ❿
      Console.in.readLine() match {
        case null => finished()
        case line => handleLine(line)
      }
    }
  }
  ...
```

❶

Start with definitions of regular expressions to parse input.

❷

Define several nested methods for printing the prompt, for reporting errors, and to finish processing and shutdown.

❸

The main handler is a partial function, exploiting that convenient syntax. It starts with a matcher for blank lines

(or “comments,” lines where the first non-whitespace character is a `#`), which are ignored. Next it handles help requests (`h` or `help`).

❹

Ask one or all workers to dump their state, the “datastore” of key-value pairs.

❺

To demonstrate Akka supervision, handle a message to crash a worker. First, handle the case where the user forgot to specify an actor number. Second, handle the correct syntax, where a message is sent to the `ServerActor`.

❻

If the command contains a letter, number, and string, it must be a “create” or “update” command. If so, send it to the `ServerActor`. Otherwise, report an error.

❼

Similarly, if just a command letter and number are input, it must be a “read” or “delete” command.

❽

Three ways to quit the application (Ctrl-D also works).

❾

Treat all other input as an error.

❿

Print the initial prompt, then loop until there’s no input, handling each line.

Note that we don’t crash or exit on invalid user commands. Unfortunately, we aren’t using a library like the Gnu *readline*, so backspaces aren’t handled correctly.

To finish this file:

```
  ...
  private val help =                                                    //
❶
  """Usage: AkkaClient [-h | --
  help]
    |Then, enter one of the following commands, one per
line:
    |  h | help      Print this help
message.
    |  c n string    Create "record" for key n for value
string.
    |  r n           Read record for key n. It's an error if n isn't
found.
    |  u n string    Update (or create) record for key n for value
string.
    |  d n           Delete record for key n. It's an error if n isn't
found.
    |  crash n       "Crash" worker n (to test
recovery).
    |  dump [n]      Dump the state of all workers (default) or worker
n.
    |  ^d | quit
Quit.

|"""    .stripMargin

  private def exit(message: String, status: Int): Nothing = {         // ❷
    for (sys <- system) sys.shutdown()
    println(message)
    sys.exit(status)
  }
}
```

❶

A detailed help message.

❷

A helper function for exiting. It shuts the `ActorSystem` down, if it was initialized (if `system` is a `Some`), prints a message, and exits.

Next, let's look at `ServerActor`:

```
//
src/main/scala/progscala2/concurrency/akka/ServerActor.scala
package progscala2.concurrency.akka
import scala.util.{Try, Success, Failure}
import scala.util.control.NonFatal
import scala.concurrent.duration._
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import akka.actor.{Actor, ActorLogging, ActorRef,
  ActorSystem, Props, OneForOneStrategy, SupervisorStrategy}
import akka.pattern.ask
```

```
import akka.util.Timeout


class ServerActor extends Actor with ActorLogging {                    //
❶
  import Messages._

  implicit val timeout = Timeout(1.seconds)

  override val supervisorStrategy: SupervisorStrategy = {              //
❷
    val decider: SupervisorStrategy.Decider = {
      case WorkerActor.CrashException => SupervisorStrategy.Restart
      case NonFatal(ex) => SupervisorStrategy.Resume
    }
    OneForOneStrategy()(decider orElse super.supervisorStrategy.decider)
  }

  var workers = Vector.empty[ActorRef]
// ❸

  def receive = initial
// ❹

  val initial: Receive = {
// ❺
    case Start(numberOfWorkers) =>
      workers = ((1 to numberOfWorkers) map makeWorker).toVector
      context become processRequests
// ❻
  }

  val processRequests: Receive = {
// ❼
    case c @ Crash(n) => workers(n % workers.size) ! c
    case DumpAll =>
// ❽
      Future.fold(workers map (_ ? DumpAll))(Vector.empty[Any])(_ :+ _)
                                "State of the
        .onComplete(askHandler(workers"              ))
    case Dump(n) =>
      (workers(n % workers.size) ? DumpAll).map(Vector(_))
                                 "State of worker
        .onComplete(askHandler(s$n"                  ))
    case request: Request =>
      val key = request.key.toInt
      val index = key % workers.size
      workers(index) ! request
    case Response(Success(message)) => printResult(message)
    case Response(Failure(ex)) => printResult(s"ERROR! $ex")
  }

  def askHandler(prefix: String): PartialFunction[Try[Any],Unit] = {
    case Success(suc) => suc match {
      case vect: Vector[_] =>
        printResult(s"$prefix:\n")
        vect foreach {
```

```
        vec.foreach {
          case Response(Success(message)) =>
            printResult(s"$message")
          case Response(Failure(ex)) =>
                          "ERROR! Success received wrapping
            printResult(s$ex"                                    )
        }
                               "BUG! Expected a vector, got
      case _ => printResult(s$suc"                               )
    }
    case Failure(ex) => printResult(s"ERROR! $ex")
  }

  protected def printResult(message: String) = {
    println(s"<< $message")
  }

  protected def makeWorker(i: Int) =
    context.actorOf(Props[WorkerActor], s"worker-$i")
}

object ServerActor {
// ❾
  def make(system: ActorSystem): ActorRef =
    system.actorOf(Props[ServerActor], "server")
}
```

❶

Mix in the `ActorLogging` trait, which adds a `log` field that can be used to log information.

❷

Override the default supervisor strategy with a custom `akka.actor.SupervisorStrategy`. If our simulated crash occurs, restart the actor. If another `NonFatal` exception occurs, just continue (risky!!). Because these workers are considered independent, use the *one-for-one* strategy. If the `Decider` doesn't have an error, handling is delegated to the parent supervisor.

❸

Keep track of the worker actors through the `akka.actor.ActorRef` instances that reference them.

❹

Define `receive` to be the `initial` request handler.

❺

`Receive` is a convenient type member of `Actor` that aliases `PartialFunction[Any,Unit]`. This line declares the `Receive` that will be used to handle the initial `Start` message to the actor. Only `Start` is expected. If other messages are sent to the actor, they will remain in the actor's mailbox until they are explicitly handled. Think of this `Receive` as implementing the first state of a state machine for this actor.

❻

When `Start` is received, construct the workers and transition to the second state of the state machine, where message handling is done by `processRequests`.

❼

This `Receive` block handles the rest of the messages after `Start` is received. The first few case classes match on `Crash`, `DumpAll`, and `Dump` messages. The `Request(request:)` clause handles the CRUD commands. Finally, `Response` messages from workers are handled. Note that user-specified worker indices are converted modulo the number of actual workers to a valid index into the `workers` vector. This actor prints the responses it receives from the workers.

❽

`DumpAll` needs to be forwarded to all workers and we would like to gather together all the responses from them and format a result message. We do that with the *ask pattern*. To use this feature, we must import `akka.pattern.ask` (at the top of the file). We use `?` to send a message, which returns a `Future`, instead of using `!`, which returns a `Unit`. Both message types are asynchronous, but in the ask pattern, a reply from the receiver is captured into the completion of the `Future`. We use `Future.fold` to join the sequence of `Future`s into a single `Future` wrapping a `Vector`. Then we use `onComplete` to register a callback, `askHandler`, to process the completed `Future`. You'll note that the nesting of types becomes complicated.

❾

The companion provides a convenient `make` method to construct the actor, following the required idiom for actor construction (discussed in the following text).

The `Actor.receive` method is *not* called every time a method is received. Instead, it is called *once*, when the actor is constructed to return a `Receive` (an alias to `PartialFunction[Any,Unit]`) that will be called repeatedly for each message. A message handler clause in this function can change the handling of all messages to a new `Receive` using `Actor.become`, as is done in the `case` clause for the `Start` message in `initial`. The `Receive` handler can be changed on every message, if desired, supporting the implementation of complex state machines. In this case, you can cut down on the boilerplate by mixing in the `FSM` (finite state machine) trait, which provides a convenient DSL for defining state machines.

`ServerActor` writes all worker replies to the console. It can't send them back to `AkkaClient`, because the latter is not an actor! Hence, if `ServerActor` calls `Actor.sender`, the method that returns the original sender `ActorRef`, `ActorSystem.deadLetters` is actually returned.

The idiom used to construct the `ServerActor`, `system.actorOf(Props[ServerActor], "server")`, is one of several possible variants. It solves two design problems. First, because actor instances are wrapped in `ActorRef`s, we can't simply call `new ServerActor`. Akka needs to properly wrap the instance and do other initialization steps.

Second, the `Props` singleton object exists primarily to solve an issue with how JVM byte code is generated. Actors need to be serializable, so they can be distributed remotely in clustered deployments; for details, see the Akka docs. When actor instances are created inside other instances, the Scala compiler will close over the scope, as needed, for the instance. This could mean that an enclosing instance of some other class is captured in the serialized byte code. That instance might not be serializable, so the actor can't be transferred to another node, or perhaps worse, state in the enclosing instance might be encapsulated with the actor, potentially leading to inconsistent behavior on the remote node. The singleton `Props` effectively prevents this issue from happening.

Finally, here is `WorkerActor`:

```
//
src/main/scala/progscala2/concurrency/akka/WorkerActor.scala
package progscala2.concurrency.akka
import scala.util.{Try, Success, Failure}
import akka.actor.{Actor, ActorLogging}

class WorkerActor extends Actor with ActorLogging {
  import Messages._

  private val datastore = collection.mutable.Map.empty[Long,String]  //
❶

  def receive = {
    case Create(key, value) =>
// ❷
      datastore += key -> value
                                "$key -> $value
      sender ! Response(Success(sadded"                ))
    case Read(key) =>
// ❸
                             "${datastore(key)} found for key =
      sender ! Response(Try(s$key"                                    ))
    case Update(key, value) =>
// ❹
      datastore += key -> value
                                "$key -> $value
      sender ! Response(Success(supdated"                 ))
    case Delete(key) =>
// ❺
      datastore -= key
                                "$key
      sender ! Response(Success(sdeleted"      ))
    case Crash(_) => throw WorkerActor.CrashException                //
❻
    case DumpAll =>
// ❼
                                "${self.path}: datastore =
      sender ! Response(Success(s$datastore"                          ))
  }
}

object WorkerActor {
  case object CrashException extends RuntimeException("Crash!")      //
❽
}
```

❶

Keep a *mutable* map of key-value pairs. Because the `Receive` handler is thread-safe (enforced by Akka itself) and this mutable state is private to the actor, it's safe to use a mutable object. Because sharing mutable state is dangerous, we'll *never* return this map to a caller through a message.

❷

Add a new key-value pair to the map. Send a response to the `sender`.

❸

Attempt to read a value for the given key. Wrapping the call to `datastore(key)` in a `Try` automatically captures into a `Failure` the exception that will be thrown if the key is not present. Otherwise, a `Success` is returned, wrapping the found value.

❹

Update an existing key with a new value (or create a new key-value pair).

❺

Delete a key-value pair. Effectively does nothing if the key isn't present.

❻

"Crash" the actor by throwing a `CrashException`. Recall that the `WorkerActor` supervision strategy is configured to restart the actor when this exception is thrown.

❼

Reply with the actor's state, namely a string built from the contents of the `datastore` map.

❽

The special `CrashException` used to simulate actor crashes.

Let's run it at the `sbt` prompt:

```
run-main progscala2.concurrency.akka.AkkaClient
```

(Or you can use `run` and select the number from the list shown.) Enter `h` to see the list of commands and try several. Use `quit` to exit. There is also a file of commands that can run through the program using the following command from your shell or command window:

```
sbt "run-main progscala2.concurrency.akka.AkkaClient" < misc/run-akka-
input.txt
```

Because the operation is inherently asynchronous, you'll see different results each time you run this script, and also if you copy and paste groups of input lines from the *misc/run-akka-input.txt* file.

Note that the data is lost when an actor is crashed. When this is unacceptable, the Akka Persistence module supports durable persistence of actor state so a restarted actor can recover the previous instance's state.

You might be concerned that the `ServerActor`'s list of workers would become invalid when an actor crashes. This is why all access to an actor goes through a "handle," an `ActorRef`, and direct access to the `Actor` instance itself is prevented. (The exception is the special API for actor testing. See the `akka.testkit` package.)

`ActorRef`s are very stable, so they make great *dependencies*. When a supervisor restarts an actor, it resets the `ActorRef` to point to the new instance. If the actor is not restarted nor resumed, all messages sent to the corresponding `ActorRef` are forwarded to the `ActorSystem.deadLetters`, which is the place where messages from dead actors go to die. Therefore, relationships between `ActorRef`s are stable and reliable.

Actors: Final Thoughts

Our application demonstrates a common pattern for handling a high volume of concurrent input traffic, delegating results to asynchronous workers, then returning the results (or just printing them in this case).

We only scratched the surface of what Akka offers. Still, you now have a sense for how a typical, nontrivial Akka application looks and works. Akka has excellent documentation at *http://akka.io*. Appendix A contains several books on Akka for more in-depth information, including the many patterns and idioms that have emerged for using Akka effectively.

Akka actors are lightweight, about 300 bytes per actor. Hence, you can easily create millions of them in a single, large JVM instance. Keeping track of that many autonomous actors would be a challenge, but if most of them are stateless workers, it can be managed, if necessary. Akka also supports clustering across thousands of nodes for very high scalability and availability requirements.

A common criticism of the actor model, including Akka, is the loss of type safety. Recall that the `Receive` type alias is `PartialFunction[Any,Unit]`, meaning it doesn't provide a way to narrow the type of messages an actor is allowed to receive. Therefore, if you send an unexpected message to an actor, you have to detect the problem at runtime. The compiler and the type system can't help enforce logical correctness. Similarly, all references between actors are `ActorRef`s, not specific `Actor` types.

Some attempts have been made to provide more restrictive typing, but without clear success so far. For most users, the loss of some type safety is compensated by the power and flexibility of the model.

It's also true that the actor model isn't a really a *functional programming* model. `Receive` returns `Unit`. Everything is done through side effects! Furthermore, the model embraces mutable state when useful, as in our `datastores`.

However, it's a strongly principled use of mutability, where the state is carefully encapsulated within an actor, whose manipulations of that state are guaranteed to be thread-safe. Messages between actors are expected to be immutable objects. Unfortunately, Scala and Akka can't enforce these principled constraints on mutability. It's up to you, but you have the tools to do it right.

It's interesting that the actor model is closely aligned with the vision of object-oriented programming espoused by Alan Kay, the coinventor of Smalltalk and the person who is believed to have cloned the term "object-oriented programming." He argued that objects should be autonomous encapsulations of state, which only communicate through message passing. In fact, invoking a method in Smalltalk was called *sending a message*.

Finally, the actor model is an example of a more general approach to large-scale, highly available, event-driven applications. But first, let's discuss two problems that arise when you distribute code between processes, along with two solutions for them, Pickling and Spores."

Pickling and Spores

A challenge of distributed programming is fast, controlled serialization and deserialization of data and code for movement around the cluster. This is an old problem and Java has had a built-in serialization mechanism since the beginning. However, far better performance is possible and the best choice involves balancing speed against other requirements. For example, does the format need to work with multiple languages, including non-JVM languages? Does it need to embed the schema and handle version changes?

The Scala Pickling library aims to provide a serialization option with minimal boilerplate in source code and a pluggable architecture for different backend formats.

We discussed a related problem earlier when describing the `Props` type in Akka, controlling what gets captured in a closure (function literal) when that closure will be distributed outside the process. The Spores project aims to solve this problem with an API that a developer uses to explicitly construct a "spore" (safe closure), where correctness is

enforced by the API. More information about the project, with motivating examples, can be found at in the Scala docs.

Both projects are currently under development and they may appear in a future release of Scala, possibly as separate libraries.

Reactive Programming

It's been recognized for a long time that large-scale applications must be *event driven* in some sense, meaning they need to respond to requests for service and send events (or messages) to other services when they need "help." The Internet is built on this premise. Such systems have been called *reactive*, because of their responsive nature, rather than attempting to drive how they work according to some internal logic.

A number of models have emerged that embrace this core principle in different ways. In addition to the actor model, here are two popular models, both of which embrace purity more than the actor model, which considers mutable state acceptable, as long as each example is localized within an actor:

Functional Reactive Programming (FRP)
: FRP is an early *dataflow* model developed first in Haskell by Conal Elliott and Paul Hudak for graphics applications, where time-oriented state updates need to propagate through a system. Rather than manually updating variables as the state they depend on changes, in FRP the dependencies between data elements are declared and the runtime manages state propagation for you. Hence, the user writes code using functional-style declarative and compositional idioms. More recently, FRP has been implemented in a language called *Elm* by Evan Czaplicki, targeting JavaScript. The paper "Deprecating the Observer Pattern" examines a similar model in Scala.

Reactive Extensions (Rx)
: Rx was developed by Erik Meijer and collaborators for .NET. It has since been ported to multiple languages, including Java and Scala (Li Haoyi's project). Rx composes asynchronous programs using observable sequences representing event streams or other data sources, along with query operators (combinators) provided by a library called LINQ (language-integrated query).

Recently, the *Reactive Manifesto* was organized in an attempt to put some concrete definitions around what a "reactive" system should be. It defines four characteristics that scalable, resilient, reactive applications should support:

Message or Event Driven
: As a baseline, the system has to be designed to respond to messages or events (for some definition of those terms).

Elastically Scalable
: The system has to scale to meet demand, which inevitably means horizontal scaling across processes, then cores, then nodes. Ideally, this process should happen dynamically in response to changing demand, both growing and shrinking. The characteristics of networks (such as performance and reliability) then become *first-class* concerns in the architecture of such systems. Services that maintain nontrivial state are hard to scale horizontally this way and it can be difficult to "shard" the state or replicate it reliably.

Resilient
: Rare events become commonplace as the size of the system grows. Hence, failures must also be *first-class* concerns. The system must be engineered from the ground up to recover gracefully from failures.

Responsive
: The system needs to be available to respond to service requests, even if *graceful degradation* is necessary in the face of failed components or extremely high traffic.

Actors, FRP, and Rx are all event-based. All have been scaled in various ways, although FRP and Rx are more

oriented toward processing pipelines for individual streams of events, rather than networks of interacting components, like actors. Arguably, the actor model offers the strongest support for responsiveness, due to its robust error-handling strategy. Finally, these systems approach responsiveness in various ways, but all of them seek to minimize blocking.

Recap and What's Next

We learned how to build scalable, robust, concurrent applications using Akka actors for large-scale systems. We also learned about Scala's support for process management and futures. Finally, we discussed the general idea of *reactive* programming, of which actors are an example, and discussed two other popular models, FRP and Rx.

The next chapter examines one of the hottest areas of our industry today, *Big Data*, and why Scala has emerged as the de facto programming language in this field.