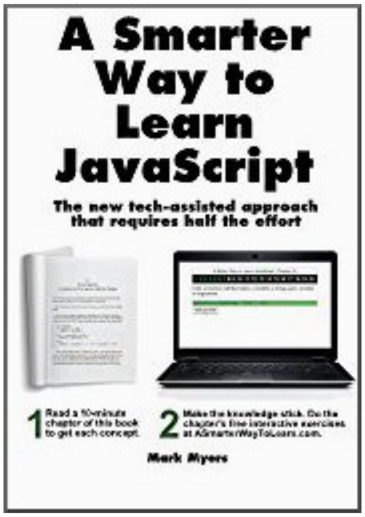


more scala

general

recursion examples
using match like switch
current date/time
if/then/else
ternary operator
for loop and yield
curly brace packaging
add methods to existing classes
spring framework dependency injection

classes and methods

creating javabeans
importing java code
multiple constructors

categories

alaska (25)
android (138)
best practices (63)
career (50)
colorado (21)
cvs (27)
design (33)
drupal (120)
eclipse (6)
funny (3)
gadgets (108)
git (15)
intellij (4)
java (429)
jdbc (26)
swing (74)
jsp (9)
latex (26)
linux/unix (289)
mac os x (315)
mysql (54)
ooa/ood (11)
perl (156)

Simple concurrency with Scala Futures (Futures tutorial)

By Alvin Alexander. Last updated: August 20 2017

named and default parameters

calling methods

class casting

equivalent of java .class

rename classes on import

private primary constructor

try/catch/finally

try, catch, and finally syntax

declare a null var before try/catch

collections

mutable arrays

string arrays

convert array to string

data types

convert java collections to scala

multidimensional arrays (2D array)

iterating over lists (foreach, for)

iterating over maps

convert array to string with mkstring

list

list, foreach, and for

merging lists

creating lists

list examples

add elements to a list

the filter method

map

map class examples

iterating over maps

php (97)
postgresql (17)
programming (43)
ruby (56)
scala (640)
sencha (23)
servlets (10)
technology (84)
testing (13)
uml (24)
zen (47)

This is an excerpt from the Scala Cookbook (partially modified for the internet). This is Recipe 13.9, “Simple concurrency with Scala Futures.”

Table of Contents

Table of Contents ▴▾

Problem

You want a simple way to run one or more tasks concurrently in a Scala application, including a way to handle their results when the tasks finish. For instance, you may want to make several web service calls in parallel, and then work with their results after they all return.

Solution

A `Future` gives you a simple way to run an algorithm concurrently. A future starts running concurrently when you create it and returns a result at some point, well, in the future. In Scala, it’s said that a future returns “eventually.”

tuple

tuple examples

map tuples in anonymous function

strings

multiline strings

string arrays

string formatting

convert array to string

split string example

convert string to int

compare strings with ==

a 'chomp' method

find regex in string

functions and functional programming

named and default parameters

pass one function to another

pass a function to a function (swing)

files

open and read files

shell script example

command line and scripts

read command line arguments

execute (exec) system commands

prompting a user, reading input

make scripts run faster

show more methods in repl

show more info on classes/objects in repl

The following examples show a variety of ways to create futures and work with their eventual results.

Run one task, but block

This first example shows how to create a future and then block to wait for its result.

Blocking is not a good thing — you should block only if you really have to — but this is useful as a first example, in part, because it's a little easier to reason about, and it also gets the bad stuff out of the way early.

The following code performs the calculation `1 + 1` at some time in the future. When it's finished with the calculation, it returns its result:

```
package actors

// 1 - the imports
import scala.concurrent.{Await, Future}
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global

object Futures1 extends App {

  // used by 'time' method
  implicit val baseTime = System.currentTimeMillis

  // 2 - create a Future
  val f = Future {
      sleep(500)
      1 + 1
  }

  // 3 - this is blocking (blocking is bad)
  val result = Await.result(f, 1 second)
  println(result)
  sleep(1000)

}
```

paste multiline commands in repl

database

jdbc connection, select

actors and concurrency

akka 'hello world'

akka ping-pong example

stop/quit an actor

stop actor and shut down

akka futures

akka ask, future, await, timeout

parallel collections, .par, and performance

akka remote example

akka remote - objects as messages

idioms

using option, some, and none

methods should have no side effects

prefer immutable code

email

imap client (using ssl and imaps)

imap client with search

play framework

play framework recipes

deploy to production

json method in controller

creating crud forms

@textarea rows and columns

convert objects to json

run play on different port

Here’s how this code works:

- The `import` statements bring the code into scope that’s needed.
- The `ExecutionContext.Implicits.global` import statement imports the “default global execution context.” You can think of an execution context as being a thread pool, and this is a simple way to get access to a thread pool.
- A `Future` is created after the second comment. Creating a `Future` is simple; you just pass it a block of code you want to run. This is the code that will be executed at some point in the future.
- The `Await.result` method call declares that it will wait for up to one second for the `Future` to return. If the `Future` doesn’t return within that time, it throws a `java.util.concurrent.TimeoutException`.
- The `sleep` statement at the end of the code is used so the program will keep running while the `Future` is off being calculated. You won’t need this in real-world programs, but in small example programs like this, you have to keep the JVM running.

I created the `sleep` method in my *package object* while creating my future and concurrency examples, and it just calls `Thread.sleep`, like this:

```
def sleep(time: Long) { Thread.sleep(time) }
```

As mentioned, *blocking* is bad; you shouldn’t write code like this unless you have to. The following examples show better approaches.

populate data on startup
using map in template
template comments
template functions
404 and 500 errors
mapping field validators
web service request with timeout
play console commands/help
testing web services with curl
logout, destroy session
read cookies

web services

rest client using apache httpclient
twitter client example
json parsing using lift-json
json array parsing using lift-json
lift framework form examples

xml

create xml literal
generate dynamic xml
xml - pretty printing
xml - save to file
xml - serialize, deserialize
xml - load a file
xml - load a url
xml - xpath searching
searching xmlns namespaces, xpath
xml - extract data from nodes
xml - extract data from arrays

The code also shows a time duration of `1 second`. This is made available by the `scala.concurrent.duration._` import. With this library, you can state time durations in several convenient ways, such as `100 nanos`, `500 millis`, `5 seconds`, `1 minute`, `1 hour`, and `3 days`. You can also create a duration as `Duration(100, MILLISECONDS)`, `Duration(200, "millis")`.

Run one thing, but don’t block, use callback

A better approach to working with a future is to use its callback methods. There are three callback methods: onComplete, onSuccess, and onFailure. The following example demonstrates onComplete:

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

object Example1 extends App {
    println("starting calculation ...")
    val f = Future {
        sleep(Random.nextInt(500))
        42
    }
    println("before onComplete")
    f.onComplete {
        case Success(value) => println(s"Got the callback, mea
        case Failure(e) => e.printStackTrace
    }
    // do the rest of your work
    println("A ..."); sleep(100)
    println("B ..."); sleep(100)
    println("C ..."); sleep(100)
    println("D ..."); sleep(100)
    println("E ..."); sleep(100)
    println("F ..."); sleep(100)
    sleep(2000)
}
```

xml – parsing, tags
xml – using match expressions
xml – many examples

build, testing, and debugging

sbt documentation (pdf)
show sbt history
scalatest – installing
scalatest – writing tdd tests
scalatest – writing bdd tests
scalatest – given/when/then with bdd
scalatest – test suite
scalatest – expected, actual
scalatest – mark test as pending
scalatest – testing exceptions
scalatest – tagging tests
scalatest – disabling tests
scalatest – mock objects
scalatest – running in eclipse

This example is similar to the previous example, though it just returns the number `42` after a random delay. The important part of this example is the `f.onComplete` method call and the code that follows it. Here's how that code works:

- The `f.onComplete` method call sets up the callback. Whenever the `Future` completes, it makes a callback to `onComplete`, at which time that code will be executed.
- The `Future` will either return the desired result (`42`), or an exception.
- The `println` statements with the slight delays represent other work your code can do while the `Future` is off and running.

Because the `Future` is off running concurrently somewhere, and you don't know exactly when the result will be computed, the output from this code is nondeterministic, but it can look like this:

```
starting calculation ...
before onComplete
A ...
B ...
C ...
D ...
E ...
Got the callback, meaning = 42
F ...
```

Because the `Future` returns *eventually*, at some nondeterministic time, the "Got the callback" message may appear anywhere in that output.

The onSuccess and onFailure callback methods

There may be times when you don't want to use `onComplete`, and in those situations, you can use the `onSuccess` and `onFailure` callback methods, as shown in this example:

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

object OnSuccessAndFailure extends App {
    val f = Future {
        sleep(Random.nextInt(500))
        if (Random.nextInt(500) > 250) throw new Exception("Yikes!") else 42
    }
    f onSuccess {
        case result => println(s"Success: $result")
    }
    f onFailure {
        case t => println(s"Exception: ${t.getMessage}")
    }

    // do the rest of your work
    println("A ..."); sleep(100)
    println("B ..."); sleep(100)
    println("C ..."); sleep(100)
    println("D ..."); sleep(100)
    println("E ..."); sleep(100)
    println("F ..."); sleep(100)
    sleep(2000)
}
```

This code is similar to the previous example, but this `Future` is wired to throw an exception about half the time, and the `onSuccess` and `onFailure` blocks are defined as *partial functions*; they only need to handle their expected conditions.

Creating a method to return a Future[T]

In the real world, you may have methods that return futures. The following example defines a method named `longRunningComputation` that returns a `Future[Int]`.

Declaring it is new, but the rest of this code is similar to the previous `onComplete` example:

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

object Futures2 extends App {
    implicit val baseTime = System.currentTimeMillis

    def longRunningComputation(i: Int): Future[Int] = future {
        sleep(100)
        i + 1
    }

    // this does not block
    longRunningComputation(11).onComplete {
        case Success(result) => println(s"result = $result")
        case Failure(e) => e.printStackTrace
    }

    // important: keep the jvm from shutting down
    sleep(1000)
}
```

The `future` method shown in this example is another way to create a `Future`. It starts the computation asynchronously and returns a `Future[T]` that will hold the result of the computation. This is a common way to define methods that return a future.

How to use multiple Futures in a for loop

The examples so far have shown how to run one computation in parallel, to keep things simple. You may occasionally do something like this, such as writing data to a database without blocking the web server, but many times you'll want to run several operations concurrently, wait for them all to complete, and then do something with their combined results.

For example, in a stock market application I wrote, I run all of my web service queries in parallel, wait for their results, and then display a web page. This is faster than running them sequentially.

The following example is a little simpler than that, but it shows how to call an algorithm that may be running in the cloud. It makes three calls to `Cloud.runAlgorithm`, which is defined elsewhere to return a `Future[Int]`. For the moment, this algorithm isn't important, other than to know that it prints its result right before returning it.

The code starts those three futures running, then joins them back together in the for-comprehension:

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

object RunningMultipleCalcs extends App {
    println("starting futures")
    val result1 = Cloud.runAlgorithm(10)
    val result2 = Cloud.runAlgorithm(20)
    val result3 = Cloud.runAlgorithm(30)

    println("before for-comprehension")
    val result = for {
        r1 <- result1
        r2 <- result2
        r3 <- result3
    } yield (r1 + r2 + r3)

    println("before onSuccess")
    result onSuccess {
        case result => println(s"total = $result")
    }

    println("before sleep at the end")
    sleep(2000)  // important: keep the jvm alive
}
```

Here’s a brief description of how this code works:

- The three calls to `Cloud.runAlgorithm` create the `result1`, `result2`, and `result3` variables, which are of type `Future[Int]`.
- When those lines are executed, those futures begin running, just like the web service calls in my stock market application.
- The for-comprehension is used as a way to join the results back together. When all three futures return, their `Int` values are assigned to the variables `r1`, `r2`, and `r3`, and the sum of those three values is returned from the `yield` expression, and assigned to the `result` variable.

- Notice that result can’t just be printed after the for-comprehension. That’s because the for-comprehension returns a new `Future`, so result has the type `Future[Int]`. (This makes sense in more complicated examples.) Therefore, the correct way to print the example is with the `onSuccess` method call, as shown.

When this code is run, the output is nondeterministic, but looks something like this:

```
starting futures
before for-comprehension
before onSuccess
before sleep at end
returning result from cloud: 30
returning result from cloud: 20
returning result from cloud: 40
total = 90
```

Notice how all of the `println` statements in the code print before the total is printed. That’s because they’re running in sequential fashion, while the future is off and running in parallel, and returns at some indeterminate time (“eventually”).

I mentioned earlier that the `Cloud.runAlgorithm` code wasn’t important — it was just something running “in the cloud,” — but for the sake of completeness, here’s that code:

```
object Cloud {
    def runAlgorithm(i: Int): Future[Int] = future {
        sleep(Random.nextInt(500))
        val result = i + 10
        println(s"returning result from cloud: $result")
        result
    }
}
```

In my real-world code, I use a future in a similar way to get information from web services. For example, in a Twitter client, I make multiple calls to the Twitter web service API using futures:

```
// get the desired info from twitter
val dailyTrendsFuture = Future { getDailyTrends(twitter) }
val usFuture = Future { getLocationTrends(twitter, woeidUnitedStates) }
val worldFuture = Future { getLocationTrends(twitter, woeidWorld) }
```

I then join them in a for comprehension, as shown in this example. This is a nice, simple way to turn single-threaded web service calls into multiple threads.

Discussion

Although using a future is straightforward, there are also many concepts behind it. The following sections summarize the most important concepts.

A future and ExecutionContext

The following statements describe the basic concepts of a future, and the ExecutionContext that a future relies on.

- A `Future[T]` is a container that runs a computation concurrently, and at some future time may return either (a) a result of type `T` or (b) an exception.
- Computation of your algorithm starts at some nondeterministic time after the future is created, running on a thread assigned to it by the execution context.
- The result of the computation becomes available once the future completes.
- When it returns a result, a future is said to be *completed*. It may either be successfully completed, or failed.
- As shown in the examples, a future provides an interface for reading the value that has been computed. This includes callback methods and other approaches, such as a for-comprehension, `map`, `flatMap`, etc.
- An `ExecutionContext` executes a task it’s given. You can think of it as being like a thread pool.
- The `ExecutionContext.Implicits.global` import statement shown in the examples imports the default global execution context.

Callback methods

The following statements describe the use of the callback methods that can be used with futures.

- Callback methods are called asynchronously when a future completes.
- The callback methods `onComplete`, `onSuccess`, `onFailure`, are demonstrated in the Solution.
- A callback method is executed by some thread, some time after the future is completed. From the Scala Futures documentation, "There is no guarantee that it will be called by the thread that completed the future or the thread that created the callback."
- The order in which callbacks are executed is not guaranteed.
- `onComplete` takes a callback function of type `Try[T] => U`.
- `onSuccess` and `onFailure` take partial functions. You only need to handle the desired case. (See Recipe 9.8, "Creating Partial Functions" for more information on partial functions.)
- `onComplete`, `onSuccess`, and `onFailure` have the result type `Unit`, so they can't be chained. This design was intentional, to avoid any suggestion that callbacks may be executed in a particular order.

For-comprehensions (combinators: map, flatMap, filter, foreach, recoverWith, fallbackTo, andThen)

As shown in the Solution, callback methods are good for some purposes. But when you need to run multiple computations in parallel, and join their results together when they're finished running, using *combinators* like `map`, `foreach`, and other approaches — like a for-comprehension — provides more concise and readable code. The for-comprehension was shown in the Solution.

The `recover`, `recoverWith`, and `fallbackTo` combinators provide ways of handling failure with futures. If the future they're applied to returns successfully, you get that (desired) result, but if it fails, these methods do what their names suggest, giving you a way to recover from the failure.

As a short example, you can use the `fallbackTo` method like this:

```
val meaning = calculateMeaningOfLife() fallbackTo 42
```

The `andThen` combinator gives you a nice syntax for running whatever code you want to run when a future returns, like this:

```
var meaning = 0
future {
    meaning = calculateMeaningOfLife()
} andThen {
    println(s"meaning of life is $meaning")
}
```

See the Scala Futures documentation for more information on their use.

See Also

- The Scala Futures documentation
- These examples (and more) are available at my GitHub repository.
- As shown in these examples, you can read a result from a future, and a promise is a way for some part of your software to put that result in there. I've linked to the best article I can find at alvinalexander.com/bookmarks/scala-futures-and-promises

The Scala Cookbook

This tutorial is sponsored by the *Scala Cookbook*, which I wrote for O'Reilly:

You can find the Scala Cookbook at these locations:

- Here on the O'Reilly website, and
- Here on Amazon.com

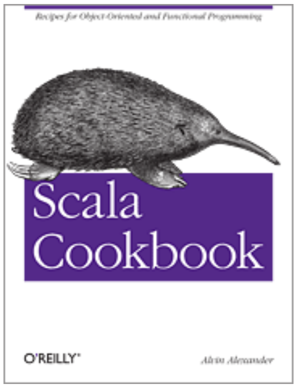
category: scala

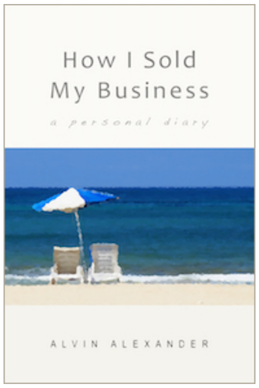
tags: scala cookbook scala oncomplete future concurrency callback recoverwith fallbackto andthen onsuccess onfailure executioncontext nondeterministic deterministic parallel concurrent

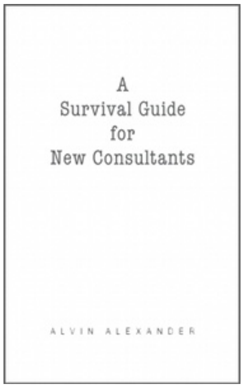
related

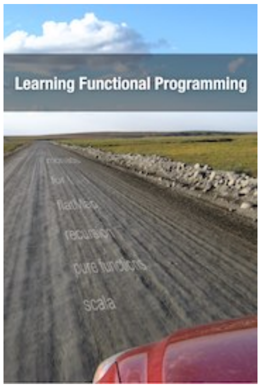
- A complete Scala Future example from the Scala Cookbook
- How to use multiple Futures in a Scala for-comprehension
- How to create a Twitter client in Scala
- The differences between a Scala Future and a Java Thread
- On the Scala Future, and semantics
- Examples of how to use parallel collections in Scala

books i've written









what's new

- Mindfulness metaphor
- I've seen a lot of my friends lose their passion and end up in a rut
- Thought it would be fun to play tennis again
- Enjoying whatever time we have left
- Dr. Jill Carnahan of Louisville, Colorado, has a good article on MCAS
- Richard Feynman on Cargo Cult Science

Add new comment

Your name

Email

The content of this field is kept private and will not be shown publicly.

Homepage

Subject

Comment

About text formats

Allowed HTML tags: <em> <strong> <cite> <code> <ul type> <ol start type> <li> <pre>
Lines and paragraphs break automatically.

By submitting this form, you accept the Mollom privacy policy.

Save Preview

Links:
front page me on twitter search privacy

java

- java applets
- java faqs
- misc content
- java source code
- test projects
- lejos

Perl

- perl faqs
- programs
- perl recipes
- perl tutorials

Unix

- man (help) pages
- unix by example
- tutorials

source code warehouse

- java examples
- drupal examples

misc

- privacy policy
- terms & conditions
- subscribe
- unsubscribe
- wincvs tutorial
- function point analysis (fpa)
- fpa tutorial

Other

- mobile website
- rss feed
- my photos
- life in alaska
- how i sold my business
- living in talkeetna, alaska
- my bookmarks
- inspirational quotes
- source code snippets