Reactive Programming

# Map, FlatMap, and For Comprehension

Suppose we have

val l1 = List(“one”, “two”, “three”)

val l2 = List(“a”, “b”, “c”)

Let’s write a For Comprehension to generate all possible pairs containing 1 element from l1 and 1 element from l2.

for (x <- l1; y <-l2) yield (x, y)

Now let’s try using map and flatMap to do the same thing.

l1 flatMap { x => l2 map { y => (x, y) } }

What about

for (x <- 1 until 4; y <- 5 until 8) yield x + y

Can we write that using Map and FlatMap? Yes we can.

Range(1,4) flatMap { x => Range(5,8) map { y => x + y } }

Recall that we can use map and flatMap on Option. Here’s an example which traverses associations in a map:

val m = Map("w" -> "x", "x" -> "y", "y" -> "z")

m.get("w") // Returns Some("x")

m.get("w").flatMap(m.get(\_)).flatMap(m.get(\_)) // returns Some("z")

m.get("a").flatMap(m.get(\_)).flatMap(m.get(\_)) // returns None

Can we write a for comprehension to do this? Yes we can.

for (x <- m.get("w");

y <- m.get(x);

z <- m.get(y)) yield z

Is this a coincidence that map and flatMap can be written as for comprehensions. No it’s not. It turns out that for comprehensions are just syntactic sugar for map, flatMap and filter. As long as the collection supports specific requirements, such as implementing map, flatMap and filter, you can use for comprehensions instead. Most container objects fit this pattern, including the ones we’ll be discussing shortly.

# Try

Scala has a Java-style way of handling exceptions:

case class NotOddException(msg: String) extends IllegalArgumentException(msg)

def onlyOdds(x: Int) =

if (x % 2 == 0) throw new NotOddException("Not odd enough") else x

def tryCatcher(x: Int) = try {

onlyOdds(x)

} catch {

case NotOddException(msg) => msg

}

tryCatcher(1)

tryCatcher(2)

This style of try/catch is ok for non-concurrent code, but doesn’t work so well for concurrent code. If an exception happens in an asynchronous call, we need to capture the exception so that it can be used by downstream processes. This is called materializing the exception. Scala provides a type which performs this function, called Try.

A Try has a type parameter representing the result of a successful computation. For example,

val t: Try[String] = Try("hey")

Try has 2 subtypes, Success and Failure. Success wraps a successful result, and Failure wraps a Throwable in the event of a Failure. Try is very similar to Option, and it can be useful to think of Option when trying to understand Try. In this comparison, Success is like Some, and Failure is like None.

A few examples:

Success("hey")

Failure(new IllegalArgumentException("too big"))

def tryOnlyOdds(x:Int): Try[Int] = Try(onlyOdds(x))

tryOnlyOdds(1)

tryOnlyOdds(2)

Our friend flatMap works…

tryOnlyOdds(1).flatMap { x => tryOnlyOdds(x + 2) }

tryOnlyOdds(2).flatMap { x => tryOnlyOdds(x + 2) }

As do for comprehensions…

for (x <- tryOnlyOdds(1);

y <- tryOnlyOdds(x + 2)) yield y

You can also use pattern matching

def matchOnlyOdds(x: Int) = tryOnlyOdds(x) match {

case Success(x) => s"$x is odd"

case Failure(ex) => s"$x is even, because $ex"

}

matchOnlyOdds(1)

matchOnlyOdds(2)

# Future

A Future is a datastructure that will eventually contain a result. That result may be a success or a failure. Futures are the fundamental building block of asynchronous programs.

A basic way to create a Future is to call the apply method on the Future companion object with a block.

val f = Future { "in the future" }

If we care about the result of the future, we can register a callback.

f.onSuccess { case x => println(s"x is $x") }

Also we can register a callback for failure

val f = Future { onlyOdds(2) }

f.onFailure { case NotOddException(x) => println(s"x is $x") }

Alternatively we can register an onComplete callback which takes a Try and handles both cases.

def oddFuture(x:Int): Future[Int] = Future { onlyOdds (x) }

val f1 = oddFuture(1)

val f2 = oddFuture(2)

val oddCallback: Try[Int] => Unit = {

case Success(x) => println(s"oddCallback result is $x")

case Failure(ex) => println(s"oddCallback failure is ${ex.getMessage}")

}

f1.onComplete(oddCallback)

f2.onComplete(oddCallback)

Suppose we wanted to chain a series of asynchronous computations. For example, we call oddFuture on a number, then use the result of that future as input to another call to oddFuture, and so on. Using callbacks might look like:

oddFuture(1).onSuccess {

case x => oddFuture(x + 2).onSuccess {

case y => oddFuture(y + 2).onSuccess {

case z => println(s"the final result is $z")

}

}

}

That’s a lot of nesting and is pretty ugly. There’s a cleaner way to chain futures, using flatMap on Future.

oddFuture(1).flatMap{ x => oddFuture(x + 2) }.flatMap{ x => oddFuture(x + 2) }.onSuccess{ case x => println(s"the final result is $x") }

Or alternatively, we can use For Comprehensions:

val f = for (x <- oddFuture(1);

y <- oddFuture(x + 2);

z <- oddFuture(y + 2)) yield z

f.onSuccess { case x => println(s"the final result is $x") }

Or in the failure case,

val f = for (x <- oddFuture(2);

y <- oddFuture(x + 2);

z <- oddFuture(y + 2)) yield z

f.onSuccess { case x => println(s"the final result is $x") }

Here, onSuccess is never called since the first line fails. Let’s register an onFailure handler

f.onFailure { case x => println(s"Failed with $x") }

Suppose we have independent computations that can happen in parallel. In this case, we setup the future computation outside of the for comprehension.

val f1 = oddFuture(1)

val f2 = oddFuture(3)

val f3 = for (x <- f1;

y <- f2;

z <- oddFuture(x + y + 1)) yield z

f3.onSuccess { case x => println(s"the final result is $x") }

At some point, you may want to get the result out of a future. To do this, use Await.result, with a timeout. This blocks until the result is returned.

import scala.concurrent.\_

import scala.concurrent.duration.\_

Await.result(f3, 5 seconds)

# Returning Async Result in Play

Recall that we did blocking I/O when we fetched a Rest.li result in our Play app. Let’s now do async I/O by returning a future. Modify the existing fetchProfileStrength method from

def fetchProfileStrength(memberId: Long)(implicit request: RequestHeader) :Long = {

val memberUrnString = s"urn:li:member:$memberId"

val restliRequest = new ProfileStrengthsBuilders().get().id(new Urn(memberUrnString)).build()

val futureResponse = RestliPlugin.getInstance.sendRequest(restliRequest)

val profileStrengthInfo = Await.result(futureResponse, 5 seconds).getEntity()

profileStrengthInfo.getScore()

}

to

def fetchProfileStrength(memberId: Long)(implicit request: RequestHeader): Future[Long] = {

val memberUrnString = s"urn:li:member:$memberId"

val restliRequest = new ProfileStrengthsBuilders().get().id(new Urn(memberUrnString)).build()

val futureResponse = RestliPlugin.getInstance.sendRequest(restliRequest)

futureResponse.map { resp => resp.getEntity().getScore }

}

Here we’re chaining the future which returns the rest.li result with a future that grabs the score from the rest.li result and returning the Future containing the score.

Modify the restliProfileStrengthForMember client from

def restliProfileStrengthForMember(memberId: Long) = Action { implicit request =>

val score = fetchProfileStrength(memberId)

val profileStrength = new ProfileStrength().setScore(score)

Ok(JsonUtil.toJsValue(profileStrength))

}

to

def restliProfileStrengthForMember(memberId: Long) = Action.async { implicit request =>

val futureScore = fetchProfileStrength(memberId)

futureScore.map { score =>

val profileStrength = new ProfileStrength().setScore(score)

Ok(JsonUtil.toJsValue(profileStrength))

}

}

Here we’re constructing a Future[Result], and using Play’s Action.async constructor to create an action which can process a Future[Result].

Do the same treatment for processForm. Change

def processForm = Action { implicit request =>

val member = memberForm.bindFromRequest.get

val memberId = member.memberid

val score = fetchProfileStrength(memberId)

renderForm(Map("mid" -> memberId, "score" -> score))

}

to

def processForm = Action.async { implicit request =>

val member = memberForm.bindFromRequest.get

val memberId = member.memberid

val futureScore = fetchProfileStrength(memberId)

futureScore.map { score =>

renderForm(Map("mid" -> memberId, "score" -> score))

}

}

Restart play (play run), and hit <http://localhost:9000/demo1/processprofilestrengthform>

# Promises

Futures provide an interface for querying a result of a deferred computation. Promises let you put a result inside a Future. You may put a result inside a future at most once. To use a promise, create a Promise, then retrieve the Future associated with that Promise.

val p = Promise[Fortune]()

val f = p.future

f.onSuccess { case x => println(s"Your fortune is: $x") }

p.success(Fortune("You will be rich and famous"))