Let's talk Monads

Scala training, 2016

Outline

Intro

Optional values vs. *null*

Failure is not an Option, it's actually a *Try*

Back to *Future[T]*

What's a monad?

- Remember List?

```
// for comprehension
for { x <- 1 to 10; y <- 1 to x } yield x * y

// raw
(1 to 10).flatMap(x => (1 to x).map(y => x * y))
```

- Does *map*, flatMap or *filter ring a bell?*

What's a monad?

- Remember List?

```
// for comprehension
for { x <- 1 to 10; y <- 1 to x } yield x * y

// raw
(1 to 10).flatMap(x => (1 to x).map(y => x * y))
```

- Does *map*, flatMap or *filter ring a bell?*
- These enable for comprehensions

What's a monad?

- Remember List?

```
// for comprehension
for { x <- 1 to 10; y <- 1 to x } yield x * y

// raw
(1 to 10).flatMap(x => (1 to x).map(y => x * y))
```

- Does *map*, flatMap or *filter ring a bell?*
- These enable for comprehensions
- Define these functions for other data type and voila - you get for comprehensions

What's a monad?

- Remember List?

```
// for comprehension
for { x <- 1 to 10; y <- 1 to x } yield x * y

// raw
(1 to 10).flatMap(x => (1 to x).map(y => x * y))
```

- Does *map*, flatMap or *filter ring a bell?*
- These enable for comprehensions
- Define these functions for other data type and voila - you get for comprehensions
- Let's simply call these **monads**

What's a monad?

- Remember List?

```
// for comprehension
for { x <- 1 to 10; y <- 1 to x } yield x * y

// raw
(1 to 10).flatMap(x => (1 to x).map(y => x * y))
```

- Does *map*, flatMap or *filter ring a bell?*
- These enable for comprehensions
- Define these functions for other data type and voila - you get for comprehensions
- Let's simply call these **monads**
- Are there any other useful monads?

The Billion Dollar Mistake

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

Tony Hoare, inventor of the null reference

Avoiding null checks

- What data structure could we use to avoid null checks?

Avoiding null checks

- What data structure could we use to avoid null checks?
- Hint: a collection with at most one element

Avoiding null checks

- What data structure could we use to avoid null checks?
- Hint: a collection with at most one element
- Differences between null and an absent value

Avoiding null checks

- What data structure could we use to avoid null checks?
- Hint: a collection with at most one element
- Differences between null and an absent value
- Let's define map / flatMap / filter in our data structure

Hello, Option[T]

```scala
sealed abstract class Option[A]

final case class Some[A](x: A) extends Option[A]
case object None extends Option[Nothing]
```

- The Some case class is a wrapper for values
- None is a singleton object representing an absent value

Hello, Option[T]

```
sealed abstract class Option[A]

final case class Some[A](x: A) extends Option[A]
case object None extends Option[Nothing]
```

- The Some case class is a wrapper for values
- None is a singleton object representing an absent value
- How do we create options?
  - Using Some and / or None directly
  - Using Option's apply method when working with Java APIs to avoid `Some(null)`

Hello, Option[T]

```scala
sealed abstract class Option[A]

final case class Some[A](x: A) extends Option[A]
case object None extends Option[Nothing]
```

- The Some case class is a wrapper for values
- None is a singleton object representing an absent value
- How do we create options?
  - Using Some and / or None directly
  - Using Option's apply method when working with Java APIs to avoid `Some(null)`
- **Q**: Why is `Some(null)` a problem?

Pattern Matching Options

- Optional values can be properly handled with pattern matching

```
val phoneBook = Map('A' -> List("Andrei" -> "123"))
def contactsOf(first: Char) =
  phoneBook.get(first) match {
    case Some(contacts) => contacts
    case None           => Nil
  }

// List((Andrei,123))
contactsOf('A')

// List()
scala> contactsOf('D')
```

Another kind of collection

- Option[T] is just a container for a value of type T
- Think of an option as a special collection containing:
  - zero elements when the value is absent
  - exactly one element when the value is present
- Mostly all collection functions are implemented here as well
- Use them in for comprehensions

Mapping an option

- map
  - on lists: turns List[A] into List[B]
  - on options: turns Option[A] into Option[B]
- Think of None being the equivalent of an empty list
  - Map an empty List[A] to get an empty List[B]
  - Map an absent (None) Option[A] to get an absent (None) Option[B]

```
val phoneBook = Map('A' > List("Andrei" -> "123"))
phoneBook.get('A').map(_.size) // Some(1)
```

flatMap on Options

- flatMap
  - on lists: turns List[List[A]] into List[B]
  - on options: turns Option[Option[A]] into Option[B]

```
// Some(Some((Andrei, 123)))
phoneBook.get('A').map(_.headOption)

// Some((Andrei, 123))
phoneBook.get('A').flatMap(_.headOption)
```

- What's the return type when calling flatMap on List[Option[T]]?

Filtering

- Options can be filtered just as lists
- Filtering an absent value results immediately in an absent value
- If the predicate does not hold for the present value, the result is None

```
// Some(List((Andrei, 123)))
phoneBook.get('A').filter(_.nonEmpty)
```

For comprehensions

- Option can be treated as a collection
- It provides map, flatMap and filter
- Hence, it can be used in for comprehensions
- It's the preferred way of working when chaining multiple map / flatMap / filter calls

```
for {
  contacts <- phoneBook.get('A')
  first    <- contacts.headOption
} yield first
// Some((Andrei, 123))
```

Chaining Options

- Options can be chained using *orElse*
- If the former is None orElse returns the Option passed to it, otherwise it returns the one on which it was called
- Use case: search for resources in multiple paths

```
phoneBook.get('A').orElse(Some(Nil))
```

Dealing with failure

- What do you do when something goes wrong?
  - C / C++ blow with segfaults, stack smashes etc
  - What does Java do about these?

Exceptions in Scala

- As in Java, Scala offers support for exceptions
- Checked vs unchecked exceptions

```scala
class AssertException(msg: String) extends Exception {
  override def getMessage: String = msg
}

def myAssert(pred: => Boolean): Unit =
  if (!pred) throw new AssertException("Failed.")

try { myAssert(1 + 1 == 3) } catch {
  case ex: AssertException => ex.getMessage
}
```

Facts about failure in Scala

- There are only *unchecked exceptions in Scala*
- There are no throws declarations
- Exceptions still blow in your face
- Objective: prevent and overcome failure, in the functional way

Let me Try[T]

```scala
sealed abstract class Try[T]

final case class Failure[T](e: Throwable) extends Try[T]
final case class Success[T](value: T) extends Try[T]
```

- Try handles computations that might fail in an elegant manners
- It resembles Option by being a container
- The difference here is that computations might throw exceptions or not
- Can Try catch every exception?

Handle Try with pattern matching

```scala
// Read entire file in scala
Try { scala.io.Source.fromFile("file.txt").mkString }
    match {
  case Success(lines) => println(lines)
  case Failure(t)     => println(t.getMessage)
}
```

- Warning: although you can read files like above, it's not recommended as resources are not closed

Mapping a Try

- Map transforms the value contained with the given function
- If the computation has failed, then the failure is passed immediately
- Otherwise the A is turned into a B

```
// Assume we have implemented this method
def readFile(file: String): Try[String] = ???

// To find out the length of the file
readFile("file.txt").map(_.length)
```

flatMap on Try

- Use it when you have nested computations that can fail
- Think of nested try blocks in Java
- Alternatively, computations that depend on other computations determine the use of flatMap

```scala
case class HttpSettings(host: String, port: Int)

// Assume we have implemented these methods
def bind(host: String, port: Int): Try[Unit] = ???
def configFrom(file: String): HttpSettings = ???

readFile("config.json")
  .map(configFrom)
  .flatMap(config => bind(config.host, config.port))
```

filter

- Try computations may be filtered just as Options
- If the computation is already failed, the failure is returned
- If the predicate does not hold, the computation will fail with a NoSuchElementException

```
Try(1).filter(_ % 2 == 0)
// Failure(NoSuchElementException)
```

Recovering from failure

- There are two options to recover from a failure:
  - *recover which is equivalent to* map
  - *recoverWith which is equivalent to* flatMap

```
// just for space considerations:
// PartialFunc = PartialFunction
// Thr = Throwable
sealed abstract class Try[T] {
 def recover[U](f: PartialFunc[Thr, U]): Try[U]
 def recoverWith[U](f: PartialFunc[Thr, Try[U]]): Try[U]
}
```

For Comprehensions

- Try provides map, flatMap and filter
- As a result it can be used within for comprehensions
- As with Option, it's the preferred way of chaining multiple primitive calls
- See below the flatMap example rewritten

```
for {
  settings <- readFile("config.json")
  config   =  configFrom(settings)
  binding  <- bind(config.host, config.port)
} yield binding
```

A Nightmare full of Locks

- Concurrency is hard
- *synchronized doesn't make our lives easier:*
  - Deadlock
  - Starvation
  - Race conditions
- What if we could reason about the result and less about the process?
  - Who handles syncs? - An ExecutorService
  - Create Callables, use Futures to access their results

Back to *Future[T]*

- What we don't like about Java's Future:
  - Check for cancellation / completion
  - Block to get the result within an optional time unit
- Scala's approach to concurrency: Futures / Actors
- It's easy to reason about concurrency with Scala's Future

```scala
trait Future[T] extends Awaitable[T]
object Future {
  def apply[T](body: => T)(implicit ec:
    ExecutionContext): Future[T]
}
```

Callbacks

- They're used when life is easy:
  - The result of a computation doesn't depend on another
  - No chained futures / nested callbacks needed
- The *onComplete callback handles both successful and unsuccessful computations*
- You can use onSuccess or onFailure as well

```
val longComputation = Future { ... }

longComputation onComplete {
  case Success(value) => println(s"Got $value")
  case Failure(thrwb) => println(s"Failed with
    ${thrwb.getMessage}")
}
```

Mapping futures

- Future[T] is also a container for a computation that:
  - Will eventually result in a value of type T
  - Will possibly fail with an exception
- map turns a Future[A] into a Future[B]
- The transformation is done only if Future[A] is successful

```scala
case class Profile(id: String, name: String)

def fetchProfile(id: String): Future[Profile] = ???

fetchProfile("123") map { profile => profile.name }
```

Staying Flat

- The computation may depend on some other future
- Keep things flat using flatMap
- The transformation succeeds only if both futures succeed

```
def friendsOf(profile: Profile): Future[List[Profile]]
    = ???

// Future[Future[List[Profile]]]
fetchProfile("123") map { friendsOf }

// Future[List[Profile]]
fetchProfile("123") flatMap { friendsOf }
```

Recovering from Failure

- Future exposes a recovery mechanism, just as Try's
- recover is the equivalent of map for failures
- recoverWith is the equivalent of flatMap for failures

```
fetchProfile("123") recover {
  // log the exception
  case _: ProfileNotFoundException => ???
}

fetchProfile("123") recoverWith {
  // retry fetch from another next server
  case _: ServerNotAvailable => ???
}
```

For Comprehensions

- Instead of flatMap, one could use for comprehensions to do the same thing:

```
for {
  profile <- fetchProfile("123")
  friends <- friendsOf(profile)
} yield friends
```

- Take care when instantiating futures that encapsulate parallel operations. What happends below?

```
for {
  p1 <- fetchProfile("1")
  p2 <- fetchProfile("2")
} yield (p1, p2)
```