Durable Stateful and Functional Programs

Chris Addie

Datacom TSS

July 20, 2016

Table of Contents

Table of Contents

Abstract and Audiance

This presentation will (hopefully) explain the concepts of:

- Event Sourcing
- The State Monad
- Combining the Two

Audiance:

- This presentation intended for programmers with some understand of functional programming principles
- Code samples will be in Scala

Table of Contents

Event Sourcing

- A pattern for developing persistent and durable applications
- Every change to application state is captured as an event object
- Event objects stored in the sequence they were applied
- Event objects are stored for the lifetime of the application state

Persistence

- Event sourcing enables application persistence by storing event objects outside the application state
- Application state can be recovered by *replaying* persisted event objects

Durability

- By persisting event objects *before* making the changes to application state that they represent, data loss can be prevented in the case of an power loss
- All changes to state become *transactional*. To rollback a transaction discard application state then replay events up to but not including the change to be rolled back

And More!

- Auditing: Event journal can be used as an audit log
- Temporal Query: Application state at any point in the past can be queried by replying the journal up to the desired point in time
- Rewriting History: If an error that has already been commited is discovered it can be corrected by removing it from the journal and replaying
- Snapshotting: Avoid having to replay the entire journal

Akka-Persistence

Akka-Persistence is a Scala implementation of event sourcing for creating persistent and durable *Actors*

```
type Receive = PartialFunction[Any, Unit]
abstract def receiveCommand: Receive
abstract def receiveRecover: Receive
persist[A](event: A)(handler: (A) => Unit): Unit
```

Persist is not a pure function :(

Table of Contents

Problem: Caching a slow RPC API

```
trait SlowApi {
  //this function is slow :(
  def rpc(i: Int): Int
}
class CachingService extends Actor {
  def cachedRequest(i: Int): Int = ???
  def receive = {
     case i: Int => sender ! cachedRequest(i)
   }
}
```

Stateful Applications: Imperitively

Obvious solution: functions mutate state as a side effect

```
class ImperitiveCachingService extends Actor {
  this: SlowApi =>
  //mutable state
  var cache = Map[Int, Int]()

  def checkCache(i: Int): Option[Int] = cache.get(i)
  def retrieve(i: Int): Int = {
    val result = rpc(i)
    cache = cache updated (i, result)
    result
  }
  def cachedRequest(i: Int) =
    checkCache(i) getOrElse retrieve(i)

  def receive = {
    case i: Int => sender ! cachedRequest(i)
  }
}
```

Stateful Applications: Imperitively

Pros:

- Extremely Concise
- Pretty readable

Cons:

- Functions can't be reused
- Functions aren't referentially transparent
- Functions aren't thread safe

Stateful Computations

A stateful computation is a function that takes some state and returns a value along with some new state

```
s -> (s, a)
```

Can we reframe our solution in terms of stateful computations?

State as a parameter

```scala
class FunctionalCachingService extends Actor {
  this: SlowApi =>
  type Cache = Map[Int, Int]
  var cache: Cache = Map()
  def checkCache(i: Int)(s: Cache): (Cache, Option[Int]) = (s, s.get(i))
  def retrieve(i: Int)(s: Cache): (Cache, Int) = {
    val result = rpc(i)
    (s updated (i, result), result)
  }
  def cachedRequest(i: Int)(s0: Cache) = checkCache(i)(s0) match {
    case (s1, Some(result)) => (s1, result)
    case (s1, None)         => retrieve(i)(s1)
  }
  def receive = {
    case i: Int => {
      val (newState, result) = cachedRequest(i)(cache)
      cache = newState
      sender ! result
    }
  }
}
```

State as a parameter

Pros:

- Application is now composed of pure functions
- Thread safety, referential transparency and re-use all acheived!

Cons:

- Functions that return tuples are akward to compose
- Argument lists are cluttered
- Easy to accidently return the wrong state
- Still can't call persist without breaking referential transparency

The State Monad

```scala
trait State[S, +A] {
  def run(initial: S): (S, A)
  //bind
  def flatMap[B](f: A => State[S, B]): State[S, B]
}

object State {
  //return
  def apply[S, A](f: S => (S, A)): State[S, A]
}
```

Naive StateMonad

```
class MonadicCachingService extends Actor {
  this: SlowApi =>
  type Cache = Map[Int, Int]
  var cache: Cache = Map()
  def checkCache(i: Int) = State{ (s: Cache) => (s, s.get(i)) }
  def retrieve(i: Int) = State[Cache, Int]{ (s) =>
    val result = rpc(i)
    (s updated (i, result), result)
  }
  def cachedRequest(i: Int): State[Cache, Int] = for {
    ro <- checkCache(i)
    r  <- ro match {
      case Some(a) => State{ (s: Cache) => (s, a) }
      case None    => retrieve(i)
    }
  } yield r
  def receive = {
    //removed for berivity
  }
}
```

Naive StateMonad

Pros:

- Simpler parameter lists
- Functions compose nicely in for comprehension using flatMap
- Functions that return State are now referentially tranparent even if they call persist

Cons:

- State.run is not referentially transparent if we call persist
- Still pretty verbose

State Combinators

We can do better using combinators!

```
//State.state: lift a value into a State
def state[S, A](a: A) = State[S, A]{ s => (s, a) }

//State.get: gets the implicit state value out of a State Monad
//Called init in Scalaz. I don't know why
def get[S] = State[S, S]{ s => (s, s) }

//State.gets: Apply f to the state and return the result
//State.gets(identity) is the same as State.get
def gets[S, A](f: S => A) = State[S, A]{ s => (s, f(s)) }

//State.put: Ignore input state and replace with new S
def put[S](s: S) = State[S, Unit]{ _ => (s, ()) }

//State.modify: Mutate the state without producing a result
def modify[S](f: S => S) = State[S, Unit]{ s => (f(s), ()) }
```

State Combinators

Can we refactor this function?

```
def checkCache(i: Int) = State{ (s: Cache) => (s, s.get(i)) }
```

State Combinators

Using State.gets:

```
def checkCache(i: Int) = State{ (s: Cache) => (s, s.get(i)) }
```

Becomes...

```
def checkCache(i: Int) = State.gets{ (s: Cache) => s.get(i) }
```

We made the explicit return of state dissapear!

State Combinators

What about retrieve?

```
def retrieve(i: Int) = State[Cache, Int]{ (s) =>
  val result = rpc(i)
  (s updated (i, result), result)
}
```

State Combinators

We can use State.state and State.modify!

```
def retrieve(i: Int) = State[Cache, Int]{ (s) =>
  val result = rpc(i)
  (s updated (i, result), result)
}
```

Becomes..

```
def retrieve(i: Int): State[Cache, Int] = for {
  r <- State.state(rpc(i))
  _ <- State.modify[Cache]{ _ updated (i, r) }
} yield r
```

The s is gone!

State Combinators

Bringing it together

```
def checkCache(i: Int) = State.gets{ (s: Cache) => s.get(i) }
def retrieve(i: Int): State[Cache, Int] = for {
  r <- State.state(rpc(i))
  _ <- State.modify[Cache]{ _ updated (i, r) }
} yield r
def cachedRequest(i: Int): State[Cache, Int] = for {
  ro <- checkCache(i)
  r  <- ro.fold(retrieve(i))(State.state)
} yield r
```

We have removed explitic references to state to produce a highly readable program while eliminating a class of error

Table of Contents

Bringing it all together

Now that we know how to write a stateful program with pure functions by composing state combiantors, let's add event sourcing to make our program persistent and durable

- Actor becomes PersistentActor
- receive becomes receiveCommand
- need to implement receiveRecover
- figure out where to call persist...

Naive Persistent Solution

```scala
class PersistentCachingService extends PersistentActor {
  case class CacheUpdate(k: Int, v: Int)
  def persistenceId = "1"

  //...

  def retrieve(i: Int): State[Cache, Int] = for {
    r <- State.state(rpc(i))
    _ <- State.modify[Cache]{ (s: Cache) =>
      persist(CacheUpdate(i, r))( _ => () )
      s updated (i, r)
    }
  } yield r

  //...

  def receiveRecover = {
    case CacheUpdate(k, v) => cache = cache updated(k, v)
  }
```

Naive Persistent Solution

```
def retrieve(i: Int): State[Cache, Int] = for {
  r <- State.state(rpc(i))
  _ <- State.modify[Cache]{ (s: Cache) =>
    persist(CacheUpdate(i, r))( _ => () )
    s updated (i, r)
  }
} yield r
```

- This seems broken
- The state update doesn't happen anywhere near the call to persist
- We should return the Events that need to be persisted when we run our State Monad

Better Persistent Solution

Let's introduce some new types that will both contain the return value and can be used to journal changes to state

```
sealed trait Event
case class CacheUpdate(val k: Int, val v: Int) extends Event
case class CacheHit(val v: Int) extends Event
```

Better Persistent Solution

Refactor retrieve to return `State[Cache, Event]` instead of `State[Cache, Int]`

```
def retrieve(i: Int): JournalingState = for {
  cu <- State.state( CacheUpdate(i, rpc(i)) )
  _ <- State.modify[Cache]{ _ updated (i, cu.v) }
} yield cu
```

Better Persistent Solution

checkCache can be refactored to return `State[Cache, Option[Event]]` so no changes to cachedRequest are needed

```
def checkCache(i: Int) = State gets {
  (s: Cache) => s.get(i) map { CacheHit(_) }
}
```

Better Persistent Solution

The receiveCommand wrapping our functional core is refactored to persist changes to state as a precondition to actually applying the new state

```
def receiveCommand = {
  case i: Int => {
    //calculate new program state by running against current state
    val (newState, event) = cachedRequest(i)(cache)
    sender ! (event match {
      case CacheHit(v) => v
      //In the case where the state changed journal, then update state
      case cu@CacheUpdate(k, v) => persist(cu){ _ => cache = newState }; v
    })
  }
}
```

Better Persistent Solution

```
class JournalingCachingService(id: String) extends PersistentActor {
  this: SlowApi =>
  type Cache = Map[Int, Int]
  type JournalingState = State[Cache, Event]
  def persistenceId = id
  var cache: Cache = Map()
  def checkCache(i: Int) = State gets {
    (s: Cache) => s.get(i) map { CacheHit(_) }
  }
  def retrieve(i: Int): JournalingState = for {
    cu <- State.state( CacheUpdate(i, rpc(i)) )
    _ <- State.modify[Cache]{ _ updated (i, cu.v) }
  } yield cu
  def cachedRequest(i: Int): JournalingState = for {
    ro <- checkCache(i)
    r  <- ro.fold(retrieve(i))(State.state)
  } yield r
  def receiveCommand = {
    case i: Int => {
      val (newState, event) = cachedRequest(i)(cache)
      sender ! (event match {
        case CacheHit(v) => v
        case cu@CacheUpdate(k, v) => persist(cu){ _ => cache = newState }; v
      })
    }
  }
  def receiveRecover = {
    case CacheUpdate(k, v) => cache = cache updated(k, v)
  }
}
```

Summing Up

- We created a persistent and durable caching service using event sourcing and composed of pure functions!
- The State Monad replaced the side effect of mutating state with composable stateful computations

Further Improvements

- Overloading the return value to also carry journal information is clunky
- Should return `Writer[List[Event], Int]`
- Writer Monad solution produces much cleaner receiveCommand
- This talk was already way too long...

Questions?