Combining Concurrency Abstractions

Philipp Haller
Typesafe, Switzerland

Correctly and Efficiently

Combining Concurrency Abstractions

Philipp Haller
Typesafe, Switzerland

The Problem

- Tendency to combine several concurrency abstractions in a single project
  - Actors, futures, threads, latches, ...
- Source of hard-to-diagnose concurrency bugs
  - Non-blocking vs. blocking
  - Threads vs. thread pools
  - Closures and state

Typesafe

Actors + X

Actors, State & Futures

```
import akka.actor.Actor
import scala.concurrent.future

class MyActor extends Actor {
  // implicit ExecutionContext of context
  import context.dispatcher

  var state = 0

  def receive = {
    case Request(x) =>
      future {
        handleRequest(x, state)
      }
    case ChangeState(newState) =>
      state = newState
  }
}
```

Typesafe

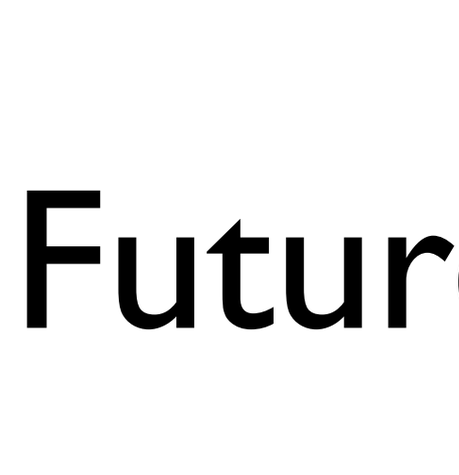
Actors, State & Futures

```
import akka.actor.Actor
import scala.concurrent.future

class MyActor extends Actor {
  // implicit ExecutionContext of context
  import context.dispatcher

  var state = 0

  def receive = {
    case Request(x) =>
      future {
        handleRequest(x, state)
      }
    case ChangeState(newState) =>
      state = newState
  }
}
```



Typesafe

Actors, State & Futures

```
import akka.actor.Actor
import scala.concurrent.future

class MyActor extends Actor {
  // implicit ExecutionContext of context
  import context.dispatcher

  var state = 0

  def receive = {
    case Request(x) =>
      future {
        handleRequest(x, state)
      }
    case ChangeState(newState) =>
      state = newState
  }
}
```

Typesafe

Safely Publishing State

```
import akka.actor.Actor
import scala.concurrent.future

class MyActor extends Actor {
  // implicit ExecutionContext of context
  import context.dispatcher

  var state = 0

  def receive = {
    case Request(x) =>
      val currentState = state
      future {
        handleRequest(x, currentState)
      }
    case ChangeState(newState) =>
      state = newState
  }
}
```

Typesafe

Actors, Futures & Senders

```
import akka.actor.Actor
import scala.concurrent.future

class MyActor extends Actor {
  // implicit ExecutionContext of context
  import context.dispatcher

  def receive = {
    case Request(x) =>
      future {
        val res = handleRequest(x)
        sender ! Response(res)
      }
  }
}
```

Typesafe

Actors, Futures & Senders

```
import akka.actor.Actor
import scala.concurrent.future

class MyActor extends Actor {
  // implicit ExecutionContext of context
  import context.dispatcher

  def receive = {
    case Request(x) =>
      future {
        val res = handleRequest(x)
        sender ! Response(res)
      }
  }
}
```

not constant!!

Typesafe

The Pipe Pattern

```
import akka.actor.Actor
import akka.pattern.pipe
import scala.concurrent.future
class MyActor extends Actor {
  // implicit ExecutionContext of context
  import context.dispatcher

  def receive = {
    case Request(x) =>
      future {
        val res = handleRequest(x)
        Response(res)
      } pipeTo sender
  }
}
```

Typesafe

The Pipe Pattern

```
import akka.actor.Actor
import akka.pattern.pipe
import scala.concurrent.future
class MyActor extends Actor {
  // implicit ExecutionContext of context
  import context.dispatcher

  def receive = {
    case Request(x) =>
      future {
        val res = handleRequest(x)
        Response(res)
      } pipeTo sender
  }
}
```

obtain sender once and store it

Typesafe

Actors + Threads

- How to exchange messages between an actor and a *regular (JVM) thread*?

Actors + Threads

- How to exchange messages between an actor and a *regular (JVM) thread*?
  - ask pattern (? operator): `val fut = actor ? msg`
  - `akka.actor.ActorDSL.Inbox` (Akka 2.1)

```
implicit val i = ActorDSL.inbox()
someActor ! someMsg // replies will go to `i`

val reply = i.receive()
val transformedReply = i.select(5.seconds) {
  case x: Int => 2 * x
}
```

Typesafe

A MapActor (not remote)

```
import akka.actor.Actor

class MapActor[K, V] extends Actor {
  var state = Map[K, V]()

  def receive = {
    case Put(k, v) =>
      state += (k -> v)
      sender ! AckPut
    case Get(k) =>
      sender ! state.get(k)
  }
}
```

Typesafe

A MapActor (not remote)

```
import akka.actor.Actor

class MapActor[K, V] extends Actor {
  var state = Map[K, V]()

  def receive = {
    case Put(k, v) =>
      state += (k -> v)
      sender ! AckPut
    case Get(k) =>
      sender ! state.get(k)
  }
}
```

Typesafe

Miscellaneous

- Thread locals
  - Scope: thread, *not* actor or future callback chain
- Shared-memory actors (same JVM)
  - Prefer sharing immutable data
  - Mutable data: Java Memory Model (`@volatile` etc.)

Typesafe

Combining Async and Blocking APIs

Blocking APIs

- java.lang.Object.wait
- java.io.Reader.read etc.
- java.util.concurrent: Future.get, CountDownLatch.await, BlockingQueue.put/take
- Scala 2.10 (SIP-14): Await.{result, ready}
- ...

Typesafe

Blocking Futures

```
import scala.concurrent._
import java.util.concurrent.{Future => JFuture}
import ExecutionContext.Implicits.global

object Main extends App {
  val futs: List[JFuture[String]] =
    // list of 4’000 Java futures

  val transformed = for (fut <- futs) yield
    future {
      fut.get(10, TimeUnit.SECONDS).toUpperCase
    }
}
```

Typesafe

Managed Blocking

```
import scala.concurrent._
import java.util.concurrent.{Future => JFuture}
import ExecutionContext.Implicits.global

object Main extends App {
  val futs: List[JFuture[String]] =
    // list of 4’000 Java futures

  val transformed = for (fut <- futs) yield
    future {
      blocking {
        fut.get(10, TimeUnit.SECONDS).toUpperCase
      }
    }
}
```

Fully Async

```
import scala.concurrent._

import ExecutionContext.Implicits.global

object Main extends App {
  val futs: List[Future[String]] =
    // list of 4’000 Scala futures

  val transformed = for (fut <- futs) yield
    fut.map(_.toUpperCase)
}
```

Preventing Misuse

Requiring Managed Blocking

```
trait Awaitable[+T] {
  def result(atMost: Duration)
            (implicit permit: CanAwait): T
}
package concurrent {
  @implicitNotFound("Use the `Await` object")
  sealed trait CanAwait

  private[concurrent] object AwaitPermission
    extends CanAwait

  object Await {
    def result[T](awaitable: Awaitable[T], ...): T =
      blocking(awaitable.result(atMost)(AwaitPermission))
  }
}
```

Typesafe

Your Turn!

- What do you find hard/confusing when combining concurrency abstractions?
- What practices do you follow/recommend to avoid concurrency hazards?

Typesafe

Thanks! Questions?

Philipp Haller
Typesafe, Switzerland