ActorRef[Typed]

ANDRZEJ KOPEĆ

>scalac

Why Scala?

Yet we have

```
PartialFunction[Any, Unit]
```

Ctrl+F based navigation

```
class BlockImporter(blockFetcher: ActorRef) extends Actor {
  blockFetcher ! ???
}
```

Mutable-by-default API

```scala
class BlockImporter extends Actor {
  var importedBlocks: List[Block] = Nil

  val importBlocks: List[Block] => Future[Unit]

  def receive = {
    case ImportBlocks(blocks) =>
      importBlocks(blocks)
        .onComplete(_ =>
          importedBlocks = importedBlocks ++ blocks
        )
  }
}
```

```scala
class BlockImporter extends Actor {
  var importedBlocks: List[Block] = Nil

  val importBlocks: List[Block] => Future[Unit]

  def receive = {
    case ImportBlocks(blocks) =>
      importBlocks(blocks)
        .onComplete(_ =>
          importedBlocks = importedBlocks ++ blocks
        )
  }
}
```

```
def receive = running(Nil)

def running(importedBlocks: List[Block]) = {
  case ImportBlocks(blocks) =>
    importBlocks(blocks)
      .onComplete(_ =>
        context become running(importedBlocks ++ blocks))
}
```

```scala
def receive = running(Nil)

def running(importedBlocks: List[Block]) = {
  case ImportBlocks(blocks) =>
    importBlocks(blocks)
      .onComplete(_ =>
        context become running(importedBlocks ++ blocks))
}
```

```scala
def receive = notImportingBlocks(Nil)

def notImportingBlocks(importedBlocks: List[Block]) = {
  case ImportBlocks(blocks) =>
    context become importingBlocks(importedBlocks)
    importBlocks(blocks)
      .map(BlocksImportCompleted(_))
      .pipeTo(self)
}

def importingBlocks(importedBlocks: List[Block]) = {
  case BlocksImportCompleted =>
    context become notImportingBlocks(importedBlocks ++ blocks)
}
```

```
def receive = notImportingBlocks(Nil)

def notImportingBlocks(importedBlocks: List[Block]) = {
  case ImportBlocks(blocks) =>
    context become importingBlocks(importedBlocks)
    importBlocks(blocks)
      .map(BlocksImportCompleted(_))
      .pipeTo(self)
}

def importingBlocks(importedBlocks: List[Block]) = {
  case BlocksImportCompleted =>
    context become notImportingBlocks(importedBlocks ++ blocks)
}
```

Future?

Say hello to `Behavior[T]`

```
package akka.actor.typed.scaladsl

object Behaviors {
  def receive[T](
    onMessage: (ActorContext[T], T) ⇒ Behavior[T]
  ): Behavior[T]
}
```

In practice it looks as follows:

Define your protocol

```scala
sealed trait BlockImporterMsg

case class ImportBlocks(
  blocks: List[Block],
  respondTo: ActorRef[BlocksImported]
  ) extends BlockImporterMsg
case class BlocksImported(blocks: List[Block])

case class BlocksImportCompleted(
  blocks: List[Block]) extends BlockImporterMsg
```

```scala
sealed trait BlockImporterMsg

case class ImportBlocks(
  blocks: List[Block],
  respondTo: ActorRef[BlocksImported]
  ) extends BlockImporterMsg
case class BlocksImported(blocks: List[Block])

case class BlocksImportCompleted(
  blocks: List[Block]) extends BlockImporterMsg
```

Define the behavior

```scala
object BlockImporter {
  def behavior(importedBlocks: List[Block] = Nil) =
    Behaviors.receive((ctx, msg) =>
      msg match {
        case ImportBlocks(blocks, respondTo) =>
          importBlocks(blocks)
            .onComplete(_ => {
              respondTo ! BlocksImported(blocks)
              ctx.self ! BlocksImportCompleted(blocks)
            })
          Behaviors.same
        case BlocksImportCompleted(blocks) =>
          behavior(importedBlocks ++ blocks)
      }
}
```

```
object BlockImporter {
  def behavior(importedBlocks: List[Block] = Nil) =
    Behaviors.receive((ctx, msg) =>
      msg match {
        case ImportBlocks(blocks, respondTo) =>
          importBlocks(blocks)
            .onComplete(_ => {
              respondTo ! BlocksImported(blocks)
              ctx.self ! BlocksImportCompleted(blocks)
            })
          Behaviors.same
        case BlocksImportCompleted(blocks) =>
          behavior(importedBlocks ++ blocks)
      }
}
```

Now the guardian...

```scala
object Guardian {
  case object Start

  def behavior = Behaviors.setup[Start.type]{ ctx =>
    val importer: ActorRef[BlockImporterMsg] =
      ctx.spawn(BlockImporter.behavior(), "block-importer")

    Behaviors.receiveMessage {
      case Start =>
        importer ! ImportBlocks(blocks)
        Behaviors.same
    }
  }
}
```

```scala
object Guardian {
  def behavior = SpawnProtocol.behavior
}
```

And the system

```scala
import akka.actor.typed.ActorSystem

object Main extends App {
  val system = ActorSystem(Guardian.behavior, "guardian")

  system ! Guardian.Start
}
```

Wait! Nice "hello, world" but I need...

Logging?

```scala
Behaviors.setup { ctx =>
  ctx.log.debug("in setup")
  someBehavior
}

Behaviors.receive((ctx, msg) => {
  ctx.log.debug("msg: {}", msg)
  Behaviors.same
})
```

```
Behaviors.logMessages(otherBehavior)
```

```scala
def behavior: Behavior[SomeMsg] =
  Behaviors
    .intercept(
      LoggingInterceptor[SomeMsg]())(
      otherBehavior)
```

```scala
class LoggingInterceptor[M] extends BehaviorInterceptor[M, M] {

  def aroundReceive(
    ctx: ActorContext[M],
    msg: M,
    target: BehaviorInterceptor.ReceiveTarget[M]
  ): Behavior[M] = {
    ctx.asScala.log.debug("Message {}", msg)
    target(ctx, msg)
  }

  def aroundSignal(ctx, signal, target): Behavior[M] = ???
}
```

To handle actor lifecycle?

```
Behaviors
  .receive(/* ... */)
  .receiveSignal {
    case (ctx, PostStop) => //PreRestart, Terminated, ChildFailed
      //cleanup
      Behaviors.same
  }
```

Supervision?

```
Behaviors
  .supervise(otherBehavior)
  .onFailure(SupervisorStrategy.restart)
```

Managing time?

```
val receive = Behaviors.receiveMessage {
  case CheckPeers =>
    //doSth
    Behaviors.same
}

Behaviors.withTimers(timers => {
  timers.startPeriodicTimer(
    "check-peers",
    CheckPeers,
    10.seconds)

    receive
})
```

And there's always...

```scala
package akka.actors.typed

abstract class ExtensibleBehavior[T] extends Behavior[T] {

  def receive(ctx: TypedActorContext[T], msg: T): Behavior[T]

  def receiveSignal(ctx: TypedActorContext[T], msg: Signal): Behavior[T]
}
```

Thank you

Andrzej Kopeć
andrzej.kopec@scalac.io
@kapke_