JVM の並行性: Akka を使ってアクター・アプリケーションを作成する

基礎を踏まえてアクターの相互作用を使用するアプリケーションを作成する

Dennis Sosnoski
Principal Consultant
Sosnoski Software Solutions Inc.

2015年 10月 15日

アクター・アプリケーションには、シングルスレッド・アプリケーションで用いられる線形アプローチとは異なるスタイルのプログラミングが必要です。ツールキットおよびランタイムとして Akka を使用してシステムを構築する方法について、Scala コードのアクターとメッセージという観点でさらに深く探ってください。

このシリーズの他の記事を見る

「JVM の並行性: Akka を使って非同期に振舞う」で、アクター・モデルと Akka というフレームワーク兼ランタイムについて紹介しました。アクター・アプリケーションを構築するのは、従来の線形アプリケーションを構築するのとは異なります。線形アプリケーションを構築する場合に検討するのは、制御フローと、目標を達成するために必要な一連のステップです。一方、アクター・モデルを有効に利用するには、アプリケーションを、状態と振る舞いからなる個々の独立したバンドル (アクター) に分解し、これらのバンドルの間での相互作用 (メッセージ) をスクリプト化します。この 2 つのコンポーネント (アクターとメッセージ) が、アプリケーションの構成単位となります。

アクターとメッセージを適切に作成すると、ほとんどの処理が非同期で行われるシステムが完成します。非同期操作は、線形アプローチよりも理解するのが難しいとは言え、スケーラビリティーにおけるメリットがあります。極めて非同期性の高いプログラムは、システム・リソース (メモリーやプロセッサーなど) が増やされると、それをより有効に利用することで、特定のタスクをより迅速に完了することや、より多くのタスク・インスタンスを同時に処理することができます。Akka を使用すれば、リモート機能を用いて、分散されたアクターとともに動作させることで、このスケーラビリティーを複数のシステムに展開することも可能です。

この連載について

マルチコア・システムが至るところで使われるようになった今、これまで以上に幅広く並行プログラミングを適用しなければならなくなっています。しかし、並行処理を適切に実装するのは難しい場合があり、並行処理を利用するための新しいツールも必要になってきます。

このようなツールは、JVM ベースの多くの言語で開発されていますが、なかでも Scala は、並行処理の分野で特に積極的です。この連載では、Java 言語と Scala 言語での新しい並行プログラミング手法をいくつか取り上げて検討します。

今回の記事では、アクターとメッセージという観点で、システムの構築について詳しく見ていきます。記事で取り上げる 2 つのサンプル・アプリケーションのうち、最初に取り上げるサンプル・アプリケーションでは、アクターとメッセージが Akka ではどのように機能するかについての基本事項を紹介します。2 番目に取り上げる、より複雑なサンプル・アプリケーションでは、アクター・システムの構造を計画して可視化する具体的な例を紹介します。サンプル・アプリケーションはどちらも Scala コードを使用していますが、Java 開発者には簡単に理解できる内容です (参考として、この連載の前回の記事で比較対照している、Akka を使用した Scala プログラミングと Java プログラミングの例を参照してください)。

この記事のサンプル・コードはダウンロードすることができます。

`Star` の紹介

前回の記事の例では、以下のようにしました。

- アクター・システムを起動するメイン・アプリケーションによって直接作成されたアクターを使用しました。
- ただ 1 つのタイプのアクターを使用しました。
- アクター間での相互作用を必要最小限にしました。

今回の最初のサンプル・アプリケーションでは、前回より少し複雑な構造を使用して、構成要素を 1 つひとつ確認します。リスト 1 に、完全なアプリケーションを記載します。

リスト 1. `Star` の生成

```
import scala.concurrent.duration._
import scala.util.Random
import akka.actor._
import akka.util._
object Stars1 extends App {
  import Star._
  val starBaseLifetime = 5000 millis
  val starVariableLifetime = 2000 millis
  val starBaseSpawntime = 2000 millis
  val starVariableSpawntime = 1000 millis

  object Namer {
    case object GetName
    case class SetName(name: String)
    def props(names: Array[String]): Props = Props(new Namer(names))
  }
  class Namer(names: Array[String]) extends Actor {
    import context.dispatcher
    import Namer._

    context.setReceiveTimeout(starBaseSpawntime + starVariableSpawntime)

    def receive = {
      case GetName => {
        val name = ...
        sender ! SetName(name)
      }
      case ReceiveTimeout => {
        println("Namer receive timeout, shutting down system")
```

```
      system shutdown
    }
  }
}

object Star {
  case class Greet(peer: ActorRef)
  case object AskName
  case class TellName(name: String)
  case object Spawn
  case object IntroduceMe
  case object Die
  def props(greeting: String, gennum: Int, parent: String) = Props(new Star(greeting, gennum, parent))
}
class Star(greeting: String, gennum: Int, parent: String) extends Actor {
  import context.dispatcher
  var myName: String = ""
  var starsKnown = Map[String, ActorRef]()
  val random = Random
  val namer = context actorSelection namerPath
  namer ! Namer.GetName

  def scaledDuration(base: FiniteDuration, variable: FiniteDuration) =
    base + variable * random.nextInt(1000) / 1000

  val killtime = scaledDuration(starBaseLifetime, starVariableLifetime)
  val killer = scheduler.scheduleOnce(killtime, self, Die)
  val spawntime = scaledDuration(starBaseSpawntime, starVariableSpawntime)
  val spawner = scheduler.schedule(spawntime, 1 second, self, Spawn)
  if (gennum > 1) scheduler.scheduleOnce(1 second, context.parent, IntroduceMe)

  def receive = {
    case Namer.SetName(name) => {
      myName = name
      println(s"$name is the ${gennum}th generation child of $parent")
      context become named
    }
  }
  def named: Receive = {
    case Greet(peer) => peer ! AskName
    case AskName => sender ! TellName(myName)
    case TellName(name) => {
      println(s"$myName says: '$greeting, $name'")
      starsKnown += name -> sender
    }
    case Spawn => {
      println(s"$myName says: A star is born!")
      context.actorOf(props(greeting, gennum + 1, myName))
    }
    case IntroduceMe => starsKnown.foreach {
      case (name, ref) => ref ! Greet(sender)
    }
    case Die => {
      println(s"$myName says: 'I'd like to thank the Academy...'")
      context stop self
    }
  }
}

val namerPath = "/user/namer"
val system = ActorSystem("actor-demo-scala")
val scheduler = system.scheduler
system.actorOf(Namer.props(Array("Bob", "Alice", "Rock", "Paper", "Scissors",
  "North", "South", "East", "West", "Up", "Down")), "namer")
val star1 = system.actorOf(props("Howya doing", 1, "Nobody"))
val star2 = system.actorOf(props("Happy to meet you", 1, "Nobody"))
Thread sleep 500
```

```
  star1 ! Greet(star2)
  star2 ! Greet(star1)
}
```

このアプリケーションは、`Namer` と `Star` という 2 つのタイプのアクターを使用してアクター・システムを作成します。`Namer` アクターはシングルトンであり、実質的に「名前」の中央ディレクトリーです。`Star` アクターは、`Namer` から名前 (ハンドル・ネーム) を付けられた後、他の `Star` への挨拶メッセージを出力します。これは前回の記事での例と同じですが、今回は `Star` アクターは子 `Star` も生成し、それらの子を既知の`Star` に紹介します。`Star` アクターは最終的に死を迎える場合があります。

リスト 2 に、このアプリケーションを実行すると表示される出力の例を記載します。

リスト 2. アプリケーションの出力

```
Bob is the 1th generation child of Nobody
Alice is the 1th generation child of Nobody
Bob says: 'Howya doing, Alice'
Alice says: 'Happy to meet you, Bob'
Bob says: A star is born!
Rock is the 2th generation child of Bob
Alice says: A star is born!
Paper is the 2th generation child of Alice
Bob says: A star is born!
Scissors is the 2th generation child of Bob
Alice says: 'Happy to meet you, Rock'
Alice says: A star is born!
North is the 2th generation child of Alice
Bob says: 'Howya doing, Paper'
Rock says: 'Howya doing, Paper'
Bob says: A star is born!
South is the 2th generation child of Bob
Alice says: 'Happy to meet you, Scissors'
Paper says: 'Happy to meet you, Scissors'
Alice says: A star is born!
East is the 2th generation child of Alice
Bob says: 'Howya doing, North'
Rock says: 'Howya doing, North'
Scissors says: 'Howya doing, North'
Paper says: A star is born!
West is the 3th generation child of Paper
Rock says: A star is born!
Up is the 3th generation child of Rock
Bob says: A star is born!
Down is the 2th generation child of Bob
Alice says: 'Happy to meet you, South'
North says: 'Happy to meet you, South'
Paper says: 'Happy to meet you, South'
Scissors says: A star is born!
Bob-Bob is the 3th generation child of Scissors
Alice says: A star is born!
Bob-Alice is the 2th generation child of Alice
Scissors says: 'Howya doing, East'
Rock says: 'Howya doing, East'
Bob says: 'Howya doing, East'
South says: 'Howya doing, East'
North says: A star is born!
Bob-Rock is the 3th generation child of North
Paper says: A star is born!
Bob-Paper is the 3th generation child of Paper
Bob says: 'I'd like to thank the Academy...'
Scissors says: 'Howya doing, West'
South says: 'Howya doing, West'
```

```
Alice says: A star is born!
Bob-Scissors is the 2th generation child of Alice
North says: A star is born!
Bob-North is the 3th generation child of North
Paper says: A star is born!
Bob-South is the 3th generation child of Paper
Alice says: 'I'd like to thank the Academy...'
Namer receive timeout, shutting down system
```

Star の生成

実際の俳優とは異なり、Star アクターは公開された劇的な形で子を生成することはありません。代わりに、Spawn メッセージを受け取るたびに、そっと子 Star を生成します。この出来事で興奮した様子が唯一見られるのは、「A star is born!」というシンプルな誕生の発表だけです。これも同じく実際の俳優とは異なり、新しく親となった Star は、誇らしげに新しい子の名前を発表することすらできません。名前は、命名機関によって決定されます。Namer は新しく生成された子 Star に名前を付けた後、子の名前とその詳細を「Ted is the 2th generation child of Bob」という形の 1 行で出力します。

Star の死は、Die メッセージを受け取った Star によってトリガーされ、Star はこのメッセージへの応答として「I'd like to thank the Academy....」というメッセージを出力します。Star はその後、context stop self ステートメントを実行し、制御側 Akka アクターのコンテキストに、用がなくなったためシャットダウンするよう通知します。通知を受けたコンテキストは、すべてのクリーンアップ処理を行い、アクターをシステムから削除します。

役割の変更

実際の俳優は、さまざま役を演じます。Akka のアクターも、メッセージ・ハンドラーのメソッドを変更することで、異なる役割を担うことができます。この仕組みは Star アクターに見ることができ、Star アクターでは、デフォルトの receive メソッドが扱うのは SetName メッセージのみであり、他のすべてのメッセージは named メソッドによって処理されます。別の役割への引継ぎは、SetName メッセージの処理の中で、context become named ステートメントによって行われます。この役割変更には、Star は名前が付けられるまでは何もできないようにして、名前が付けられた後は決して名前を変更できないようにする、という意図があります。

単一の receive メソッドですべてのメッセージ処理に対処することは常に可能ですが、その場合、現在のアクターの状態を判定基準とした条件文で取り散らかったコードになってしまいがちです。異なる状態ごとに別個の receive メソッドを使用すれば、簡潔で直接的なコードが維持されます。一般に、アクターの状態に応じて別のメッセージを使用したほうが適切な場合には、その状態を表す新しい receive メソッドを使用することを常に優先してください。

アクターの役割を変更するときは、有効なメッセージの処理を除外しないように注意する必要があります。例えば、Star アクターに随時名前の変更が許可されているとしたら、リスト 1 の named メソッドが SetName メッセージを処理する必要があります。アクターの現在の receive メソッドで処理されないメッセージは、いずれも事実上、破棄されます (実際には、デフォルトでデッドレター (配信不能) メールボックスに送信されますが、ユーザーのアクターに関するものであれば、破棄されることになります)。

メッセージ・ハンドラーに変更を加えるのに代わる方法として、現在のメッセージ・ハンドラーをスタックにプッシュして、2 つの引数を指定した形式 become(named, false) を使用して、

新しいメッセージ・ハンドラーを設定することもできます。こうすれば、最終的には `context unbecome` という呼び出しによって元のハンドラーを復元することができます。この方法では、必要なだけ幾重にも `become/unbecome` の呼び出しをネストすることできますが、最終的にコードがそれぞれの `become` に対応する `unbecome` を実行するように注意しなければなりません。対応していない `become` は、メモリー・リークを意味します。

`Namer` アクター

`Namer` アクターには、名前ストリングの配列がコンストラクターに格納されて渡されます。`Namer` アクターは `GetName` メッセージを受け取るたびに、配列に含まれる次の名前を `SetName` メッセージで返します。単純な名前を使い果たすと、今度はハイフンでつなげた名前を返します。`Namer` アクターの核心は、`Star` アクターに名前 (理想的には一意の名前) を付けることです。したがって、このシステムに `Namer` アクターのインスタンスが複数必要になる理由はありません。アクター・システムを起動するアプリケーション・コードは、このシングルトン・インスタンスを直接作成して、すべての `Star` が使用できるようにします。

アプリケーションはシングルトン `Namer` アクターを作成することから、このアクターの `ActorRef` を各 `Star` に渡せます。`Star` アクターがそれを子に渡すこともできます。ただし、Akka には、このタイプの有名なアクターをより簡潔に処理する方法があります。`Star` アクターの初期化コードに含まれる行 `val namer = context actorSelection namerPath` は、`Namer` アクターをアクター・システム内でのそのパス (この例では `/user/namer`) で検索します (`/user` プレフィックスは、ユーザーが作成するすべてのアクターに適用されます。`namer` は、`system.actorOf` を使用して `Namer` アクターが作成されたときに設定された名前です)。`namer` の値は、アプリケーションに含まれるすべてのアクターから見えるため、この値は必要に応じて直接使用することができます。

スケジューリング済みメッセージ

The リスト 1 のサンプル・アプリケーションは、各種のアクターに処理を促すために、複数のスケジューリング済みメッセージを使用します。`Star` アクターは初期化中に、2 つか 3 つのスケジューリング済みメッセージを作成します。`val killer = scheduler.scheduleOnce(killtime, self, Die)` ステートメントが作成するのは、ワンタイム・メッセージ・スケジューラーです。このスケジューラーは、`Star` に対し、舞台でのその活躍が終わった時点で `Die` メッセージを送信し、`Star` の死をトリガーします。`val spawner = scheduler.schedule(spawntime, 1 second, self, Spawn)` ステートメントは、反復型のスケジューラーを作成します。このスケジューラーは、新しく `Star` を生成するための初期遅延の後、1 秒間隔で `Spawn` メッセージを送信します。

`Star` に対する 3 つ目のタイプのスケジューリング済みメッセージが使用されるのは、`Star` が別の `Star` の子である場合のみです (アクター・システム外部のアプリケーション・コードによって作成された場合は使用されません)。`if (gennum > 1) scheduler.scheduleOnce(1 second, context.parent, IntroduceMe)` ステートメントは、新しい `Star` が 2 世代目以降の場合、`Star` が初期化された 1 秒後にその `Star` の親に送信するためのスケジューリング済みメッセージを作成します。親 `Star` はこのメッセージを受け取ると、すでに紹介されている他の `Star` のそれぞれに対し、この親 `Star` の子に自己紹介するようお願いする `Greet` メッセージを送信します。

`Namer` アクターも、受信タイムアウトという形でスケジューリング済みメッセージを使用します。`context.setReceiveTimeout(starBaseSpawntime + starVariab# leSpawntime)` ステートメントは、タイムアウトの値を子 `Star` の最大生成時間に設定します。このタイムアウトは、アク

ターがメッセージを受け取るたびに、コンテキストによってリセットされます。したがって、メッセージをまったく受信せずに指定された時間が経過した場合にのみ、タイムアウトが発生します。`Star` は継続的に新しい子 `Star` を生成し、それらの子によってメッセージが `Namer` に送信されるため、タイムアウトが発生するのは、すべての `Star` アクターがなくなった場合のみです。タイムアウトが発生すると、`Namer` はこのイベントによる (`akka.actor` パッケージの中で定義されている) `ReceiveTimeout` メッセージを処理する方法として、アクター・システム全体をシャットダウンします。

洞察力の鋭い読者は、`Namer` のタイムアウトは一体どのようにして起こるのだろうと不思議に思っていることでしょう。`Star` の存続期間は常に、少なくとも 5 秒に設定されます。各 `Star` は、最大 3 秒が経過するまでに子 `Star` の生成を開始します。このことから、`Star` は増え続けて膨大な数になるように思えます (リアリティー TV でスターが増え続けるようなものです)。それなのに、どのようにして `Namer` のタイムアウトは起こるのでしょう？その答えは、Akka のアクター監視モデルと親子関係にあります。

アクターの家族

Akka がアクターに適用する監視階層は、親子関係に基づきます。あるアクターが別のアクターを生成すると、生成されたアクターは、それを生成したアクターの従属アクターになります。つまり、親アクターは、子アクターに責任を持つということです (これは、実際の俳優にも適用されていて欲しいと思うことがよくある原則です)。この責任は、主に障害への対処に関するものですが、アクターがどのように機能するかに、何らかの影響を及ぼしているのは確かです。

リスト 1 のアクター・システムがシャットダウンする理由は、この監視階層にあります。この階層では、親アクターが有効であることが要求されるため、親アクターを終了すると、そのすべての子アクターも自動的に終了されます。リスト 1 で、アプリケーションが最初に作成する `Star` アクターは 2 つだけです (これらのアクターは、必ずそれぞれが `Bob`、`Alice` という名前を受け取ります)。その他すべての `Star` は、この 2 つの初期 `Star` アクターのいずれか、またはこれらのアクターの子 `Star` または孫 Star のいずれかによって作成されます。したがって、これらのルート `Star` のいずれかが終了すると、その子孫のすべてが一緒に終了されます。初期 `Star` アクターが両方とも終了した後は、`Star` は残っていません。いずれの `Star` も子 `Star` を生成することがなく、`Namer` に名前のリクエストが送られなければ、最終的に `Namer` のタイムアウトが起動されて、システムがシャットダウンするというわけです。

より複雑なアクター・システム

You saw in リスト 1 では、単純なアクター・システムがどのように機能するのか、その例を示しましたが、実際のアプリケーション・システムでは、一般にこれよりも多くのタイプのアクター (通常は、数十から数百になります) が使用され、アクター間でより複雑な相互作用が行われます。複雑なアクター・システムを設計して体系化する最善の方法の 1 つは、アクター間のメッセージ・フローを指定することです。

より複雑な例として、リスト 1 のアプリケーションを拡張し、映画製作の単純なモデルを実装しました。このモデルでは、以下に記載する 4 つの主要なアクター・タイプと 2 つの特化したアクター・サブタイプを使用します。

- `Star`: 映画に出演する俳優

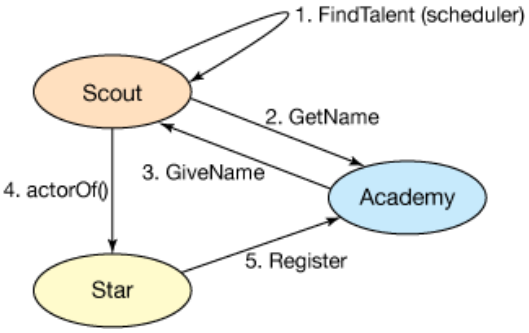
- `Scout`: 新しい `Star` を発掘する、タレントのスカウト
- `Academy`: アクティブに活動するすべての `Star` を追跡するシングルトン・レジストリー
- `Director`: 映画の製作者
  - `CastingAssistant`: 映画のキャスティングを担当する、`Director` のアシスタント
  - `ProductionAssistant`: 映画の製作を担当する、`Director` のアシスタント

リスト 1 の `Star` と同じように、このアプリケーションの `Star` アクターの生存期間は限られています。`Director` は映画製作に取り掛かる際に、映画にキャスティングできる、現在活動している `Star` のリストを取得します。`Director` はまず始めに、`Star` の出演契約を取り付ける必要があります。すべての `Star` の出演契約が済んだら、映画の製作に入ります。映画が完成する前に、出演している `Star` のいずれかが仕事を辞めた場合 (アクター用語で言うと、「死を迎えた」場合)、映画は失敗に終わります。

メッセージを図解する

リスト 1 のアプリケーションは単純であったため、アクターの相互作用を散文で説明することができましたが、この遥かに複雑な新しいアプリケーションには、その相互作用を表すための、より適切な方法が求められます。こうした相互作用を表す最適な方法となるのは、メッセージ・パッシング図です。図 1 に、`Scout` が新しい `Star` を見つけて (アクター用語で言うと、「`Star` を生成して」)、その新しい `Star` を `Academy` に登録する際に関係する相互作用のシーケンスを示します。

`Star` の作成と初期化



以下に記載するのが、`Star` を追加するために必要なメッセージのシーケンス (および作成ステップ) です。

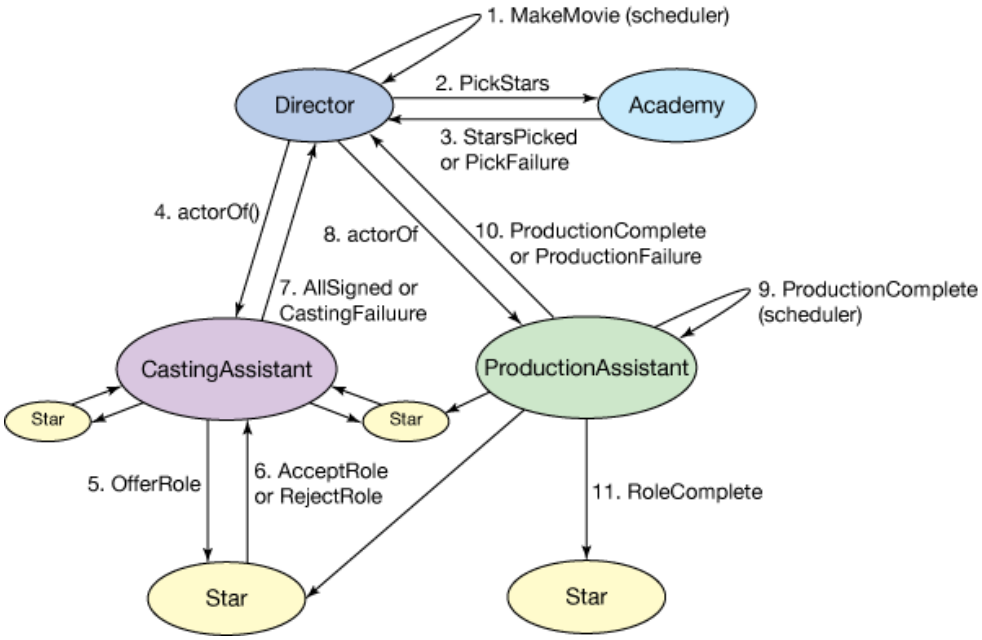
1. `FindTalent` (`Scheduler` から `Scout` へ): `Star` を追加するトリガーとなります。
2. `GetName` (`Scout` から `Academy` へ): `Star` に名前を付けます。
3. `GiveName` (`Academy` からの応答): 付けられた名前を提供します。
4. `actorOf()`: `Scout` が、提供された名前を設定した新しい `Star` アクターを作成します。
5. `Register` (`Star` から `Academy` へ): `Star` を `Academy` に登録します。

このメッセージ・シーケンスは、スケーラビリティーと柔軟性を持つように設計されています。各メッセージはそれぞれ別々に処理されるので、メッセージ交換を処理するためにアクターがその内部状態を変える必要はありません (`Academy` シングルトンは状態を変えますが、これはメッセージ交換の目的全体の一部です)。内部状態が変わらないという理由で、厳密なメッセージ・シーケンスを強制する必要はありません。例えば、複数の `GetName` メッセージを `Academy` に送信すれば、1 つの `FindTalent` メッセージで複数の `Star` を作成することができます。最後の `Star` の

作成を完了する前に、複数の `FindTalent` メッセージを続けて処理することもできます。また、システムには任意の数の `Scout` アクターを追加して、それらのアクターをそれぞれ別々に実行させたとしても、競合は起こりません。

映画の製作には、状態の変更や潜在的な障害の状況がより多く関わるため、映画の製作プロセスは、新しい `Star` を作成するプロセスより遥かに複雑になります。図 2 に、映画の製作に関わる主要なアプリケーション・メッセージを示します。

映画の製作



以下に記載するのが、映画の製作に関わるメッセージのシーケンスです。このシーケンスは、ほとんどのところ、すべてが問題なく運ぶ場合を想定しています。

1. `MakeMovie` (`Scheduler` から `Director` へ): 映画製作を開始するトリガーとなります。
2. `PickStars` (`Director` から `Academy` へ): 映画に出演させる Star を選びます。
3. `StarsPicked` または `PickFailure` (`Academy` からの応答): 映画を製作するのに十分なだけの `Star` がいる場合、`Academy` は必要な数を選択し、`StarsPicked` メッセージでリストを送り返します。そうでない場合、`Academy` は `PickFailure` メッセージを応答として送信します。
4. `actorOf()`: `Director` が、映画のキャスティングを担当する `CastingAssistant` アクターを作成します。
5. OfferRole (`CastingAssistant` から映画に出演する各 `Star` へ): `CastingAssistant` は、`Star` に役をオファーします。
6. `AcceptRole` または `RejectRole` (各 `Star` からの応答): `Star` は、すでに別の配役が決まっている場合、オファーされた役を辞退しますが、そうでなければオファーを受け入れます。
7. `AllSigned` または `CastingFailure` (`CastingAssistant` から親へ): すべての `Star` が役を受け入れた時点で、`CastingAssistant` の任務は完了します。したがって、`AllSigned` メッセージで親 `Director` に成功を報告します。`Star` をキャスティングできない場合 (特に、`Star` が死を迎えた場合)、`CastingAssistant` は親に失敗を報告します。いずれにしても、`CastingAssistant` の仕事は済んだので、このアクターを終了することができます。

8. `actorOf()`: `Director` が、映画の撮影を担当する `ProductionAssistant` アクターを作成します。
9. `ProductionComplete` (from `Scheduler` から `ProductionAssistant` へ): 所要時間が経過した後、映画の完成をトリガーします。
10. `ProductionComplete` または `ProductionFailure` (`ProductionAssistant` から親へ): 映画の完成用のタイマーが走っている場合、`ProductionAssistant` は映画が完成したことを親に報告します。
11. `RoleComplete` (`ProductionAssistant` から映画に出演する各 `Star` へ): `ProductionAssistant` は、各 `Star` にも映画の完成を通知し、別の映画に出演できるようにする必要があります。

このメッセージ・シーケンスは、処理の一環として、一部のアクターで状態変更を使用します。`Star` は、出演可能な状態と、映画に出演中の状態との間で、状態を変更する必要があります。`CastingAssistant` アクターは、どの `Star` と出演交渉をしなければならないかを把握するために、製作する映画の役をどの `Star` が引き受けてくれたかを追跡する必要があります。一方、`Director` アクターは、受信したメッセージ (子アクターからのメッセージを含む) に対して応答するだけなので、状態を変更する必要はありません。`ProductionAssistant` アクターも状態の変更は不要です。このアクターに必要なのは、映画が終了したときに、そのことを他のアクターに通知することだけだからです。

`CastingAssistant` アクターと `ProductionAssistant` アクターを別途使用しなくても済むように、この 2 つのアクターの役割を `Director` アクターにマージすることもできますが、これらのアクターを排除すると、`Director` が遥かに複雑になります。したがって、この例では一部の機能を他のアクターに振り分けておくのが妥当です。このことは、障害への対処について検討する場合に、特に言えることです。

障害に対処する

図 1 と図 2 のメッセージ・フローには、このアプリケーションの重要な側面の 1 つが含まれていません。`Star` の生存期間は限られているため、`Star` を扱うすべてのアクターは、`Star` が死を迎えるときを認識する必要があります。特に、映画に出演している `Star` が映画の完成前に死を迎えると、必然的に映画は失敗します。

Akka アクター・システムでの障害への対処は、親が監視するという形がとられ、障害の状況はアクターの階層の下から上へと渡されていきます。障害は、通常は JVM における例外として表されるため、Akka では障害の発生を検出するために通常の例外処理を用いています。アクターが自身のコードで例外を処理しない場合、Akka はキャッチされない例外を処理するために、アクターを終了し、親アクターに失敗を報告します。その後は、親が障害に対処する場合もあれば、親自身が失敗して、その親に失敗を報告する場合もあります。

Akka に組み込まれている障害対処の機能は、I/O 関連の障害といった状況には有効に機能しますが、映画製作システムの場合、例外が必要以上に複雑化することになります。このケースで必要なのは、他のアクターを監視することです。幸い、Akka にはこれを行うための簡単な方法があります。アクター・システムの `DeathWatch` コンポーネントを使用すると、アクターは、他のアクターを監視するアクターとして自身を登録することができます。登録された監視側アクターは、監視対象のアクターが死を迎えると、システム・メッセージ `Terminated` を受け取ります (レース・コンディションになるのを回避するため、監視対象アクターがすでに死を迎えている場合に

は、監視を開始すると即座に、監視側アクターのメールボックスに `Terminated` メッセージが現れます)。

`DeathWatch` をアクティブにするには、`context.watch()` メソッドを呼び出します。このメソッドは、監視対象とするアクターの `ActorRef` を引数に取ります。映画製作の例で必要な障害対処は、監視対象のアクターが死を迎えると生成される `Terminated` メッセージだけです。

`Star` を生成するコード

リスト 3 に、アプリケーションを起動して新しい `Star` を生成するために必要なコードを記載します。このコードは、図 1 に示したメッセージ・フローに対応します。

リスト 3. `Star` を生成するコード

```
object Stars2 extends App {  object Scout {
    case object FindTalent
    val starBaseLifetime = 7 seconds
    val starVariableLifetime = 3 seconds
    val findBaseTime = 1 seconds
    val findVariableTime = 3 seconds
    def props(): Props = Props(new Scout())
}
class Scout extends Actor {
    import Scout._
    import Academy._
    import context.dispatcher

    val random = Random
    scheduleFind

    def scheduleFind = {
      val nextTime = scaledDuration(findBaseTime, findVariableTime)
      scheduler.scheduleOnce(nextTime, self, FindTalent)
    }

    def scaledDuration(base: FiniteDuration, variable: FiniteDuration) =
      base + variable * random.nextInt(1000) / 1000

    def receive = {
      case FindTalent => academy ! GetName
      case GiveName(name) => {
        system.actorOf(Star.props(name, scaledDuration(starBaseLifetime, starVariableLifetime)), name)
        println(s"$name has been discovered")
        scheduleFind
      }
    }
  }

  object Academy {
    case object GetName
    case class GiveName(name: String)
    case class Register(name: String)
    ...
    def props(names: Array[String]): Props = Props(new Academy(names))
  }
class Academy(names: Array[String]) extends Actor {
    import Academy._

    var nextNameIndex = 0
    val nameIndexLimit = names.length * (names.length + 1)
    val liveStars = Buffer[(ActorRef, String)]()
    ...
    def receive = {
```

```
      case GetName => {
        val name =
          if (nextNameIndex < names.length) names(nextNameIndex)
          else {
            val first = nextNameIndex / names.length - 1
            val second = nextNameIndex % names.length
            names(first) + "-" + names(second)
          }
        sender ! GiveName(name)
        nextNameIndex = (nextNameIndex + 1) % nameIndexLimit
      }
      case Register(name) => {
        liveStars += ((sender, name))
        context.watch(sender)
        println(s"Academy now tracking ${liveStars.size} stars")
      }
     case Terminated(ref) => {
        val star = (liveStars.find(_._1 == ref)).get
        liveStars -= star
        println(s"${star._2} has left the business\nAcademy now tracking ${liveStars.size} Stars")
      }
      ...
      }
    }
  }

  object Star {
    ...
    def props(name: String, lifespan: FiniteDuration) = Props(new Star(name, lifespan))
  }
class Star(name: String, lifespan: FiniteDuration) extends Actor {
    import Star._
    import context.dispatcher

    academy ! Academy.Register(name)

    scheduler.scheduleOnce(lifespan, self, PoisonPill)
  }
  ...
  val system = ActorSystem("actor-demo-scala")
  val scheduler = system.scheduler
  val academy = system.actorOf(Academy.props(Array("Bob", "Alice", "Rock",
    "Paper", "Scissors", "North", "South", "East",  "West", "Up", "Down")), "Academy")
  system.actorOf(Scout.props(), "Sam")
  system.actorOf(Scout.props(), "Dean")
  system.actorOf(Director.props("Astro"), "Astro")
  system.actorOf(Director.props("Cosmo"), "Cosmo")
  Thread sleep 15000
  system.shutdown
}
```

The リスト 3 のコードは、その大部分でリスト 1 での `Star` の例と同じ Akka 機能を使用していますが、`DeathWatch` をアクティブにする `context.watch()` 呼び出しが追加されています。このメソッドは、`Academy` アクターが新しい `Star` からの `Register` メッセージを処理する際に呼び出します。`Academy` アクターは、各 `Star` の `ActorRef` と名前を両方とも記録しておき、`Terminated` メッセージを処理する際には、`ActorRef` を使用することで、死を迎えた Star を検出して削除します。こうして、アクティブな `Star` の `Buffer` (基本的には `ArrayList`) は最新の状態に保たれます。

メインのアプリケーション・コードは、最初にシングルトン `Academy` アクターを作成してから、一組の `Scout` を作成し、最後に一組の `Director` を作成します。このアプリケーションは、15 秒間アクター・システムの実行を許可し、その後システムをシャットダウンして終了します。

映画製作を開始する

リスト 4 に、映画製作に関係するコードの最初の部分として、映画に参加する `Star` のキャスティングを行うコードを記載します。このコードは、図 2 のメッセージ・フローの上の部分に対応しており、`Scheduler` が含まれるほか、`Director` と `Academy` アクターとの間での相互作用が含まれています。

リスト 4. 映画を製作するためのコード

```
object Stars2 extends App {
  ...
  object Director {
    case object MakeMovie

    val starCountBase = 2
    val starCountVariable = 4
    val productionTime = 3 seconds
    val recoveryTime = 3 seconds

    def props(name: String) = Props(new Director(name))
  }

  class Director(name: String) extends Actor {
    import Academy._
    import Director._
    import ProductionAssistant._
    import context.dispatcher

    val random = Random

    def makeMovie = {
      val numstars = random.nextInt(starCountVariable) + starCountBase
      academy ! PickStars(numstars)
    }
    def retryMovie = scheduler.scheduleOnce(recoveryTime, self, MakeMovie)
    makeMovie

    def receive = {
      case MakeMovie => makeMovie
      case PickFailure => retryMovie
      case StarsPicked(stars) => {
        println(s"$name wants to make a movie with ${stars.length} actors")
        context.actorOf(CastingAssistant.props(name, stars.map(_._1)), name + ":Casting")
        context become casting
      }
    }
    ...
  }
  ...
  object Academy {
    ...
    case class PickStars(count: Int)
    case object PickFailure
    case class StarsPicked(ref: List[(ActorRef, String)])

    def props(names: Array[String]): Props = Props(new Academy(names))
  }

  class Academy(names: Array[String]) extends Actor {
    ...
    def pickStars(n: Int): Seq[(ActorRef, String)] = ...

    def receive = {
      ...
      case PickStars(n) => {
```

```
        if (liveStars.size < n) sender ! PickFailure
        else sender ! StarsPicked(pickStars(n).toList)
      }
    }
  }
```

リスト 4 のコードの最初の部分では、`Director` オブジェクトと、アクターの一部を定義し、`Scheduler` が `Director` に `MakeMovie`メッセージを送信することによってトリガーされる映画製作の最初を示しています。`Director` はこの `MakeMovie` メッセージを受け取ると、映画製作のプロセスを開始し、`Academy` に対し、その映画に出演させる `Star` を選ぶよう、`PickStars` メッセージによって要求します。リスト 4 の最後に示されている、`PickStars` メッセージを扱う `Academy` のコードは、`PickFailure` メッセージ (十分な数の `Star` がいない場合) または `StarsPicked` メッセージのいずれかを送信します。`Director` は `PickFailure` メッセージを受け取ると、映画製作を後ほどもう一度試みるようスケジューリングします。`Director` が `StarsPicked` メッセージを受け取った場合は、`CastingAssistant` アクターを起動し、`Academy` によって映画の役に選ばれた `Star` のリストをこのアクターに渡した後、`CastingAssistant` からの応答に対処するための状態に変わります。この時点から引き継いでいるリスト 5 は、`Director` アクターが `Receive` メソッドをキャスティングするところから始まります。

リスト 5. `CastingAssistant` の動作

```
  class Director(name: String) extends Actor {
    ...
    def casting: Receive = {
      case CastingAssistant.AllSigned(stars) => {
        println(s"$name cast ${stars.length} actors for movie, starting production")
        context.actorOf(ProductionAssistant.props(productionTime, stars), name + ":Production")
        context become making
      }
      case CastingAssistant.CastingFailure => {
        println(s"$name failed casting a movie")
        retryMovie
        context become receive
      }
    }
    ...
  }

object CastingAssistant {
    case class AllSigned(stars: List[ActorRef])
    case object CastingFailure

    val retryTime = 1 second

    def props(dirname: String, stars: List[ActorRef]) = Props(new CastingAssistant(dirname, stars))
  }

  class CastingAssistant(dirname: String, stars: List[ActorRef]) extends Actor {
    import CastingAssistant._
    import Star._
    import context.dispatcher

    var signed = Set[ActorRef]()
    stars.foreach { star =>
      {
        star ! OfferRole
        context.watch(star)
      }
    }
```

```
    def receive = {
      case AcceptRole => {
        signed += sender
        println(s"Signed star ${signed.size} of ${stars.size} for director $dirname")
        if (signed.size == stars.size) {
          context.parent ! AllSigned(stars)
          context.stop(self)
        }
      }
      case RejectRole => scheduler.scheduleOnce(retryTime, sender, OfferRole)
      case Terminated(ref) => {
        context.parent ! CastingFailure
        stars.foreach { _ ! Star.CancelOffer }
        context.stop(self)
      }
    }
  }

  object Star {
    case object OfferRole
    case object AcceptRole
    case object RejectRole
    case object CancelOffer
    case object RoleComplete
    ...
  }

  class Star(name: String, lifespan: FiniteDuration) extends Actor {
    ...
    var acceptedOffer: ActorRef = null

    scheduler.scheduleOnce(lifespan, self, PoisonPill)

    def receive = {
      case OfferRole => {
        sender ! AcceptRole
        acceptedOffer = sender
        context become booked
      }
    }

    def booked: Receive = {
      case OfferRole => sender ! RejectRole
      case CancelOffer => if (sender == acceptedOffer) context become receive
      case RoleComplete => context become receive
    }
  }
```

Director は、映画にキャスティングする Star の ActorRef のリストを使用して、CastingAssistant を作成します。CastingAssistant はまず、リストに含まれる Star のそれぞれに OfferRole を送信し、自身を監視者として各 Star に登録します。その後、CastingAssistant は各 Star から返される AcceptRole メッセージまたは RejectRole メッセージ、あるいはいずれかの Star が死を迎えたことを報告する、アクター・システムからの Terminated メッセージを待機します。

CastingAssistant がその映画のキャストのすべての Star から AcceptRole を受け取ると、AllSigned メッセージを親 Director に返します。このメッセージには、便宜上、Star の actorRef のリストが格納されます。というのも、このリストを次の処理ステップに渡す必要があるためです。

CastingAssistant がいずれかの Star から RejectRole メッセージを受け取った場合は、少し時間を置いてから同じアクターに OfferRole を再送信するようスケジューリングします (Star と連絡が取れないことはよくあるため、ある Star に映画に出演してもらいたい場合は、Star が出演を承諾するまで依頼し続ける必要があります)。

CastingAssistant が Terminated メッセージを受け取った場合、それは、その映画のキャストに選んだ Star のいずれかが死を迎えたことを意味します。この悲しむべき事態が発生すると、CastingAssistant は親 Director に CastingFailure を報告し、自身を終了します。ただし終了する前に、リストに含まれる各 Star に CancelOffer メッセージを送信し、その映画での役が決まっていた Star が別の役を引き受けられるよう解放します。

CastingAssistant が、AcceptRole メッセージを処理していない Star も含め、すべての Star に CancelOffer メッセージを送信するのは不思議に思うかもしれませんが、そうする理由は、CastingAssistant が Terminated メッセージに対処する時点で、リストに含まれる Star によって送信された AcceptRole メッセージが、メールボックスにまだ存在する可能性が考えられるからです。さらに、分散アクター・システムの通常のケースでは、Star が出演を承諾したとしても、その AcceptRole メッセージが送信中であったり、紛失されたりする可能性もあります。いずれの場合も、CancelOffer メッセージを各 Star に送信すれば、障害対処が簡潔になります。製作予定の映画の役を承諾していない Star は、CancelOffer メッセージを無視すればよいのです。

リスト 6 に、映画製作プロセスの最後の部分を記載します。これは、ProductionAssistant アクターの動作です (図 2 の右下の部分に対応します)。ProductionAssistant に必要なのは、SchedulerProductionComplete メッセージ、または Terminated メッセージに対処することだけなので、単純なコードになっています。

リスト 6. ProductionAssistant の動作

```
class Director(name: String) extends Actor {
  ...
  def making: Receive = {
    case m: ProductionAssistant.ProductionEnd => {
      m match {
        case ProductionComplete => println(s"$name made a movie!")
        case ProductionFailed => println(s"$name failed making a movie")
      }
      makeMovie
      context become receive
    }
  }
}

object ProductionAssistant {
  sealed trait ProductionEnd
  case object ProductionComplete extends ProductionEnd
  case object ProductionFailed extends ProductionEnd

  def props(time: FiniteDuration, stars: List[ActorRef]) = Props(new ProductionAssistant(time, stars))
}

class ProductionAssistant(time: FiniteDuration, stars: List[ActorRef]) extends Actor {
  import ProductionAssistant._
  import context.dispatcher

  stars.foreach { star => context.watch(star) }
  scheduler.scheduleOnce(time, self, ProductionComplete)
```

```
    def endProduction(end: ProductionEnd) = {
      context.parent ! end
      stars.foreach { star => star ! Star.RoleComplete }
      context.stop(self)
    }

    def receive = {
      case ProductionComplete => endProduction(ProductionComplete)
      case Terminated(ref) => endProduction(ProductionFailed)
    }
  }
```

`ProductionAssistant` は、`Scheduler` から `ProductionComplete` メッセージを受け取った場合には、成功したことを親 `Director` に報告することができます。`Terminated` メッセージを最初に受け取った場合には、失敗を報告しなければなりません。いずれにしても、その映画に関わっているすべての `Star` に仕事が終わったことを通知して、クリーンアップします。

リスト 7 は、このプログラムを実行すると表示される出力の例です、映画製作の結果は、太字で示されています。

リスト 7. 出力例

```
Bob has been discovered
Academy now tracking 1 stars
Alice has been discovered
Academy now tracking 2 stars
Rock has been discovered
Academy now tracking 3 stars
Paper has been discovered
Academy now tracking 4 stars
Cosmo wants to make a movie with 4 actors
Astro wants to make a movie with 3 actors
Signed star 1 of 4 for director Cosmo
Signed star 2 of 4 for director Cosmo
Signed star 3 of 4 for director Cosmo
Signed star 4 of 4 for director Cosmo
Cosmo cast 4 actors for movie, starting production
Scissors has been discovered
Academy now tracking 5 stars
Cosmo made a movie!
Cosmo wants to make a movie with 4 actors
Signed star 1 of 4 for director Cosmo
Signed star 2 of 4 for director Cosmo
Signed star 3 of 4 for director Cosmo
Signed star 4 of 4 for director Cosmo
Cosmo cast 4 actors for movie, starting production
North has been discovered
Academy now tracking 6 stars
South has been discovered
Academy now tracking 7 stars
Cosmo failed making a movieAstro failed casting a movie
Bob has left the business
Academy now tracking 6 Stars
Cosmo wants to make a movie with 3 actors
Signed star 1 of 3 for director Cosmo
Signed star 2 of 3 for director Cosmo
Signed star 3 of 3 for director Cosmo
Cosmo cast 3 actors for movie, starting production
East has been discovered
Academy now tracking 7 stars
West has been discovered
Academy now tracking 8 stars
Alice has left the business
```

```
Academy now tracking 7 Stars
Rock has left the business
Academy now tracking 6 Stars
Up has been discovered
Academy now tracking 7 stars
Astro wants to make a movie with 2 actors
Signed star 1 of 2 for director Astro
Signed star 2 of 2 for director Astro
Astro cast 2 actors for movie, starting production
Cosmo made a movie!
Cosmo wants to make a movie with 3 actors
Signed star 1 of 3 for director Cosmo
Signed star 2 of 3 for director Cosmo
Signed star 3 of 3 for director Cosmo
Cosmo cast 3 actors for movie, starting production
Down has been discovered
Academy now tracking 8 stars
```

リストの中ほどにある二重の失敗は、興味深い出力シーケンスを示しています。最初に行「`Cosmo failed making a movie`」があり、次に「`Astro failed casting a movie`」が続き、その後に「`Bob has left the business`」となっています。これらの行は、`Bob` という `Star` が死を迎えたことによる相互作用を示します。このケースでは、`Bob` は `Cosmo` が製作する映画の役を承諾し、映画製作はすでに始まっているので、`Cosmo` の `ProductionAssistant` は `Terminated` メッセージを受け取ると、映画の製作が失敗します。`Bob` は、`Astro` が製作する映画での役にも選ばれていますが、その役をまだ承諾していません (`Bob` はすでに `Cosmo` の映画に出演する契約になっていたためです)。そのため、`Astro` の `CastingAssistant` が `Terminated` メッセージを受け取ると、映画のキャスティングが失敗します。3 番目のメッセージは、`Terminated` メッセージを受信した `Academy` によって生成されたものです。

まとめ

実際のアクター・システム・アプリケーションには、複数の (通常は多数の) アクターと、アクター間のメッセージが必要になります。この記事では、アクター・システムを構築する方法と、システムの操作を理解するための支援としてアクター間の相互作用を図にする方法を説明しました。アクターとメッセージを扱うのは、逐次コードを作成するのとは異なるプログラミング手法です。ある程度の経験を積んでくると、アクター手法では、スケーラビリティーが高い非同期実行のプログラムを容易に作成できることがわかってきます。

アクターとメッセージ交換を構築すると、アクター・システムが機能するようにだけなります。ある時点で、アクターが誤った動作をしている箇所を突き止める必要も出てくるでしょう。しかし、アクター・システムの非同期という性質が、問題のある相互作用の特定をより困難にします。アクターの相互作用を追跡してデバッグする方法は、それだけで 1 つの記事に値するトピックです。

著者について

Dennis Sosnoski

Dennis Sosnoski は、スケーラブルなシステムの開発経験が豊富にある、Java および Scala の開発者です。XML と Web サービスの分野で有名な彼のバックグラウンドとしては、JiBX XML データ・バインディングの開発や、いくつかのオープンソース Web サービス・フレームワーク (一番最近のものでは Apache CXF) に関する取り組みなどがあります。Dennis は Java ユーザー・グループや Java カンファレンスで頻繁にプレゼンターを務めており、人気のある連載「Java Web サービス」をはじめとし、developerWorks の数多くの記事を執筆しています。彼が行っている Web サービスのトレーニングと、コンサルティング作業について Sosnoski Software Associates Ltd サイトで詳しい情報を得てください。また、彼が現在行っている JVM に関する並行プログラミングの探求を Scalable Scala サイトでチェックして読んでください。