JVM の並行性: Akka を使って非同期に振舞う

並行処理アプリケーションのためのアクター・システムを構築する

Dennis Sosnoski
Principal Consultant
Sosnoski Software Solutions Inc.

2015年 10月 22日

アクター・モデルはこれまで長い間、並行処理プログラムを分析する際の理論基盤としても、並行処理プログラムを実装するための実用的手法としても使われてきました。メッセージを介して相互作用する、単純なアクター・エンティティーの構造をベースとするアクター・モデルは、並行性とスケーラビリティーに優れたアプリケーションを簡単に作成する手段となります。この記事でアクター・モデルについて学び、Akka で実装したモデルを Scala または Java で扱うようになってください。

このシリーズの他の記事を見る

この連載のこれまでの記事では、以下のいずれかの方法による並行性の実装を検討してきました。

- 複数のデータ・セットに対して同じ処理を並行して実行する方法 (Java 8 ストリームを使用した場合)
- 特定の処理が非同期で実行されるよう、明示的に複数の計算処理を構成し、それらの結果を結合する方法 (future を使用した場合)

どちらも並行性を実現する素晴らしい方法ですが、いずれも明示的に設計してアプリケーションに組み込まなければなりません。

今回の記事から何回かにわたり、並行性に対する、これらとは異なる方法に焦点を当てます。その方法は、明示的なコーディングをベースとしているのではなく、特定のプログラム構造をベースとしており、そのプログラム構造が「アクター・モデル」です。この数回の記事では、特にアクター・モデルの Akka 実装を扱う方法を取り上げたいと思います (Akka は並行分散型 JVM アプリケーションを作成するためのツールキット兼ランタイムです)。この記事で使用する完全なサンプル・コードへのリンクについては、「参考文献」を参照してください。

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。

アクター・モデルの基礎

並行計算のアクター・モデルでは、「アクター」と呼ばれるプリミティブをベースにシステムを構築します。アクターは、「メッセージ」と呼ばれる入力に応答してアクションを実行します。実行されるアクションには、アクター自体の内部状態の変更、他のメッセージの送信、さらには他のアクターの作成などがあります。すべてのメッセージは非同期で配信されるため、メッセージの送信側と受信側は切り離されます。このように分離されているおかげで、アクター・システムには本質的に並行性が備わっており、入力メッセージを受け取っているアクターは、すべて制限なく並行して実行することが可能です。

Akka におけるアクターは、メッセージを介して相互作用する振る舞いの (いくらか神経質な) バンドルであるという印象を受けます。実際の俳優のように、Akka のアクターはある程度のプライバシーを必要とするため、Akka のアクターに直接メッセージを送信することはできません。メッセージは、私書箱に相当するアクター参照に送信します。受信されるメッセージは、その参照を介してアクターのメールボックスにルーティングされ、後でアクターに配信されます。Akka のアクターは、他のアクターから汚染されるのを避けるために、すべての受信メッセージが無菌 (JVM の用語では、「不変」) であることも要求します。

Akka の基本を超えた機能

Akka が実装するアクター・モデルには、柔軟性を高めるいくつかの工夫が加えられています。そのような工夫の 1 つは、完全な分散型アーキテクチャーになっており、アクター・システムをネットワーク内の複数のノードに分散できることです。また、Akka はアクターを階層的に配置します。この階層的な配置では、各アクター (システムのルート・スーパーバイザーを除く) に親アクターがあり、その親アクターには子アクターの障害に対処する責任があります。これらはどちらも Akka の重要な機能ですが、多少高度なトピックになるため、ここでは触れておくだけにします。

実際の俳優の要求とは異なり、Akka においてアクターによるこのような一見過剰なまでの制約が存在しているのには、理由があります。アクターを対象とした参照を使用すると、メッセージ交換以外の手段で相互作用が行われるのを防ぐことができます (メッセージ交換以外の手段で相互作用が行われると、アクター・モデルの中核となっている、メッセージの送信側と受信側の切り離しが損なわれる恐れがあります)。アクターはシングルスレッドで実行されるため (ある特定のアクター・インスタンスが、複数のスレッドによって実行されることはありません)、メールボックスは、メッセージを処理可能になるまで保持するバッファーの役割を果たします。また、メッセージの不変性 (現時点では JVM の制限があるため、Akka が要件として適用していませんが、メッセージの不変性は規定された要件です) は、アクター間で共有されるデータに影響する同期の問題を懸念しなくても済むことを意味します。唯一共有されるデータが不変であれば、同期が必要になることはありません。

実際のコード

アクター・モデルと Akka の特質についての概要を理解したところで、実際のコードを見ていきましょう。コーディング・サンプルとして Hello を使用するのはありきたりですが、これによって、言語やシステムのスナップショットを迅速かつ容易に理解することができます。リスト 1 に、Scala で作成した Akka を記載します。

リスト 1. Scala での単純な Hello

```
import akka.actor._
import akka.util._

/** Simple hello from an actor in Scala. */
object Hello1 extends App {

  val system = ActorSystem("actor-demo-scala")
  val hello = system.actorOf(Props[Hello])
  hello ! "Bob"
  Thread sleep 1000
  system shutdown

  class Hello extends Actor {
    def receive = {
      case name: String => println(s"Hello $name")
    }
  }
}
```

リスト 1 のコードを構成する 2 つの個別のチャンクは、どちらも `Hello1` アプリケーション・オブジェクトに含まれています。最初のコード・チャンクは、以下の処理を行う Akka アプリケーション・インフラストラクチャーです。

1. アクター・システムを作成します (`ActorSystem(...)` の行)。
2. アクター・システム内にアクターを作成します (`system.actorOf(...)` の行。このコード行は、作成したアクターのアクター参照を返します)。
3. アクター参照を使用して、アクターにメッセージを送信します (`hello ! "Bob"` の行)。
4. 1 秒待機してから、アクター・システムをシャットダウンします (`system shutdown` の行)。

`system.actorOf(Props[Hello])` 呼び出しは、アクター・インスタンスを作成する際に推奨される方法であり、`Hello` アクター・タイプに特化した構成プロパティーを使用しています。この単純なアクター (台詞が 1 行しかない端役を演じる俳優) の場合、構成情報は何もないので、`Props` オブジェクトにはパラメーターがありません。アクターに構成情報を設定する必要がある場合、そのアクターに固有の `Props` クラスを定義して、そこに必要なすべての情報を含めることができます (後で記載する例に、その方法を示します)。

`hello ! "Bob"` ステートメントは、作成されたアクターにメッセージ (この例では、`Bob` というストリングだけ) を送信します。`!` 演算子は、Akka におけるアクターへのメッセージ送信を表す便利な手段です (このパターンでは、メッセージを送信した後はお任せです)。特化された演算子スタイルが好みでなければ、代わりに `tell()` を使用して同じことを実行できます。

`class Hello extends Actor` で始まる 2 番目のコード・チャンクは、`Hello` アクター定義です。この特定のアクター定義は最大限単純なものになっていて、(すべてのアクターに) 必要な部分関数 `receive` を定義しています。この部分関数が、入力メッセージの処理を実装します。(`receive` が部分関数になっているのは、一部の入力 (この例では、`String` メッセージ入力のみ) に対してだけ定義されているためです)。このアクターに実装された処理は、`String` メッセージを受信するたびに、メッセージの値を使用して挨拶を出力します。

Java での Hello

リスト 1 の Akka による Hello を通常の Java で作成すると、リスト 2 のようになります。

リスト 2. Java での Hello

```
import akka.actor.*;

public class Hello1 {

    public static void main(String[] args) {
        ActorSystem system = ActorSystem.create("actor-demo-java");
        ActorRef hello = system.actorOf(Props.create(Hello.class));
        hello.tell("Bob", ActorRef.noSender());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) { /* ignore */ }
        system.shutdown();
    }

    private static class Hello extends UntypedActor {

        public void onReceive(Object message) throws Exception {
            if (message instanceof String) {
                System.out.println("Hello " + message);
            }
        }
    }
}
```

リスト 3 に、Java 8 でのラムダ式を使用したアクター定義と、ラムダ式をサポートする `ReceiveBuilder` クラスに必要なインポートを示します。リスト 3 は、多少簡潔になっているかもしれませんが、その点以外はリスト 2 とほとんど同じです。

リスト 3. Java 8 での Akka による Hello

```
import akka.japi.pf.ReceiveBuilder;
...
    private static class Hello extends AbstractActor {

        public Hello() {
            receive(ReceiveBuilder.
                match(String.class, s -> { System.out.println("Hello " + s); }).
                build());
        }
    }
```

リスト 2 と比べると、リスト 3 の Java 8 コードが使用している基底クラスは異なっており (`UntypedActor` ではなく `AbstractActor`)、またメッセージ処理の代替手段を定義する方法も異なっています。`ReceiveBuilder` クラスを使用すると、多少 Scala に似たマッチング構文で、ラムダ式を使用してメッセージの処理を定義することができます。ほとんどの場合に Scala で開発を行っているとしたら、この手法で Java での Akka コードを少し簡潔にできる場合もありますが、そうでなければ、Java 8 固有のコードを使用するメリットは非常に小さいように思えます。

待機する理由

メインのアプリケーション・コードには、メッセージをアクターに送信した後、システムをシャットダウンする前に、`Thread sleep 1000` という形の待機が含まれていますが、読者の皆さんはどうして待機する必要があるのか不思議に思うかもしれません。何だかんだ言っても、メッセージを処理するのは簡単です。だとしたら、メッセージがすぐにアクターに渡されて、`hello ! "Bob"` ステートメントが完了するまでに処理されないのでしょうか？

この質問に対する簡単な答えは、「いいえ」です。Akka アクターは非同期で実行されるため、ターゲット・アクターが送信側と同じ JVM 内にあるとしても、ターゲット・アクターは即時に実行されません。メッセージ送信を実行するスレッドは、メッセージをターゲット・アクターのメールボックスに追加します。メッセージがメールボックスに追加されるとアクターの `receive` メソッドが呼び出され、それによって、メッセージをメールボックスから取り出して処理するスレッドの実行がトリガーされます。ただし、通常は、メッセージをメールボックスから取り出すスレッドは、メッセージをメールボックスに追加したスレッドとは別のものです。

メッセージ配信のタイミングと保証

「なぜ待機するのか」という質問に対する簡単な答えの背後には、より深い原則があります。Akka がサポートするアクターのリモート機能には、ロケーション透過性が備わっています。つまり、コードでは、特定のアクターが同じ JVM 内にあるか、それともクラウド内のどこかにあるシステムで実行されているのかを把握する直接的な方法がありません。しかし、アクターが同じ JVM 内で実行されているか、それともクラウド内の別のシステムで実行されているかによって、当然、実際の処理における特性はかなり違ってきます。

> *“ Akka は、メッセージが配信されることを保証しません。この配信が保証されないことのベースにある論理的かつ哲学的な根拠は、Akka の中核を成す原則の 1 つになっています。 ”*

その違いの 1 つは、メッセージの喪失に関係します。Akka はメッセージが配信されることを保証しません。このことは、アプリケーション同士を接続するためにメッセージング・システムを使い慣れている開発者にとっては、驚きかもしれません。この配信が保証されないことのベースにある論理的かつ哲学的な根拠は、Akka の中核を成す原則の 1 つになっています。その原則とは、障害を考慮した設計です。単純化し過ぎるかもしれませんが敢えて言うと、配信を保証するとなると、メッセージ転送システムがかなり複雑なものになりますが、それでも、より複雑になったこれらのシステムが想定通りの動作をしない場合があります。その場合、アプリケーション・コードをリカバリーに含めなければなりません。そのことを考えると、常にアプリケーション・コード自体で配信の失敗に対処し、メッセージ転送システムを単純なまま維持できれば、そのほうが理に適っています。

Akka が保証するのは、メッセージが配信されるのは最大でも 1 回であること、そしてあるアクター・インスタンスから別のアクター・インスタンスに送信されるメッセージは順序通りに受信されることです。後者の保証は、特定のアクター・ペアにだけ適用されるものであり、結合的な保証ではありません。どういうことかと言うと、アクター A がアクター B に複数のメッセージを送信する場合、それらのメッセージの順序が狂うことはありません。このことは、アクター A がアクター C にメッセージを送信する場合にも当てはまります。しかし、アクター B もアクター C にメッセージを送信するとなると (例えば、アクター A からのメッセージをアクター C に転送する場合)、アクター B のメッセージは、アクター A から送信されたメッセージに関しては、順序通りに送信されない可能性があります。

リスト 1 のコードは単一の JVM 内で実行され、重いメッセージ負荷は生じることはないので、メッセージ喪失が発生する可能性はごくわずかです (重いメッセージ負荷は、メッセージ喪失の

原因となります。例えば、Akka にメッセージを保管するためのスペースがなくなった場合、メッセージを破棄する以外に対処の方法はありません)。しかしリスト 1 のコードは、メッセージ配信のタイミングに関して何も前提を設けず、アクター・システムを非同期で処理できるように作られています。

アクターと状態

Akka のアクター・モデルは柔軟性があり、あらゆるタイプのアクターを許容します。(`Hello1` の例のように) 状態に関する情報がないアクターを使用することもできますが、そのようなアクターは、メソッド呼び出しと同等なものになりがちです。状態情報を追加すると、遥かに柔軟性に優れたアクター機能を使用できるようになります。

リスト 1 には、アクター・システムの (平凡とは言え) 完全な例が記載されていますが、アクターは常にまったく同じ処理を何度も実行するように制限されています。俳優も、ただ同じ台詞を繰り返しているだけでは飽きてくるので、リスト 4 ではもう少し面白くするために、状態情報をアクターに追加しています。

リスト 4. Scala で作成された、多言語での Hello

```
object Hello2 extends App {

  case class Greeting(greet: String)
  case class Greet(name: String)

  val system = ActorSystem("actor-demo-scala")
  val hello = system.actorOf(Props[Hello], "hello")
  hello ! Greeting("Hello")
  hello ! Greet("Bob")
  hello ! Greet("Alice")
  hello ! Greeting("Hola")
  hello ! Greet("Alice")
  hello ! Greet("Bob")
  Thread sleep 1000
  system shutdown

  class Hello extends Actor {
    var greeting = ""
    def receive = {
      case Greeting(greet) => greeting = greet
      case Greet(name) => println(s"$greeting $name")
    }
  }
}
```

リスト 4 のアクターは、リストの先頭近くで定義された、それぞれにストリング値をラップする `Greeting` メッセージと `Greet` メッセージという 2 つの異なるタイプのメッセージを処理する方法を認識します。変更後の `Hello` アクターは、Greeting メッセージを受信すると、そのメッセージにラップされたストリングを `greeting` 値として保存します。`Greet` メッセージを受信すると、保存した greeting 値にその `Greet` ストリングを結合し、完全なメッセージにします。以下に、このアプリケーションを実行するとコンソールに出力される内容を示します (アクターの実行順序は確定的ではないため、必ずしもここに記載されている順序で出力されるわけではありません)。

```
Hello Bob
Hello Alice
Hola Alice
Hola Bob
```

リスト 4 のコードに目新しい内容はほとんどないので、このコードの Java バージョンは記載しません。Java で作成したコードは、コード・ダウンロード (「参考文献」を参照) に `com.sosnoski.concur.article5java.Hello2` および `com.sosnoski.concur.article5java8.Hello2` として用意されています。

プロパティーと相互作用

実際のアクター・システムでは、互いにメッセージを送信して相互作用する複数のアクターを使用して、作業を行います。通常、これらのアクターには構成情報も提供し、それぞれに固有の役割を演じる準備をさせなければなりません。リスト 5 に、Hello の例で使用した手法を基にした、アクターの構成と相互作用の単純なバージョンを示します。

リスト 5. アクターのプロパティーと相互作用

```
object Hello3 extends App {

  import Greeter._
  val system = ActorSystem("actor-demo-scala")
  val bob = system.actorOf(props("Bob", "Howya doing"))
  val alice = system.actorOf(props("Alice", "Happy to meet you"))
  bob ! Greet(alice)
  alice ! Greet(bob)
  Thread sleep 1000
  system shutdown

  object Greeter {
    case class Greet(peer: ActorRef)
    case object AskName
    case class TellName(name: String)
    def props(name: String, greeting: String) = Props(new Greeter(name, greeting))
  }

  class Greeter(myName: String, greeting: String) extends Actor {
    import Greeter._
    def receive = {
      case Greet(peer) => peer ! AskName
      case AskName => sender ! TellName(myName)
      case TellName(name) => println(s"$greeting, $name")
    }
  }
}
```

リスト 5 には、主役となる新しいアクターとして、`Greeter` アクターが導入されています。`Hello2` の例を進化させて、`Greeter` には以下の要素を追加しました。

- `Greeter` インスタンスを構成するために渡されるプロパティー
- 構成プロパティーとメッセージを定義する Scala コンパニオン・オブジェクト (Java のバックグラウンドをお持ちの方は、コンパニオン・オブジェクトは、アクター・クラスと同じ名前を持つ静的ヘルパー・クラスだと思ってください)。
- `Greeter` アクターのインスタンス間で送信されるメッセージ

上記のコードは、以下の単純な出力を生成します。

```
Howya doing, Alice
Happy to meet you, Bob
```

このコードを何回か実行してみると、行が逆順に出力されることがわかります。このように順序が変わるのも、メッセージが処理される順序は確定的ではないという、Akka アクター・システムの動的な特性を表す一例です (ただし、「メッセージ配信のタイミングと保証」で説明した、いくつかの重要な例外があります)。

Java での `Greeter`

リスト 6 に、通常の Java で作成された、Listing 5 の Akka による `Greeter` コードを記載します。

リスト 6. Java での `Greeter`

```
public class Hello3 {

    public static void main(String[] args) {
        ActorSystem system = ActorSystem.create("actor-demo-java");
        ActorRef bob = system.actorOf(Greeter.props("Bob", "Howya doing"));
        ActorRef alice = system.actorOf(Greeter.props("Alice", "Happy to meet you"));
        bob.tell(new Greet(alice), ActorRef.noSender());
        alice.tell(new Greet(bob), ActorRef.noSender());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) { /* ignore */ }
        system.shutdown();
    }

    // messages
    private static class Greet {
        public final ActorRef target;

        public Greet(ActorRef actor) {
            target = actor;
        }
    }

    private static Object AskName = new Object();

    private static class TellName {
        public final String name;

        public TellName(String name) {
            this.name = name;
        }
    }

    // actor implementation
    private static class Greeter extends UntypedActor {
        private final String myName;
        private final String greeting;

        Greeter(String name, String greeting) {
            myName = name;
            this.greeting = greeting;
        }

        public static Props props(String name, String greeting) {
            return Props.create(Greeter.class, name, greeting);
          }

        public void onReceive(Object message) throws Exception {
            if (message instanceof Greet) {
                ((Greet)message).target.tell(AskName, self());
            } else if (message == AskName) {
                sender().tell(new TellName(myName), self());
            } else if (message instanceof TellName) {
                System.out.println(greeting + ", " + ((TellName)message).name);
```

```
            }
        }
    }
}
```

リスト 7 は、ラムダ式を使用した Java 8 で作成したコードです。このコードも、メッセージ処理の実装は少し簡潔になっていますが、それ以外の点では通常の Java で作成されたコードと同じです。

リスト 7. Java 8 によるコード

```
import akka.japi.pf.ReceiveBuilder;
...
    private static class Greeter extends AbstractActor {
        private final String myName;
        private final String greeting;

        Greeter(String name, String greeting) {
            myName = name;
            this.greeting = greeting;
            receive(ReceiveBuilder.
                match(Greet.class, g -> { g.target.tell(AskName, self()); }).
                matchEquals(AskName, a -> { sender().tell(new TellName(myName), self()); }).
                match(TellName.class, t -> { System.out.println(greeting + ", " + t.name); }).
                build());
        }

        public static Props props(String name, String greeting) {
            return Props.create(Greeter.class, name, greeting);
          }
    }
```

プロパティーの受け渡し

Akka が構成プロパティーをアクターに渡すには、`Props` オブジェクトを使用します。各 `Props` インスタンスは、アクター・クラスが必要とするコンストラクター引数のコピーを、そのクラスへの参照と一緒にラップします。この情報を `Props` コンストラクターに渡すには、2 つの方法があります。リスト 5 の例では、アクターのコンストラクターを名前渡しの引数として `Props` コンストラクターに渡します。この方法では、コンストラクターを即座に呼び出して結果を渡すようにはならないことに注意してください。この方法で渡すのは、コンストラクター呼び出しです (Java のバックグラウンドをお持ちの方は、奇妙に思えるかもしれません)。

アクターの構成情報を `Props` コンストラクターに渡すもう 1 つの方法は、アクターのクラスを 1 番目の引数として指定し、アクターのコンストラクター引数を残りのパラメーターとして指定することです。リスト 5 の例の場合、この形での呼び出しは `Props(classOf[Greeter], name, greeting)` となります。

どちらの形の `Props` コントラクターを使用するかに関わらず、アクター・インスタンスが実行されるときに必要に応じて `Props` をネットワークで送信できるよう、新しく誕生したアクターに渡す値はシリアライズ可能でなければなりません。リスト 5 で使用している名前渡しのコンストラクター呼び出しでは、呼び出しのクロージャーを JVM から送信する必要があるときに、クロージャーはシリアライズされてから送信されます。

Scala コードで `Props` オブジェクトを作成する場合、Akka で推奨しているプラクティスは、リスト 5 で行っているように、コンパニオン・オブジェクト内にファクトリー・メソッドを定義する

ことです。この手法により、`Props` に対する名前渡しのコンストラクター呼び出し手法を使用する際に、誤ってアクター・オブジェクトへの `this` 参照を閉じてしまったときに起こり得るあらゆる問題を防ぐことができます。コンパニオン・オブジェクトは、アクターが受信するメッセージを定義するのにも最適な場所です。ここにメッセージを定義すれば、すべての関連する情報が 1 つの場所に集約されます。Java アクターの場合、リスト 6 で使用しているようなアクター・クラス内部の静的コンストラクター・メソッドが功を奏します。

メッセージを送信するアクター

リスト 5 の `Greeter` アクターのそれぞれは、名前と挨拶で構成されていますが、別のアクターに挨拶するよう指示された場合は、まず始めに、その別のアクターの名前を見つける必要があります。`Greeter` アクターはそのために、その別のアクターに `AskName` メッセージを別途送信します。`AskName` メッセージ自体には情報が含まれませんが、このメッセージを受信した `Greeter` インスタンスは、`TellName` 送信側の名前を格納した `TellName` メッセージで応答することを認識しています。メッセージを送信する `Greeter` は `TellName` メッセージを受信すると、その挨拶を出力します。

アクターに送信される各メッセージには、Akka が提供する追加情報を伴っています。中でも最も注目すべき情報は、メッセージ送信元としての `ActorRef` です。メッセージの処理中は随時、アクターの基底クラスに定義された `sender()` メソッドを呼び出すことで、この送信元の情報にアクセスすることができます。`Greeter` アクターは、`AskName` メッセージの処理を行う際にこの送信元参照を使用するため、`TellName` レスポンスが正しいアクターに送信されます。

Akka では、ユーザーが別のアクターに代わってメッセージを送信できるようになっています (なりすましの無害な形)。その場合、メッセージを受信するアクターは、その別のアクターを送信元とみなします。この機能は、アクター・システムで使用するのに役立つことがよくあります。これは特に、リクエスト/レスポンス・タイプのメッセージ交換で、リクエストを行ったアクターではないアクターに応答する必要がある場合に言えることです。アクター外部のアプリケーション・コードによって送信されるメッセージでは、デフォルトの送信元として、「デッドレター」アクターと呼ばれる特殊な Akka アクターが使用されます。デッドレター・アクターは、メッセージをアクターに配信できないときにも使用されます。この場合、デッドレター・アクターは適切なロギングを有効にすることで、アクター・システムにおける配信不能メッセージを追跡する便利な手段を提供します (これについては、次回の記事で取り上げます)。

アクターのタイプの指定

お気付きかもしれませんが、今までの例に記載されている一連のメッセージにはどこにも、メッセージのターゲットが `Greeter` インスタンスであることを明示的に示すタイプ情報がありません。これは、Akka のアクターとこれらのアクター間で交換されるメッセージでは通常のことです。メッセージのターゲット・アクターを識別するために使用される `ActorRef` にさえも、タイプは指定されません。

タイプが指定されていないアクター・システムをプログラミングすることには、実用面でのメリットがあります。アクター・タイプを定義することは可能ですが (例えば、アクターが処理できるメッセージのセットによって定義することができます)、そうすると、誤った結果を導きかねません。Akka では、アクターの振る舞いが変更される可能性があります (これについては、次回の記事で詳しく説明します)。アクターの状態が変われば、適切なメッセージのセットも異なってき

ます。アクター・モデルでは、すべてのアクターが一応は任意のメッセージを処理できるものとして扱うため、タイプはアクター・モデルを単純明快にする妨げにもなりがちです。

そうは言っても、どうしてもタイプを指定しなければならないのであれば、Akka ではタイプが指定されたアクターをサポートしています。このサポートが主に役立つのは、アクターと非アクター・コードとの間のインターフェースを取る場合です。非アクター・コードから使用できるインターフェースを定義してアクターと連動できるようにすることで、アクターを通常のプログラム・コンポーネントのように見せることができます。ほとんどの目的において、アクター・システムの外部からでも簡単に直接アクターにメッセージを送信できることを考えると (このことは、これまでに記載した非アクター・コードがメッセージを送信するサンプル・アプリケーションのどれをとっても明らかです)、タイプを指定するのは面倒な割に得るものが少ないかもしれませんが、こうした方法を使用できることは素晴らしいことです。

メッセージと可変性

Akka では、誤って可変のデータをアクター間で共有しないよう注意する必要があります。可変のデータを共有すると、非常にまずい事態になる可能性があります。ゴーストと戦っているときにプロトン・パックのビームに当たった場合ほど深刻な事態ではありませんが (何を言っているのかわからない読者は、「ゴーストバスターズ」を参照してください)、それでもかなりまずい事態です。可変データを共有することによって起こる問題は、複数のアクターが別々のスレッドで実行されることにあります。アクター間で可変のデータを共有すると、アクターを実行するスレッド間での調整が行われないため、アクターは他のスレッドが何を実行しているかがわからずに、互いの処理をさまざまに妨害する結果となる可能性があります。分散システムを実行していて、アクターごとに固有の可変データのコピーがあるとしたら、事態はさらに悪化します。

したがって、メッセージは不変でなければなりません。これは表面レベルだけの話ではありません。メッセージ・データに含まれているオブジェクトも同じく不変でなければならず、メッセージから参照されるすべての要素のクロージャーに至るまでのあらゆるものが不変でなければならないのです。現時点では、Akka がこの要件を適用することはできませんが、Akka の開発者たちは将来のある時点で、制約を設けようとしています。Akka の今後のバージョンでもコードを使用できるようにしたい場合は、今からこの要件に注意する必要があります。

ask と tell の違い

リスト 5 のコードは、メッセージを送信するために標準的な `tell` の処理を使用しています。Akka では、補助的な処理として ask メッセージ・パターンを使用することもできます。`ask` 処理 (`?` 演算子、または `ask` 関数を使って示されます) は、レスポンスを受け取るために `Future` を使用してメッセージを送信します。リスト 8 に、`tell` ではなく `ask` だけを使用するように再構築したリスト 5 のコードを示します。

リスト 8. `ask` を使用するコード

```
import scala.concurrent.duration._
import akka.actor._
import akka.util._
import akka.pattern.ask

object Hello4 extends App {

  import Greeter._
  val system = ActorSystem("actor-demo-scala")
```

```
  val bob = system.actorOf(props("Bob", "Howya doing"))
  val alice = system.actorOf(props("Alice", "Happy to meet you"))
  bob ! Greet(alice)
  alice ! Greet(bob)
  Thread sleep 1000
  system shutdown

  object Greeter {
    case class Greet(peer: ActorRef)
    case object AskName
    def props(name: String, greeting: String) = Props(new Greeter(name, greeting))
  }

  class Greeter(myName: String, greeting: String) extends Actor {
    import Greeter._
    import system.dispatcher
    implicit val timeout = Timeout(5 seconds)
    def receive = {
      case Greet(peer) => {
        val futureName = peer ? AskName
        futureName.foreach { name => println(s"$greeting, $name") }
      }
      case AskName => sender ! myName
    }
  }
}
```

リスト 8 のコードでは、`TellName` メッセージが `ask` で置き換えられています。`ask` 処理によって返される future の型は、`Future[Any]` です。これは、返される結果について、コンパイラーは何の情報も持っていないためです。future の処理が完了すると、`foreach` は、`import system.dispatcher` ステートメントによって定義された暗黙的なディスパッチャーを使用して `println` を実行します。future が、許容されるタイムアウト期間内 (タイムアウトも暗黙的な値です。この例では 5 秒として定義されています) にレスポンス・メッセージを受け取って完了しなければ、代わりにタイムアウト例外で完了します。

裏では、`ask` パターンによって、特殊化されたワンショットのアクターが作成されます。このアクターは、メッセージ交換で仲介役として機能します。仲介役には、目的のアクター参照だけでなく、`Promise` と送信対象のメッセージも渡されます。仲介役はそのメッセージを送信した後、想定されるレスポンス・メッセージを待機します。レスポンスを受け取ると、Promise を履行して、元のアクターによって使用された future の処理を完了します。

`ask` を使用する手法には、いくつかの制約事項があります。特に、アクターの状態が公開されるのを避けるには (アクターの状態が公開されると、スレッド化の問題が生じる可能性があります)、future の処理完了時に実行されるコードでは、アクターの可変の状態を使用しないようにしなければなりません。実用的な観点では、アクター間で送信されるメッセージには、`tell` パターンを使用するほうが通常は簡単です。ask パターンのほうが有用になるのは、例えば、(アクターのタイプが指定されているかどうかに関わらず) アクターからのレスポンスを受け取らなければならないアプリケーション・コードを扱っている場合です (アクター・システムを起動して、最初のアクターを作成するメイン・プログラムなど)。

端役のアクター

> “新しいアクターを導入することが、簡潔な方法で非同期処理に対処するのに役立つのであれば、ためらわずにいつでも設計に導入してください。”

ask パターンで作成されるワンショットのアクターは、Akka を使用する際に念頭に置いておくべき有効な設計原則の一例です。特定の目的のために作成される特殊なアクターが中間処理のステップを実行するようにアクター・システムを構築することが望ましいケースはよくあります。その一般的な例の 1 つは、異なる非同期処理の結果をマージしてから、処理の次の段階に進まなければならない場合です。異なる結果に対してメッセージを使用すれば、アクターに結果を収集させて、すべての結果が揃ってから次のステージを開始することができます。これは基本的に、ask パターンで使用されるワンショットのアクターを一般化したものです。

Akka のアクターは軽量なので (アクター・インスタンスあたり約 300 バイトから 400 バイトと、アクター・クラスが使用する何らかのストレージのみ)、必要に応じて多数のアクターを使用するような構造の設計にしても害はありません。特殊化したアクターを使用すると、コードを単純かつわかりやすい状態に維持することができます。このメリットは、手続き型プログラムを作成する場合より、並行プログラムを作成する場合には、より大きなメリットになります。新しいアクターを導入することが、簡潔な方法で非同期処理に対処するのに役立つのであれば、ためらわずにいつでも設計に導入してください。

幕あい

Akka は強力なシステムですが、Akka とアクター・モデルは一般に、直接的な手続き型コードとは異なるスタイルのプログラミングが必要になります。手続き型コードの場合、プログラム構成でのすべての呼び出しは確定的なので、プログラムの呼び出しツリー全体を把握することができます。アクター・モデルでは、メッセージは最終的に配信されることが保証されずにオプティミスティックに起動されるため、大抵は処理の順序を確立するのが困難です。アクター・モデルのメリットは、並行性とスケーラビリティーに優れたアプリケーションを簡単に構築できることにあります。この点については、今度の記事で再び取り上げます。

この記事を読んで、皆さんが Akka を十分に味わい、さらに興味が刺激されたことを願います。次回は、アクター・システムとアクターの相互作用についてさらに深く探り、システム内のアクター間の相互作用を簡単に追跡する方法についても見ていきます。

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。