JVM の並行性: Scala での非同期イベント処理

async マクロによる簡単なノンブロッキング型コードを含め、ブロッキング手法とノンブロッキング手法を学ぶ

Dennis Sosnoski
Principal Consultant
Sosnoski Software Solutions Inc.

2014年 11月 20日

Scala の `Future` クラスと `Promise` クラスは、イベントを編成して順序付けるノンブロッキング手法を含め、非同期処理を扱う強力な手段を提供します。しかも Scala では、マクロとして実装された `async` / `await` 構成体によって、ノンブロッキング型のイベント処理をより簡単に作成することができます。このチュートリアルでは、`Future` と `Promise` を使用する基本手法の実際を紹介した後、`async` / `await` を利用して、単純なブロッキング型のように見えるコードをノンブロッキング方式で実行されるように変換する方法をデモンストレーションします。

このシリーズの他の記事を見る

並行処理アプリケーションにおいて、非同期イベント処理は極めて重要です。イベントのソースが何であれ (個々の計算タスク、I/O 処理、あるいは外部システムとのやりとりである場合もあります)、並行処理アプリケーションのコードはイベントを追跡するとともに、それらのイベントに応じて行われるアクションを調整しなければなりません。アプリケーションでは非同期イベント処理に対して、以下の 2 種類の基本手法のうちのいずれかを採ることができます。

- **ブロッキング**: 調整スレッドにイベントを待機させます。
- **ノンブロッキング**: 明示的な形でスレッドにイベントを待機させなくても、イベントが何らかの形でアプリケーションへの通知を生成します。

記事「JVM の並行性: ブロックすべきか、すべきでないか？」では、Java 8 で非同期イベントを処理する方法を説明しました。そこではブロッキング手法とノンブロッキング手法のそれぞれで `CompletableFuture` クラスを使用しました。今回のチュートリアルでは、Scala で非同期イベント処理を行う方法のいくつかを紹介します。まずは単純なブロッキング手法から始め、その後でノンブロッキング手法をいくつか紹介します。そして最後に、`async` / `await` 構成体によって、単純なブロッキング型のように見えるコードをノンブロッキング方式で実行されるように変換する方法をデモンストレーションします (著者の GitHub リポジトリーから完全なサンプル・コードを入手してください)。

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。

複数のイベントの作成

`scala.concurrent.Promise` クラスと `scala.concurrent.Future` クラスは、Java 8 開発者が `CompletableFuture` を使用して得られるのと同様の選択肢を Scala 開発者に提供します。具体的に言うと、`Future` にはイベントの完了を扱うための手段として、ブロッキング手法とノンブロッキング手法の両方が用意されています。このレベルでは共通しているとは言え、2 つのタイプの Future を扱うために使用する手法は異なります。

このセクションでは、`Future` で表されるイベントをブロッキング手法で扱う例とノンブロッキング手法で扱う例を検討します。このチュートリアルで使用する並行処理タスクは、前回セットアップしたものと同じです。コードの詳細を探る前に、そのセットアップについて簡単に復習しておきましょう。

タスクと順序付け

アプリケーションでは、ある特定のオペレーションの中で、複数の処理ステップを実行しなければならないことがよくあります。例えば Web アプリケーションで、ユーザーにレスポンスを返す前に、以下の処理を実行しなければならないとします。

1. データベースでユーザーの情報を検索する
2. 検索した情報を使用して、Web サービスの呼び出し、さらに場合によっては別のデータベース・クエリーを実行する
3. 上記 2 つのステップでの処理の結果に基づいて、データベースを更新する

図 1 に、このようなタイプのアプリケーションを構成するタスクのフローを示します。

図 1. アプリケーションのタスク・フロー

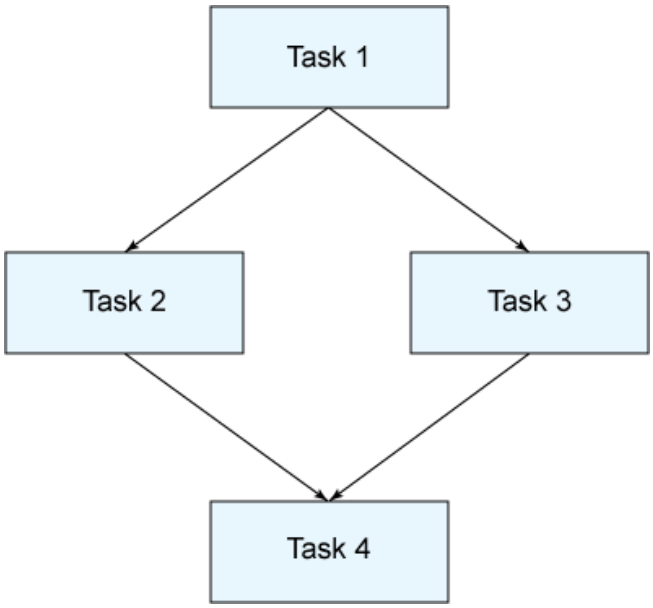

図 1 では、全体の処理を 4 つの個別のタスクに分割し、タスク間を矢印で結んで順序の依存関係を表しています。タスク 1 はすぐに実行することができます。タスク 2 とタスク 3 は両方ともタスク 1 が完了した後に実行されます。タスク 4 は、タスク 2 とタスク 3 の両方が完了した後に実行されます。

非同期イベントのモデル化

実際のシステムでは、非同期イベントのソースは一般に、並列計算処理または何らかの形での I/O 処理のいずれかとなります。しかし上記の例のようなシステムをモデル化するには、単純な時間遅延を使用したほうが簡単なので、ここでは時間遅延の手法を使用します。リスト 1 に、完了済み `Future` の形でイベントを生成するために使用する、基本的な時間指定イベントのコードを記載します。

リスト 1. 時間指定イベントのコード

```
import java.util.Timer
import java.util.TimerTask

import scala.concurrent._

object TimedEvent {
  val timer = new Timer

  /** Return a Future which completes successfully with the supplied value after secs seconds. */
  def delayedSuccess[T](secs: Int, value: T): Future[T] = {
    val result = Promise[T]
    timer.schedule(new TimerTask() {
      def run() = {
        result.success(value)
      }
    }, secs * 1000)
    result.future
  }

  /** Return a Future which completes failing with an IllegalArgumentException after secs
    * seconds. */
```

```
  def delayedFailure(secs: Int, msg: String): Future[Int] = {
    val result = Promise[Int]
    timer.schedule(new TimerTask() {
      def run() = {
        result.failure(new IllegalArgumentException(msg))
      }
    }, secs * 1000)
    result.future
  }
```

前回の記事に記載した Java コードと同じように、リスト 1 の Scala コードでは遅延後の `java.util.TimerTask` の実行をスケジューリングするために `java.util.Timer` を使用しています。各 `TimerTask` は実行時に、それぞれに関連付けられた future を完了します。`delayedSuccess` 関数は、実行されると Scala の `Future[T]` が成功して完了するようにタスクをスケジューリングし、future を呼び出し側に返します。`delayedFailure` 関数は同じタイプの future を返しますが、この関数が使用するタスクは、`llegalArgumentException` で失敗して future を完了するタスクです。

リスト 2 に、リスト 1 のコードを使用して、図 1 の 4 つのタスクに対応するイベントを `Future[Int]` の形で作成する方法を示します (このコードは、サンプル・コードの `AsyncHappy` クラスから抜粋したものです)。

リスト 2. サンプル・タスクに対応するイベント

```
// task definitions
def task1(input: Int) = TimedEvent.delayedSuccess(1, input + 1)
def task2(input: Int) = TimedEvent.delayedSuccess(2, input + 2)
def task3(input: Int) = TimedEvent.delayedSuccess(3, input + 3)
def task4(input: Int) = TimedEvent.delayedSuccess(1, input + 4)
```

リスト 2 の 4 つのタスク・メソッドのそれぞれは、特定の遅延値を用いて、そのタスクが完了するときに対応します。遅延値としては、`task1` には 1 秒、`task2` には 2 秒、`task3` には 3 秒、そして `task4` には再び 1 秒が使用されます。また、それぞれのタスク・メソッドは入力値を引数に取り、その入力値にタスク番号を加算したものを future の (最終的な) 結果値として使用します。これらのメソッドはすべて future の成功した形を用いています。失敗した形を用いる例は、後で記載します。

これらのタスクで意図しているのは、図 1 に示した順序でタスクを実行し、各タスクに先行するタスクから返された結果値 (`task4` の場合、2 つの先行するタスクからの結果値の合計) を渡すことです。2 つの中間タスクが同時に実行された場合、実行時間の合計は約 5 秒となります (1 秒 + max(2 秒, 3 秒) + 1 秒)。`task1` への入力値が 1 の場合、結果値は 2 です。その結果値が `task2` と `task3` に渡されると、その結果値は 4 と 5 となります。この 2 つの結果値の合計 (9) が `task4` に入力値として渡されると、最終的な結果値は 13 となります。

ブロッキング待機

これで準備はできたので、早速 Scala ではイベントの完了をどのように扱うのかを見てみましょう。前回の記事の Java コードと同じく、4 つのタスクの実行を調整する最も簡単な方法は、ブロッキング待機を使用することです。つまり、メイン・スレッドに、4 つのタスクのそれぞれが順に完了するのを待機させます。リスト 3 (同じくサンプル・コードの `AsyncHappy` クラスからの抜粋) に、この手法を示します。

リスト 3. タスクのブロッキング待機

```
def runBlocking() = {
  val v1 = Await.result(task1(1), Duration.Inf)
  val future2 = task2(v1)
  val future3 = task3(v1)
  val v2 = Await.result(future2, Duration.Inf)
  val v3 = Await.result(future3, Duration.Inf)
  val v4 = Await.result(task4(v2 + v3), Duration.Inf)
  val result = Promise[Int]
  result.success(v4)
  result.future
}
```

リスト 3 では、Scala の `scala.concurrent.Await` オブジェクトの `result()` メソッドを使用してブロッキング待機を行っています。このコードはまず `task1` の結果を待機します。その後、`task2` と `task3` の両方の future を作成して順にそれぞれを待機し、最後に `task4` の結果を待機します。最後の 3 行 (`result` を作成し、設定する部分) により、メソッドは `Future[Int]` を返すことができます。future を返すことによって、このメソッドと次に説明するノンブロッキング手法との整合性を取っているわけですが、実際にはメソッドがリターンする前に、future は完了するはずです。

future を結合する

リスト 4 (同じくサンプル・コードの `AsyncHappy` クラスからの抜粋) に、複数の future を互いに関連付けて、タスクを正しい順序と依存関係で一切ブロッキングすることなく実行する 1 つの方法を示します。

リスト 4. `onSuccess()` による完了の処理

```
def runOnSuccess() = {
  val result = Promise[Int]
  task1(1).onSuccess(v => v match {
    case v1 => {
      val a = task2(v1)
      val b = task3(v1)
      a.onSuccess(v => v match {
        case v2 =>
          b.onSuccess(v => v match {
            case v3 => task4(v2 + v3).onSuccess(v4 => v4 match {
              case x => result.success(x)
            })
          })
      })
    }
  })
  result.future
}
```

リスト 4 のコードは、`onSuccess()` メソッドを使用して各 future の完了時に実行される関数を設定しています (厳密に言うと、このコードは正常に完了したケースだけを扱うため、部分関数です)。`onSuccess()` 呼び出しはネストされていることから、(future がすべて順序通りに完了しないとしても) 順に実行されます。

リスト 4 のコードはそこそこ理解しやすいものの、冗長です。リスト 5 に、`flatMap()` メソッドを使用して、同じケースをより簡潔な方法で処理する例を示します。

リスト 5. `flatMap()` による完了の処理

```
def runFlatMap() = {
  task1(1) flatMap {v1 =>
    val a = task2(v1)
    val b = task3(v1)
    a flatMap { v2 =>
      b flatMap { v3 => task4(v2 + v3) }}
  }
}
```

リスト 5 のコードで行っていることは、実質的にはリスト 4 のコードと同じです。ただし、リスト 5 では `flatMap()` メソッドを使用して、各 future から単一の結果値を抽出します。`flatMap()` を使用すると、リスト 4 では必要だった `match` / `case` 構成体が不要になるため、より簡潔な形になる一方、同じステップ・バイ・ステップの実行パスが維持されます。

サンプル・コードを試してみる

サンプル・コードでは、Scala の `App` を使用してイベント・コードの各バージョンを順に実行し、完了するまでの時間 (約 5 秒) が適切で結果 (13) が正しくなるようにしています。このコードは、リスト 6 に示すように (余分な Maven 出力は削除されています)、Maven を使用してコマンド・ラインから実行することができます。

リスト 6. イベント・コードの実行

```
dennis@linux-9qea:~/devworks/scala4/code> mvn scala:run -Dlauncher=happypath
...
[INFO] launcher 'happypath' selected => com.sosnoski.concur.article4.AsyncHappy
Starting runBlocking
runBlocking returned 13 in 5029 ms.
Starting runOnSuccess
runOnSuccess returned 13 in 5011 ms.
Starting runFlatMap
runFlatMap returned 13 in 5002 ms.
##
```

アンハッピー・パス

ここまでは、常にタスクの完了に成功する future という形でイベントを調整するコードを見てきました。実際のアプリケーションでは、常にこのハッピー・パスにとどまることを当てにはできません。タスクの処理中に問題が発生した場合、これらの問題は、JVM 言語の用語で言う、`Throwable` で表現されるのが通常です。

リスト 2 のタスク定義を変更して、`delayedSuccess()` メソッドの代わりに `delayedFailure()` が使用されるようにするのは簡単です。以下に、このように変更した `task4` のタスク定義を記載します。

```
def task4(input: Int) = TimedEvent.delayedFailure(1, "This won't work!")
```

task4 だけを例外で完了するように変更してリスト 3 のコードを実行すると、`task4` に対する `Await.result()` 呼び出しにより、当然 `IllegalArgumentException` がスローされます。この問題が `runBlocking()` メソッドでキャッチされない場合、最終的にキャッチされるまで呼び出し

チェーンで例外が渡されます (最後までキャッチされなければスレッドが強制終了されることになります)。タスクのいずれかが例外で完了した場合、その例外が呼び出し側に渡されるようにして、呼び出し側では返された future を使用して処理が行われるようにコードを変更するのは、幸いにも簡単です。リスト 7 に、変更後のコードを記載します。

リスト 7. 例外を考慮したブロッキング待機

```
def runBlocking() = {
  val result = Promise[Int]
  try {
    val v1 = Await.result(task1(1), Duration.Inf)
    val future2 = task2(v1)
    val future3 = task3(v1)
    val v2 = Await.result(future2, Duration.Inf)
    val v3 = Await.result(future3, Duration.Inf)
    val v4 = Await.result(task4(v2 + v3), Duration.Inf)
    result.success(v4)
  } catch {
    case t: Throwable => result.failure(t)
  }
  result.future
}
```

リスト 7 では、元のコードは `try` / `catch` でラップされており、future が完了して返されるときに `catch` は例外を渡します。この手法はコードを多少複雑にするものの、Scala 開発者であれば簡単にコードを理解できるはずです。

リスト 4 とリスト 5 でのイベント処理コードをノンブロッキング・バージョンにすると、どうなるでしょう？リスト 4 で使用した `onSuccess()` メソッドは、その名の通り、future が成功して完了した場合だけを扱います。成功して完了した場合と失敗して完了した場合の両方を扱うには、代わりに `onComplete()` メソッドを使用して、どちらのタイプの完了が適用されるのかを確かめる必要があります。この手法がイベント処理コードにどのように適用されるかをリスト 8 に示します。

リスト 8. 成功と失敗の両方を扱う `onComplete()`

```
def runOnComplete() = {
  val result = Promise[Int]
  task1(1).onComplete(v => v match {
    case Success(v1) => {
      val a = task2(v1)
      val b = task3(v1)
      a.onComplete(v => v match {
        case Success(v2) =>
          b.onComplete(v => v match {
            case Success(v3) => task4(v2 + v3).onComplete(v4 => v4 match {
              case Success(x) => result.success(x)
              case Failure(t) => result.failure(t)
            })
            case Failure(t) => result.failure(t)
          })
        case Failure(t) => result.failure(t)
      })
    }
    case Failure(t) => result.failure(t)
  })
  result.future
}
```

リスト 8 は面倒なコードのように見えますが、幸い、これよりも遥かに簡単な方法があります。それは、代わりにリスト 5 の `flatMap()` コードを使用するというものです。`flatMap()` の手法では、変更を一切加えることなく、成功した場合と失敗した場合の両方を扱います。

`async` を使用する方法

Scala の最近のバージョンには、「マクロ」を使用してコンパイル時にコードを変換する機能が組み込まれています。これまでに実装されたマクロのうち、最も有用なマクロの 1 つとして挙げられるのは、`async` です。このマクロは、future を使用する明らかにシーケンシャルなコードを、コンパイル時に非同期コードに変換します。リスト 9 に、このチュートリアルで使用しているタスク・コードを `async` によって単純化する方法を示します。

リスト 9. future と `async {}` の結合

```
def runAsync(): Future[Int] = {
  async {
    val v1 = await(task1(1))
    val a = task2(v1)
    val b = task3(v1)
    await(task4(await(a) + await(b)))
  }
}
```

リスト 9 の `async {...}` ブロックは、`async` マクロを呼び出します。この呼び出しは、このブロックが非同期であること、そしてデフォルトでは非同期で実行されることを宣言し、ブロックの実行結果として future を返します。ブロック内部の `await()` メソッド (実際には、真のメソッドではなく、このマクロのキーワード) は、future の結果が必要となる場所を示します。async マクロは、コンパイル時に Scala プログラムの抽象構文木 (AST) に変更を加え、このブロックを変換してコールバックを使用するコードにします。変換後のコードはリスト 4 のコードとほぼ同じになります。

`async {...}` ラッパーを別とすれば、リスト 9 のコードは、リスト 3 に記載されている元のブロッキング・コードとほとんど同じように見えます。これがまさに、このマクロの偉業です。このマクロによって、非同期イベントの複雑な処理はすべて抽象化され、単純な線形コードを作成しているように見えるコードにすることができます。その一方、裏ではかなりの量の複雑な処理が行われています。

`async` の内部構造

Scala コンパイラーによってソース・コードから生成されたクラスを調べると、`AsyncHappy$$anonfun$1.class` のような名前を持つ複数の内部クラスがあることに気付くはずです。名前から想像できるように、これらの内部クラスは、コンパラーが非同期関数 (`onSuccess()` メソッドまたは `flatMap()` メソッドに渡されるステートメントなど) を対象に生成します。

Scala 2.11.1 コンパイラーと Async 0.9.2 実装の場合、`AsyncUnhappy$stateMachine$macro$1$1.class` という名前のクラスもあります。これは、`async` マクロによって生成される実際の実装コードであり、非同期タスクを扱うステート・マシンの形になっています。リスト 10 に、このクラスをデコンパイルしたビューの一部を記載します。

リスト 10. デコンパイルされた `AsyncUnhappy$stateMachine$macro$1$1.class`

```
public class AsyncUnhappy$stateMachine$macro$1$1
  implements Function1<Try<Object>, BoxedUnit>, Function0.mcV.sp
{
  private int state;
  private final Promise<Object> result;
  private int await$macro$3$macro$13;
  private int await$macro$7$macro$14;
  private int await$macro$5$macro$15;
  private int await$macro$11$macro$16;
  ...
  public void resume() {
    ...
  }

  public void apply(Try<Object> tr) {
    int i = this.state;
    switch (i) {
      default:
        throw new MatchError(BoxesRunTime.boxToInteger(i));
      case 3:
        if (tr.isFailure()) {
          result().complete(tr);
        } else {
          this.await$macro$11$macro$16 = BoxesRunTime.unboxToInt(tr.get());
          this.state = 4;
          resume();
        }
        break;
      case 2:
        if (tr.isFailure()) {
          result().complete(tr);
        } else {
          this.await$macro$7$macro$14 = BoxesRunTime.unboxToInt(tr.get());
          this.state = 3;
          resume();
        }
        break;
      case 1:
        if (tr.isFailure()) {
          result().complete(tr);
        } else {
          this.await$macro$5$macro$15 = BoxesRunTime.unboxToInt(tr.get());
          this.state = 2;
          resume();
        }
        break;
      case 0:
        if (tr.isFailure()) {
          result().complete(tr);
        } else {
          this.await$macro$3$macro$13 = BoxesRunTime.unboxToInt(tr.get());
          this.state = 1;
          resume();
        }
        break;
    }
  }
  ...
}
```

リスト 10 の `apply()` メソッドは、future の結果を評価し、それに併せて出力ステートを変更することで、実際のステート変更を扱います。入力ステートはコードに対し、現在評価対象となっ

ている future を示します。つまり、それぞれのステート値が、async ブロック内の 1 つの特定の future に対応するということです。リスト 10 に記載したコードの抜粋を見ただけで、このことを理解するのは難しいと思いますが、他のバイトコードをある程度見ると、ステート・コードがタスクに対応していることがわかるはずです。具体的には、ステート `0` は `task1` の結果が要求されていることを意味し、ステート `1` は `task2` の結果が要求されていることを意味するといった具合です。

リスト 10 には、`resume()` メソッドは示されていません。デコンパイラーは、このメソッドを Java コードに変換する方法を明らかにすることができなかったためです。この課題を掘り下げることはしませんが、バイトコードを見ると、`resume()` メソッドはステート・コードに対して Java の `switch` と同等の処理を行うことがわかります。非終了ステートのそれぞれで、`resume()` はコードの適切なフラグメントを実行して次に要求される future をセットアップし、その future の `onComplete()` メソッドのターゲットとして `AsyncUnhappy$stateMachine$macro$1$1` を設定して終了します。終了ステートに対しては、`resume()` は結果値を設定し、最終結果の promise を完了します。

実際には、async を理解する上で、生成されたコードを掘り下げる必要はありません (ただし、生成されたコードを掘り下げると興味深いことがわかる可能性があります)。async の仕組みについての詳細は、SIP-22 - Async のプロポーザルに記載されています。

`async` の制限事項

`async` マクロ使用する際には、いくつかの制限事項があります。こうした制限事項が存在する理由は、`async` マクロがコードをステート・マシン・クラスに変換する方法にあります。最も大きな制限事項は、`async` ブロック内の別のオブジェクトやクロージャー (関数定義を含む) の中では `await()` をネストできないことです。また、`try` あるいは `catch` の中で `await()` をネストすることもできません。

これらの使用法に関する制限事項は別として、`async` で最大の問題となるのは、デバッグです。デバッグするとなると、非同期コードに関連することの多いコールバック地獄に陥ってしまいます。async の場合、それは見掛け上のコード構造を反映していないコール・スタックを理解するのが難しいためです。あいにく、現在のデバッガー設計では、これらの問題を回避する手段はありません。この領域については、Scala で新しい取り組みが行われています (「Rethinking the debugger」を参照)。その一方で、デバッグしやすくするには、とりあえず `async` ブロックの非同期実行を無効にするという方法を取ることができます (シーケンシャルに実行すると、修正しようとしている問題がまだ発生するという前提です)。

最後に伝えておくべき点として、Scala のマクロは現在進行中の取り組みです。この取り組みの目的は、将来のリリースで、`async` を Scala 言語に正式に組み込むことですが、Scale 言語チームがマクロの機能に満足しない限り実現しないでしょう。その日が来るまで、`async` の形が変わらないという保証はありません。

まとめ

非同期イベント処理に対する Scala の手法のいくつかは、「JVM の並行性: ブロックすべきか、すべきでないか？」で説明した Java コードとは大幅に異なります。Scala では、`flatMap()` および `async` マクロの両方により、簡潔で理解しやすい非同期イベント処理手法を提供しています。特

に `async` は、通常のシーケンシャルなコードのように見えるコードを作成できる一方、コンパイルされたコードは並行して実行されるという点で、興味深いマクロとなっています。このタイプの手法を提供している言語は Scala だけではありませんが、マクロ・ベースの実装は、他の手法に極めて高い柔軟性をもたらします。

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。