ゼミ: Learning Concurrent Programming in Scala

久野　靖 *

2015.9.11

□ Aleksandar Prokopec, Learning Concurrent Programming in Scala, Pack Publishing, 2014.

- 文字通り Scala で並行プログラミングの本
- さまざまなレベル (従来の低レベルから抽象化されたフレームワークまで) をひととおり扱う

1 Preface

□ 並行性が広く使われるようになった ← マルチコア CPU

- かっては学術的な話題だったが今では実用的に必要
- このため多くのライブラリやフレームワークが登場

□ 言語/ライブラリ/フレームワークの抽象度 UP →いつ何を使うかの判断必要

- 従来型のスレッド/ロック/モニタなどの知識だけでは不十分
- 従来型のものの弱点を補う新たな並行フレームワークも知る必要

□ 本書では Scala を用いて高レベルの並行プログラミングを扱う

- 並行に関するさまざまな話題を詳細まで扱う
- 現代的な並行フレームワークについても扱う

□ 本書を通じて、理論的基盤の理解と実践的スキルの両方を獲得

1.1 How this book is organized

□ 実際に役立つため動く例題プログラムを扱う

□ 各プログラムは短く自己完結

- しかし架空の例というわけではなく実際に必要な場面の抽出

□ 実際に自分で動かしてみることを薦める

- 各例題で新しい概念が学べるが、理解には実地に試すことが必要
- だから SBT をダウンロードしてぜひ動かしてみて

□ 各章末にはプログラミングの練習問題

- それぞれの章の概念を理解したか試せる→せひ 2〜3 個はやってみてほしい

□ API メソッドの一覧を示したりは基本的にしない

- API は ScalaDoc で見られるし、変更も頻繁にあるので本に向かない

□ 本書は並列 API の詳細まで学ぶのが目的ではない

- 重要な概念を把握することが目的
- 本書を読了したら各ライブラリの働きだけでなく「いかに考えるか」が分かる

1.2 What this book covers

□ おおまかな話題単位で章に分割してある

- 標準の並行 API に加え、複雑な機能や高レベルの抽象化なども
- ch1 Introduction --- 並行プログラミングの必要、Scala 入門
- ch2 Concurrency on JVM and Java Memory Model --- 並行プログラミングの基本、スレッドの生成消滅、共有メモリの操作
- ch3 Traditional Building Blocs of Concurrency --- 伝統的な並行動作の道具だて、スレッドプール、アトミック変数、並行コレクション（あくまで基本的なところ）
- ch4 Asynchronous Programming with Futures and Promises --- Scala 固有の部分。future/promise API とその使い方
- ch5 Data-Parallel Collections --- 並列コレクションを用いて操作を並列化することについて
- ch6 Concurrent Programming with Reactive Extensions --- イベント駆動/非同期プログラミングで用いるフレームワーク
- ch7 Software Transactional Memory --- ScalaSTM ライブラリを用いてより直観的に扱える共有メモリ操作を学ぶ

*筑波大学ビジネスサイエンス系

- ch8 Actors --- これまでとは違うモデル。どのように並行性のための抽象を選ぶか/メッセージパッシングを使うやりかた
- ch9 Concurrency in Practice --- ここまでのさまざまなライブラリをまとめ、どのように適切なものを選ぶかを扱う

□ 順番に読むことを薦めるが、必要なところを拾って読むことも可

2 What you need for this book

2.1 Installing the JDK

□ (別にいいよね)

2.2 Installing and using SBT

□ やってみたけど、JAR を取って来て java コマンドで動かすだけ

- scalaVersion は設定しない方がいいような…
- いちいち package はソースに書かなかったとのこと
- 多くの例題ではプログラムを SBT と同じ JVM で動かしている (違う場合注記)

2.3 Using Eclips, IntelliJ IDEA, or another IDE

□ 略)

2.4 Who this book is for

□ Scala を使ってはいるが一般的なことが学べる

2.5 Conventions

□ (以下ずっと略)

3 1. Introduction

□ 「1CPU は限界で、これからはマルチ CPU だよ」(Gene Amdahl, 1967)

□ 並行プログラミングの歴史は長いが今日のマルチコアが牽引力

- 従来型の並行プログラミングだけでなく大きなパラダイムシフトも
- すべての開発者にとって必須のスキルに

□ 本章では:

- 並行プログラミングの簡単な紹介
- 並行プログラミングで Scala を使うと何がいいか
- 本書に必要な範囲の Scala 入門

3.1 Concurrent Programming

□ 並行プログラミング: プログラムは一群の並行計算の集まり

- 並行計算どうしは重なって実行され実行の協調が必要
- 並行プログラミングを正しくやるのは大変
- 直列プログラミングのすべての落とし穴はそのまま存続
- それに加えて多数の新たな落とし穴
- なんでそんなことするの?

□ 複数の利点

□ 性能が向上する「かもしれない」

- マルチコア CPU が普及したので、この理由から並行プログラミングに注文

□ I/O 操作が高速化する「かもしれない」

- 直列プログラミングではポーリングとか色々しかけが必要
- 並行プログラムならすぐに応答するようにできる
- このためマルチコア以前から並行プログラミングがあった (そうなの?)

□ 実装や保守性の面で利点がある「かもしれない」

- 並行単位に分けて書く方が簡潔に書けるプログラムもある
- UI、Web サーバ、ゲームエンジンなどはそういう例

□ 本書では共有メモリを介してやりとりする並行プログラムを扱う

- これと対照的にメモリを共有しない複数のコンピュータの並行性も→分散プログラミング
- 分散プログラミングでは他のノードがいつこけるか分からないことが前提になるのでやっかい ←前提次第では?
- 本書でもところどころで分散プログラミングに触れる箇所がある

3.2 A brief overview of traditional concurrency

□ コンピュータシステムにおける並行性 → HW レベル、OS レベル、◎言語レベル

- 並行システムにおける複数の実行の調整→同期 (synchronization)
- 同期には同期メカニズムが含まれる
- 同期方式 → 複数の実行主体がどのようにやりとりするか
- 共有メモリ型: 実行主体どうしはメモリを介し情報をやり取り
- メッセージパッシング型: 分散計算では実行主体はメッセージでやり取り

□ 一番下のレベルでは実行主体ハプロセス/スレッド → ch2

- プロセス間の同期はロックとモニタ
- 同期を通じて実行順を制御することであるスレッドがメモリに加えた変更が他のスレッドからはその後に見える

□ スレッドやロックで並行プログラムを記述するのはしばしば繁雑

- このため通信チャネル、並行コレクション、バリア、計数ラッチ等いろいろなものが生み出された
- これらを用いると特定の並行パターンが容易に記述できる → ch3

□ 伝統的な並行プログラミングは低レベルでさまざまな誤りを産みがち

- 例: デッドロック、窮乏、データ競合、競合条件
- Scala で並行プログラミングをする場合はこれらは使わないのが普通
- しかしこれらの低レベルの知識を持っておくことは価値がある

3.3 Modern concurrency paradigms

□ より新しい並行パラダイムでは「どうやって」の代わりに「何を達成」で記述する

- 実際には低レベル/高レベルん違いはあまり明確でないし、異なるフレームワークどうしの間も連続している
- ではあるが、新しいものは「宣言的」「関数型」の方向

□ 2 章で扱うが、並行して値を計算するにはスレッドを作り開始させ、終了を待ち合わせて値を受け取る必要

- やりたいのは「並列に計算して、値ができたら教えてね」
- よりよいのは、値はあるものと思って先に進み、使う時に使えること→ Future を使った非同期プログラミング
- さらにストリームを使うことで宣言的に並行性を扱うこともできる

□ 直列計算でも関数型スタイルは普及

- Python, Haskell, Ruby, Scala ではコレクションに対する操作を関数で表現可能だし、フィルタなども使える
- これらでは実装ではなく目標を記述していると言える
- 5 章ではデータ並列コレクションを示すが、これもこの一環

□ 高レベル並行のもう 1 つのトレンドは特定タスクに特化したフレームワーク

- STM はメモリトランザクションに特化し、並列計算の開始とかは扱わない
- メモリトランザクションはメモリの読み書きで「瞬時に起こる」と見なせるものを実現 → 7 章
- これにより低レベル並行のときに起きる問題を回避

□ 高レベル並行フレームワークの中には分散をサポートするものも

- データパラレル、メッセージパッシグがそう → 8 章

3.4 The advantage of Scala

□ Scala は発展途上だが Java のように広く使われつつある

- 並行プログラミングのサポートが充実
- 多様なあらゆるスタイルの並行フレームワークがある
- 並行プログラミングのための近代的で高レベルな API

□ Scala の構文的な柔軟さのため、フレームワークを入れやすい

- 1 級関数、名前パラメタ、型推論、パターンマッチなど→あたかも組み込み機能であるかのように API を使える
- これらの API は内部 DSL のように特定のプログラミングモデルを提供
- 組み込みのように見せられる機能の例: アクター、STM、Future
- しかし実際にはライブラリ。Scala は新言語を作る必要を無くす

- 他の言語のように構文で苦労しない→多くのユーザにとって魅力

□ Scala は安全な言語 --- GC、配列の境界検査、ポインタ演算無し

- 静的型検査→多くのエラーを実行前に検出
- 並行プログラミングでは多くの問題が産まれるので安全なほど良い

□ Scala は JVM 上の言語→インタオペラビリティ

- 既存の Java ライブラリが自由に参照可能
- Java からの移行が容易
- スレッドモデル、メモリモデルも JVM で規定→並行動作もポータブル
- Java 互換フレームワークでも Scala が開発言語として使われる

3.5 Preliminaries

□ (プログラミングとか OOP とかは知っているものとしますよ…)

3.6 Execution of a Scala program

□ 簡単なプログラムから

```
object SquareOf5 extends App {
  def square(x: Int): Int = x * x
  val s = square(5)
  println(s"Result: $s")
}
```

- これを実行するときのスタックとヒープのようすを図 (p18) に示す
- 以下、そのほかのさまざまな機能

3.7 A Scala primer

□ 本書を読む上で必要な Scala の概観（詳細な入門ではない)

□ あいさつメッセージを出すクラス

```
class Printer(val greeting: String) {
  def printMessage(): Unit = println(greeting + "!")
  def printNumber(x: Int): Unit = {
    println("Number: " + x)
  }
}
```

- printMessage は引数 0 個、1 つの文のみを持つ
- printNumber は引数 1 個 (型 Int)
- どちらも値を返さない (→ Unit)。Unit は省略してよい

□ 前記のをインスタンス生成して動かすには:

```
object Test2 extends App {
  val printy = new Printer("Hi")
  printy.printMessage()
  printy.printNumber(5)
}
```

□ Scala では単一のオブジェクトを生成できる

```
object Test {
  val Pi = 3.14
}
```

□ Scala では trait を extends したクラスが作れる

- trait はインタフェース、値、メソッド実装を含んだもの

```
trait Logging {
  def log(s: String): Unit
  def warn(s: String) = log("WARN: " + s)
  def error(s: String) = log("ERROR: " + s)
}
class printLogging extends Logging {
  def log(s: String) = println(s)
}
```

□ クラスは型パラメタを持てる

```
class Pair[P, Q](val first: P, val second: Q)
```

□ 1 級関数オブジェクト (lambda)

```
val twice: Int => Int = (x: Int) => x * 2
```

□ 「=>」は引数と返値の間にないといけない。型は次のように省略可

```
val twice = (x: Int) => x * 2
```

□ 次のように宣言側を書いて定義側を省略も可

```
val twice: Int => Int = x => x * 2
```

□ 引数が lambda の本体内で 1 回しか現れないなら次のようにできる

```
val twice: Int => Int = _ * 2
```

□ 1 級関数オブジェクトを使うことでコードのかたまりを値のように扱える。次の例では名前パラメタを使っている

```
def runTwice(body: =>Unit) = {
  body
  body
}
```

□ 名前パラメタは型の前に「=>」を付けることで指定できる。使う例

```
runTwice {
  println("Hello")
}
```

□ for 式はコレクションの辿り/変換に便利

```
for(i <- 0 until 10) println(i)
```

□ 「0 until 10」は「0.until(10)」と同等 (オブジェクトのメソッド呼び出しで「.」を省略できるところがある)

□ for は常に foreach と同等

```
(0 until 10).foreach(i => println(i))
```

□ for-comprehension でデータの変換ができる

```
val negatives = for(i <- 0 until 10) yields -i
```

□ これは次と同等

```
val negative = (0 until 10).map(i => -1*i)
```

□ 複数の入力からマップすることもできる

```
val pairs =
    for(x <- 0 until 4; y <- 0 until 4) yield(x, y)
```

□ これは次と同等

```
val pairs =
     (0 until 4).flatmap(x =>
       (0 until 4).map(y => (x, y)))
```

□ このように for-comprehension では何重でもネスト可

□ よく使うコレクションの例として Seq[T], Map[T], Set[T]

```
val mesg: Seq[String] = Seq("Hello","world", "!")
```

□ 文字列埋め込み聞こう「s"..."」

```
val magic = 7
val mynum = s"My Magic No. is $magic"
```

□ パターンマッチも重要な機能

□ 次の例ではマップから値を取り出し、有無をチェック

```
val sucs = Map(1->2, 2->3, 3->4)
sucs.get(5) match {
  case Some(n) => println(s"suc is: $n)
  case None    => println("not found")
```

□ 大部分のオペレータはオーバーロード (再定義) 可能

```
class Position(val x:Int, val y:Int) {
  def +(that: Position) =
    new Position(x + that.x, y + that.y)
}
```

□ パッケージオブジェクトを定義してトップレベルのメソッドや値定義を格納できる

```
package org
package object learningconcurrency {
  def log(msg: String): Unit =
    println(s"${Thread.currentThread.getNmae}: $msg")
```

4 Summary

□ 並行プログラミングとは何か

□ Scala はなぜよいか

5 Exercises

□ 少しはやってみましょうね?