Learning Concurrent Programming in Scala

第3章 前半

久野研究室 2015年 輪読会

斎藤 祐一郎

留意事項

Code/Slide: github.com/koemu/LCPiS

```
$ brew info sbt
sbt: stable 0.13.9 (bottled)

$ scala -version
Scala code runner version 2.11.7 -- Copyright 2002-2013, LAMP/EPFL

$ java -version
java version "1.7.0_04"
Java(TM) SE Runtime Environment (build 1.7.0_04-b21)
Java HotSpot(TM) 64-Bit Server VM (build 23.0-b21, mixed mode)

$ sysctl -n machdep.cpu.brand_string
Intel(R) Core(TM) i7-4578U CPU @ 3.00GHz
(2cores 4threads)
```

3章: Traditional Building Blocks of Concurrency

- ※冒頭の一節は、C++の大家であるビャーネ・ストロヴストルップ先生の言葉です。
- 2章でJVM上の並列プログラミングの基本について確認した。
- しかし、低レベルな部分ではエラーが出やすかったりデリケートだったりする。
  - データ競合、再整列、可視性、デッドロック、非決定性...
- 幸運なことに、私たちには典型的な並列プログラミングに関するより応用的な積み木(ライブラリ)がある！
- その使い方について学ぶ。

- 一般的に、並列プログラミングには2つの側面がある。
  - 並列プログラミングで実装してみたい。
  - 並列でデータにアクセスしたい。
- 2章で学んだように、並列プログラムはスレッドごとに定義し開始することができる。本章では、より軽量に実装する方法を学ぶ。
- 2章で `synchronized` ステートメント、揮発性のある変数を用いるのは確認した。本章では、より詳細に迫る。

- (続き)
  - `Executor`,`ExecutionContext` オブジェクトの利用。
  - 原始性のある単純なノンブロッキング同期
  - 並列中の遅延変数とのやりとり
  - 並列キュー、セット、マップ。
  - プロセスの作成と通信のしかた。
- 究極の目標は、並列でも安全にファイルをハンドリングするプログラムをインプリすること。本章では再利用可能なファイルハンドリングAPIをつくる。

The Executor and ExecutionContext objects

- 2章で、スレッド生成はプロセス生成よりも計算負荷の小さいこと、オブジェクトを確保するよりも大きいこと、モニターロックの獲得、エントリの中のコレクション更新について議論した。
  - アプリケーションは数多くの小さな並列タスクと高いスループットを求められる。
  - 全てのタスクにおいてフレッシュなスレッドを作ることはできない
- スレッドの起動には、メモリのアサイン、コンテキストスイッチが求められる。
- そこで **スレッドプール** : 多くの並列プログラミングフレームワークには、あらかじめスレッドを起動して、必要に応じてタスクを割り当てられるようになっている。

- プログラマは、並列タスクの実行内容をどのようにするかカプセル化できる。
  - JDKでは抽象的に Executor .ExecutorRunnable というシンプルなインタフェイスがある。1つのexecuteメソッドを割り当てられる。
  - Runnableオブジェクトを用意し、run()メソッドが呼ばれるようになっている。
- Executorオブジェクトは定義されたexecuteかRunnableオブジェクトを呼び出し元のスレッドから開始する。これらはスレッドプールから呼び出される。
- JDK7にはForkJoinPoolがある。ScalaではJDK6でも使える。
  - コード解説。(p65)
  - SBTのfork設定をfalseにするとsleep(500)のコードは不要とのこと。
  - 私の環境だと設定無しにsleepなしでもちゃんと動きました。皆様検証ください。

- なんでExecutorを最初に持ってこないと行けないか？先のサンプルでは簡単にExecutorの実装をRunnableの変更無しに変えることが可能。なぜなら切り離されて動いているから。
- このあたり、焦点絞って動きをより探ってみる。
  - ForkJoinPoolクラスがExecutorServiceを呼び出す。こいつは便利なメソッドがいくつか用意されていて、shutdownが最重要。
  - shutdownメソッドは、Graceful shutdownができる(ミドルウェアには重要ですこれ)。
  - awaitTerminationメソッドを定義すると、スレッド終了までの待機時間を定義可能。
  - コード解説。(p65_2)

- scala.concurrentパッケージのExecutionContextはExecutorオブジェクトと同じ、またはそれ以上の機能がある。
  - オブジェクトのコンテキストの抽象executorメソッドはExecutorのそれと同じ。更にreportFailureメソッドがあり、例外を投げられる。
  - デフォルトコンテキストであるglobalコンテキストは、内部でForkJoinPoolインスタンスを持っている。
  - コード解説。(p66)

- ForkJoinPoolで生成したスレッドプールをExecutionContextで使うこともできる。
  - コード解説。(p67)
- executeをブロックコードでも書ける。
  - コード解説。(p67_2)
- ExecutorやExecutionContextは素晴らしい並列プログラミングの抽象部分を持っているが、問題が無い訳ではない。
  - スループットを確保するためにスレッドを再利用する機構が仇となることがある。
  - コード解説。(p67_3) ※秒を変えて実行

※皆さんの環境でもお試しください。マシンスペックにより結果が変わるはずです。

```
$ sysctl -n machdep.cpu.brand_string
Intel(R) Core(TM) i7-4578U CPU @ 3.00GHz

$ sbt run
[info] Set current project to p67_3 (in build file:/Users/saito/repos/LCPiS/p67_3/)
[info] Running ExecutionContextSleep
ForkJoinPool-1-worker-1: Task 2 completed.
ForkJoinPool-1-worker-5: Task 0 completed.
ForkJoinPool-1-worker-7: Task 3 completed.
ForkJoinPool-1-worker-3: Task 1 completed.
ForkJoinPool-1-worker-1: Task 4 completed.
ForkJoinPool-1-worker-7: Task 6 completed.
ForkJoinPool-1-worker-3: Task 7 completed.
ForkJoinPool-1-worker-5: Task 5 completed.
ForkJoinPool-1-worker-3: Task 10 completed.
ForkJoinPool-1-worker-5: Task 11 completed.
ForkJoinPool-1-worker-1: Task 8 completed.
ForkJoinPool-1-worker-7: Task 9 completed.
ForkJoinPool-1-worker-3: Task 12 completed.
ForkJoinPool-1-worker-1: Task 14 completed.
ForkJoinPool-1-worker-7: Task 15 completed.
ForkJoinPool-1-worker-5: Task 13 completed.
ForkJoinPool-1-worker-5: Task 19 completed.
ForkJoinPool-1-worker-3: Task 16 completed.
[success] Total time: 11 s, completed 2015/09/24 0:16:45
```

- 全てのスレッドが2秒で終わるはずだが、そうはならない。
  - 4cores 8threadsのCPUだと、`ExecutionContext`はスレッド数分の8スレッドをスレッドプールに用意する。
  - あらかじめ用意されたスレッド数以上を実行すると、ブロックイディオムにガードされる(2章参照)。`notify`が呼ばれると改めて立ち上がる。
  - 最初の2秒で同時に8つ、次の2秒で次の8つ...とやっていくと、8秒強かかっておわる。
  - 10秒経つと、親スレッドが終了する。
  - (そう、32threadsのマシンなら2秒で終わる！7threads未満だと32回実行できません。)
- 解放されるまで永遠に実行がブロックされる。このような状況を **スタベーション** という。

Atomic primitives

- 2章で、適切な同期が適用されない限り、(共有)メモリへの書き込みは直ちに反映されないことを学んだ。
  - 原始性
- 先行発生の関係が担保されることにより可視性が確認できた上で、`synchronized`ステートメントが信頼を得る。(訳？)
- Volatileフィールドはより軽量な先行発生の関係の担保方法だが、同期構造は弱い。
- `getUniqueId`メソッドを正しく実装するには？(訳？)
- 本章では、複数箇所からの読み書き可能なAtomic変数について学ぶ。
  - Atomic変数はVolatile変数の兄弟分みたいなものだが、より豊か(強力)。
  - より複雑な並列操作を`synchronized`ステートメントの信頼無しに実行可能。

Atomic variables

- **Atomic変数** とは、Complex Linearizable Operationを可能になした記憶域である。
- **Linearizable Operation** とは、システム内においてあらゆる同時操作ができるものである。
  - 例: Volatileな書き込み。
- **Complex Linearizable Operation** とは、少なくとも同時2並列の読み書きが発生するLinearizable Operationと同義。
  - ここでの原始性は、Complex Linearizable Operationを指す。
- 各種のAtomic変数は`java.util.concurrent.atomic`パッケージに定義され、Complex Linearizable OperationはBoolean, integer, long, refrence型でサポートされている。
- 2章で出て来た`getUniqueId`を、`AtomicLong`を使って再実装してみる。
  - コード参照。(p69)

- Atomic変数は、getAndSetメソッドとして、別の実装方法もある。数値型だとdecrementAndSetもいる。
- compareAndSetは、基本的なAtomic変数の実装である。CASと呼ばれることもある。
  - 教科書 pp.70のコード参照。
  - ロックフリーである。
- CASで書き直したコードを見てみる。
  - コード参照。 (p70)
  - 教科書 pp.71のフロー図参照。
  - Tail recursion をしてスタック溢れを防止している点に注目。

Lock-free programming

- **ロック** とは同期処理の一つであり、複数スレッドからのアクセスを制限するものである。
- 2章ではJVMの本質的なロック機構であるsynchronizedステートメントを利用した方式を学んだ。これは、不要になるまで全てのスレッドからのアクセスをロックする。本章では別の例を学ぶ。
- ロックはデッドロックを許してしまう。OSのプリエンプションがロックを保持していると、他のスレッドの実行が遅れてしまう。そこで、ロック・フリープログラムなら、パフォーマンスの妥協の影響を抑えることができる。
- Atomic変数が必要な理由は、 **ロック・フリー操作** を行えるからである。ロック・フリー操作にはいかなるロックの獲得も行っていない。多くのロック・フリーアルゴリズムは、首尾一貫したスループットを獲得できる。

- ロック・フリーアルゴリズムはOSからのプリエンプションがあったときにロックを保持していない時、他のスレッドからの一時的なブロックは行えない。
- さらに、ロックフリー操作は、ロック無しに不定なブロックを得ないため、デッドロックを受け付けない。
- ロック・フリー操作をもとにしたgetUniqueIdの実装を見てみる。
  - 他のスレッドから永続的な中断のためにロックを必要としない。
  - もしある1つのスレッドがCASの操作に失敗しても、随時再起動しgetUniqueIdを再試行する。
  - コード参照。(p72)

- 2章で、モダンなOSではプリエンプティブなマルチタスキングを用いていることを学んだ。(pp.39)
  - ロック状態だと、他のスレッドもロックが解除されるまで待機状態。
- ロック・フリー操作なら、足を引っ張るスレッドによって他のスレッドの実行がブロックされることは無い。
  - 並列実行される場合でも、それぞれのスレッドは必ず決まった時間に処理が終わる。
- ロック・フリーダムについてより形式的な定義では、ロック・フリー操作は、実装がしんどいと感じるはず。より複雑な実装だと周知のごとく難しい問題である。
- CASベースの`getUniqueId`の実装は実際にロック・フリー操作である。CASの操作に失敗→別の視点から見ると、他のスレッドは実行に成功している。この事実はロック・フリーダムを実証している。

Implementing locks explicitly

- 呼び出しもとのブロックを行わずとも、Atomic変数のロックを取りたい時がある。
  - この2章で登場した固有のオブジェクトロックの問題は、スレッドがロックを獲得できるか確認できない。
  - `synchronized`を直ちに呼び出してブロック。
  - ロックが無いときに違ったアクションを起こしたい時がある。
- 本章では、並列ファイルシステムAPIメソッドを作る。
  - 古き良きDOSのファイルマネージャとか...
- モダンなファイルマネージャは同時にファイル転送ができたり、転送をキャンセルできたり、削除を同時に行えたりできる。

- 今回のAPIで担保が必須となるのは
  - ファイル生成時に、コピーや削除ができない。
  - ファイルコピー時に、ファイルは削除ができない。
  - ファイル削除時に、ファイルはコピーできない。
  - 削除はシングルスレッド。
- `Entry`クラス: 1つのファイル・ディレクトリの情報
  - `isDir: true`ならディレクトリ
  - `state`: 操作の状態(Idle, Creating, Copying, Deleting)
  - Copyingには並列数の情報がある
  - pp.75の図も参照

- それではファイルを削除してみる。
  - コード参照。(p74)