

Learning Concurrent Programming in Scala

第5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 "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.66-b17, mixed mode)
```

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

5章: Data-Parallel Collections

- ※冒頭の一節は、TeXの生みの親でもあるドナルド・クヌース先生という言葉です。ついつい時期尚早な最適化ってやってしまいがちですよ～
- 振り返り
 - ここまで、マルチスレッドを利用した並列プログラミングについて学習した。
 - そうすることで、正確さを保証できることに焦点を当ててきた。
 - 並行プログラミングからブロッキングをなくし、非同期な計算を、そして並列したデータ構造をスレッド間で通信するのかを理解した。
 - ツールを利用すると楽にできる。

- 本章では
 - 良いパフォーマンスをもたらす方法について焦点を当てる。
 - 既に存在するプログラムをあまり変えることなく、処理時間を短くする方法を学ぶ。
 - 前の章の内容でもできんことは無いけど、比較的重たい・非効率な時にやれる。
- **Data Palarellism** とは異なったデータエレメントを持つ同一の計算処理を続ける方法。
- 同期をもちいた並列計算処理よりも、並行データ処理、ゆくゆくはマージする。(?)
- 並行なデータを入力(多くはデータセット)し、また違ったデータセットを出力する。

- 本章は次のトピックを取り上げる
 - データ並行の操作
 - 並行度の設定
 - パフォーマンスの計測とその重要性
 - 直列処理と並行処理の違い
 - 並行コレクションと並列コレクション
 - カスタム並行コレクションの実装
 - その他 データ並行フレームワークについて

Scala collections in a nutshell

- Scalaのコレクションモジュールは標準ライブラリにパッケージングされており、一般的な利用に向けたコレクションタイプにまとめられている。
- 関数を組み合わせることで、一般的かつ簡単に宣言的に操作ができる。
- P.138の真ん中の例では、`filter`コンビネータを使って0～10万字の反対になった文字列を並べることができる。
- 3つの基本コレクションタイプ: `sequences`, `maps`, `sets`
- `sequences`は、`apply`メソッドを用いると、インデックスを検索することができる。
- `maps`は、Key-Value形式で、Keyに基づき値を検索できる。
- `sets`は`apply`メソッドを用いることで、エレメントのメンバーを検索できる。

- Immutable CollectionとMutable Collectionがある。前者は生成後は変更不可、後者は変更可能である。
 - 前者: Vector, ArrayBuffer
 - 後者: HashMap, HashSet
- par メソッド: データを並行に処理する
 - P.138 末尾の例を参照
- この後詳しく学んで行く。

Using parallel collections

- 多くの並列プログラミングユーティリティでは、他のスレッドとのデータのやり取りができた。
- アトミック変数、同期ステートメント、コンカレントキューなど、並列プログラムが正確さを保証していた。
- 並行コレクションプログラミングモデルでは、直列のScalaコレクションとして一致した大きなものとしてデザインされている。
- 並行コレクションは、それ1つだけで実行時間を改善できる。
- 本節では、並行コレクションを使った時の実行時間差を見てみる。
- この後の共通ライブラリとなるコード解説 (P.139 中央)
 - JVMによって最適化がかかるため、その影響を排除するため、bodyブロックを呼び出すときにその影響を排除するよう計算する。

- プログラムのパフォーマンスの要因はいろいろあり、予測も大変。そこで仮説を評価して行く。
- コード解説 (p139_1)
 - Vectorクラスに5百万件のデータを放り込む
 - そんでもってシャッフル
 - 2つの方法で最大値を探す
 - 実行結果例は次のページに

- 準備していて困ったことが発生
 - 教科書では並行の方が速いのだが、私の手元だと直列の方が速い。

```
$ 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)
```

```
$ sbt run
[info] Set current project to p139_1 (in build file:/Users/saito/repos/LCPiS/src/p139_1/)
[info] Running ParBasic
largest number 4999999
run-main-0: Sequential time 109.569 ms
largest number 4999999
run-main-0: Parallel time 681.545 ms
[success] Total time: 5 s, completed 2015/10/29 19:57:21
```

- ということでJava 8に変えました。
- JVMの作りによって動きが違う最たる例ですね。
- (この後は特記が無い限りJava 8)

```
$ java -version
java version "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.66-b17, mixed mode)
```

```
$ sbt run
[info] Set current project to p139_1 (in build file:/Users/saito/repos/LCPiS/src/p139_1/)
[info] Compiling 2 Scala sources to /Users/saito/repos/LCPiS/src/p139_1/target/scala-2.10/classes...
[info] Running ParBasic
largest number 4999999
run-main-0: Sequential time 294.003 ms
largest number 4999999
run-main-0: Parallel time 251.869 ms
```

- 参考: 別の環境での実行結果
 - わずかに並行の方が速い。っていうかなぜCore i5の方が速い...?

```
$ 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_60"
Java(TM) SE Runtime Environment (build 1.7.0_60-b19)
Java HotSpot(TM) 64-Bit Server VM (build 24.60-b09, mixed mode)
$ sysctl -n machdep.cpu.brand_string
Intel(R) Core(TM) i5-4250U CPU @ 1.30GHz

$ sbt run
[info] Set current project to p139_1 (in build file:/Users/koemu/repos/LCPiS/src/p139_1/)
[info] Running ParBasic
largest number 4999999
run-main-0: Sequential time 128.808 ms
largest number 4999999
run-main-0: Parallel time 114.129 ms
[success] Total time: 5 s, completed 2015/10/29 20:29:08
```

- maxメソッドは並行処理にもってこい。
- **trivially parallelizable**: 今回のmaxメソッドの動きで例えると...
- 各ワーカーが最大値を探す
- 最後に全てのワーカーの最大値の最大値を探して答えを出す

- 一般的に、データの並行操作は、maxメソッドに比べプロセッサ間の通信をより求められる。
- incrementAndGetメソッドについて改めて考えてみる。(3章で出てきてます)
- コード解説 (p140_1)
 - 並行処理を行っているのはforeachを呼び出しているのと同じ。
 - 並行処理している方は、それぞれのスレッドの動きを待っているなので結果遅くなる。

```
$ sbt run
[info] Set current project to p140_1 (in build file:/Users/saito/repos/LCPiS/src/p140_1/)
[info] Running ParUId
run-main-0: Sequential time 201.927 ms
run-main-0: Parallel time 397.328 ms
[success] Total time: 1 s, completed 2015/10/29 23:20:08
```

- マルチコアでさばけるからと言って万事速いという訳ではない。
 - 今の例の通り。
- 現代のCPUは、RAMに直接アクセスしている訳ではなく、L1, L2等のキャッシュに情報を貯めている。
 - コアが分かれて処理されると、RAMを通さずともキャッシュもその処理が所属するコアに分かれてコピーされ、同期が取られる。これをMESI(別名Illinois protocol)という。
- 処理の流れはP.142の図を基に解説

- 並行処理プログラムは、他のリソースも共有するため、計算機性能をより求められる。
- **resource contention:** 資源の競合
- **memory contention:** メモリの使用量の競合
- パフォーマンスを劣化させる要因は他にもある
 - 同じオブジェクトに対する同期処理
 - 並列処理可能なマップに対する同一キーへの変更
 - コンカレントキューへ同時にキューイング
- これらは全て同じメモリ領域にアクセスする
- いくつかのアプリでは非定期的に並列に書き込みを行う

Parallel collection class hierarchy

- これまで並列実行した別のワーカーが同時実行していることを学んだ。
- uidの保存の競合については第2章で既に学んでいる。
- この成り行きとして、並行コレクションは直列コレクションのサブタイプではないことがわかる。
 - これはリスコフの置換原則に違反する。
- リスコフの置換原則: SがTのサブタイプだった場合、TはSにプログラムの正当性の評価無しに置換できる。

- もし、並行コレクションが直列コレクションのサブタイプだったとすると、それぞれのメソッドは直列のコレクションを返す。
- クライアントはforeachが存在すると知らずに同期可能なメソッドを呼び出せるようにした場合場合、正確に動かない。
- P.143のダイアグラムに、ヒエラルキーを示した。
- (あー、この後良くわからん。)

Configuring the parallelism level

- 並行コレクションは、デフォルトでは全てのプロセッサを使う。その下には、たくさんのワーカーが動いている。
- TaskSupportオブジェクトを編集してデフォルトの動きを変えてみる。
- ForkJoinTaskSupportはTaskSupportの典型的な実装である。
- コード参照 (p145_1)
 - 2スレッドに制限されています
 - 2から変更すると実行速度が変わりますので時間があれば試します

Measuring the performance on the JVM

- JVM上のプログラムの実行時間の計算は簡単じゃない。
 - 表面上、わからないことがある。
- Scalaは実行可能形式のバイナリはコンパイルできない。Javaのバイトコードを出力する。
- **interpreted mode**: Javaのバイトコードなら、逐次実行可能。
- JIT: バイトコードから、マシン語にコンパイル。
- 何でバイトコード: 様々なプラットフォームで動くようにしたいから。でも遅い。
- **steady state**: 起動してしばらくすると最高のパフォーマンスに達する、その時の状態。

- 影響を参考にするために、<TEXTAREA>タグを探すHTMLパーサーを書いてみる。
- コード参照 (p146_1)
 - 私のコードは最後にThread.sleep(10000)がありますが、これは入れないと速く終わりすぎてしまうので入れました。
 - めっちゃエラッタ投げたい気分。30分消費した。

- 5回実行目
 - なんでSequentialの方が速いんだ泣きたい

```
$ sbt run
[info] Set current project to p146_1 (in build file:/Users/saito/repos/LCPiS/src/p146_1/)
[info] Compiling 1 Scala source to /Users/saito/repos/LCPiS/src/p146_1/target/scala-2.10/classes...
[info] Running ParHtmlSpecSearch
ForkJoinPool-1-worker-5: Sequential time 13.739 ms
ForkJoinPool-1-worker-5: Parallel time 24.503 ms
[success] Total time: 14 s, completed 2015/10/30 0:10:43
```

- warmedTimed()を入れると、steady stateになってから実行できるので結果が安定する。
- コード参照 (p146_2)
 - ここだと並行の方が速い
 - JVMの起動オーバーヘッドが結構効いてた？

```
$ sbt run
[info] Set current project to p146_2 (in build file:/Users/saito/repos/LCPiS/src/p146_2/)
[info] Running ParHtmlSpecSearch
ForkJoinPool-1-worker-5: Sequential time 1.665 ms
ForkJoinPool-1-worker-5: Parallel time 1.503 ms
[success] Total time: 11 s, completed 2015/10/30 0:16:18
```


- まあ、ここまで来るとわかりますが、JVMのパフォーマンスを測定するのはなかなか大変。
- **JIT**コンパイラがどこかで実行を止めてコードを最適化するにあたって、逆に遅くしている。
- **GC**があることで、deleteステートメントは無い。使わなくなったオブジェクトを自動的に回収して破棄してくれる。
- GCが動くときに、また計測結果がぶれる。
- 9章でより精度の高い分析方法をやりますが、それまでには長い道のりがある...