Learning Concurrent Programming in Scala

第7章 前半

久野研究室 2015年 輪読会

斎藤 祐一郎

7章: Data-Parallel Collections

- ※冒頭の一節は、C++の権威であるハーブ・サター氏の言葉です。
- 振り返り
  - 2章にある並行に関する基本的原則をたどってきた。
  - 共有アクセスが行われるプログラムの箇所を守る必要があることを理解した。
  - `synchronized` ステートメントは保護を行う基本的な方法である。シングルスレッドで実行するよう、個別のロックを担保できる。
  - これの欠点として、デッドロックがある。

- 本章では STM(Software Transactional Memory)について学ぶ。
  - デッドロックや競合に対する共有メモリに対する並行処理の制御である。
  - STMはコードの重要な部分を手掛ける。
  - STMは共有メモリの読み書きの追跡、および直列化を行う。
  - STMを使うのは簡単！
  - スケーラビリティも可用性も高い。

- メモリトランザクションは、データベースのトランザクションから来ている。
  - クエリ実行プロセスの隔離を担保する。
  - 1回で、論理的に共有メモリを読み書きできる。
  - **isolation**: 各々の実行を隔離するさま。
- **composability**: STMのもう一つの優位性
  - ロックベースで操作するハッシュテーブルベースのデータの追加・削除を考える。
    - これは同時に削除しながら追加できない。

- 伝統的に、STMはコンパイル時にトランザクションの制限をかけられるよう、プログラミング言語の一部として提案されてきた。
  - 実装は言語によって違う。
- 今回はScalaSTMについて取り上げる。
  - アトミック変数の欠点
  - STMのセマンティックと内部構造
  - トランクショナルリファレンス
  - トランザクションと外部に対する効果の間のインタラクション
  - 単一操作のトランザクション、ネストされたトランザクションについてのセマンティック
  - トランザクションの再試行及びタイムアウト
  - ローカル変数・配列・マップに関するトランザクション

The trouble with atomic variables

- 第3章でatomic variablesをやった。
  - 読み書きの整合性を担保。
  - デッドロックのリスクを低減。
  - それでもなお、atomic variablesでは充分ではない状況がある。
- 6章で、Rxフレームワークを使ったWebブラウザを実装した。
  - いくつかの追加機能を入れたい。
  - 履歴: `List[String]` に入れればいい
  - 全てのURLの一覧

- それでは、synchronizedからatomic variablesを使って実装し直してみる。
- コード参照 (p209_1)
  - 何回か実行するとArrayIndexOutOfBoundsExceptionが発生する。
  - append()とaddAndGet()がずれる可能性がある。
  - これじゃatomicじゃない。
- **linearizable**: 1つの直線にまとめるように
- **linearlization point**: linearizableするポイント

- atomic variables以外ではsynchronizedステートメントを使うのが簡単。
  - ネストもできる。
- ただしジレンマを抱える
  - 巨大なプログラムだと競合を抱える
  - デッドロックの問題を抱える
- STMならこれらの問題は2つの世界にとっても良い
  - 2つのジレンマから解放される

Using Software Transactional Memory

- 歴史的に、ScalaとJVMプラットフォームのSTMがある。
  - ここではScalaSTMについて学ぶ。
  - 2つの理由
    - ScalaSTMはSTMのエキスパートにより開発され標準化されている。
    - 複数のSTMの実装に対応している。
- 標準化されているSTM APIを使っていれば、別の実装に乗り換えるのはスムーズ。

- atomic ステートメントは全てのSTMにおける基本的な枠組み。
  - **memory transaction**: 囲った所がatomicかつ直列に読み書きされるようになる
- `synchronized`に似ているが、次の点に着目。
  - 独立性を担保
  - 処理の衝突の阻止
  - 競合の防止
  - デッドロックが発生しない

- P.213中央のコード参照
  - `swap()`と`inc()`の解説
- こうしてSTMを組み込める
  - 同時に2つのスレッドが同じ場所を書き換えることは無くなる。
  - STMのステートメントが読み書きの操作を記録している。
  - **commited**: atomic ステートメントが終わった(DBのコミットと似ている)
  - **transactional conflict**: 実行中に他のスレッドがアクセスしたとき。衝突した場合、キャンセルするか再実行する。
  - **roll back**: 衝突時に、ステートメントを反映しない。
  - **optimistic**: これらのオペレーションの総称。
  - **completed**: commitedか、rollbackして再実行できたとき。

- P.214の図を参照。
    - 図を見ながら流れを説明します
- 深くは掘り下げない。本のスコープから出てしまう。
- 使い方について焦点を当てて行く。

- ScalaSTMではマークされた部分だけ追いかける
  - atimicステートメントで囲われていない所の処理は保証されない
  - JVMのSTMフレームワークが読み書きを正確にキャプチャするためにポストコンパイルかバイトコードを自己利用する必要がある。
  - ScalaSTMはライブラリのみのSTM実装であり、コンパイル時に分析することはできない。
- トランザクショナルリファレンスの呼び方は次で学ぶ。

Transactional references

- **transactional references**: ある1点のメモリ域に対してトランザクショナルな読み書きを可能にする。
- ScalaSTMでは、`TをRef[T]`にカプセル化する。
- (使い方は省略)
- インスタンス化には`Ref.apply`ファクトリーメソッドを利用する。

- ここで、ブラウザの履歴をtransactional memoryを使って書き直してみる。
- `apply` メソッドを呼ぶとトランザクショナルリファレンスがかえってくる。
  - updateメソッドで内容を書き換えられる。
  - これらはトランザクションの外からは呼び出せない。
  - ScalaSTMのセーフティメカニズムである。

Using the atomic statement

- (前の節でおかしくなってしまった)urlsとclen変数について、トランザクショナルリファレンスにして実装し直す。
  - addUrlをatomicステートメントとして分離する。
- getUrlArrayを再実装する。
  - 先ほどと同様にatomicステートメントとして分離する。
- コード参照 (p217_1)