ここから始める
並行プログラミング

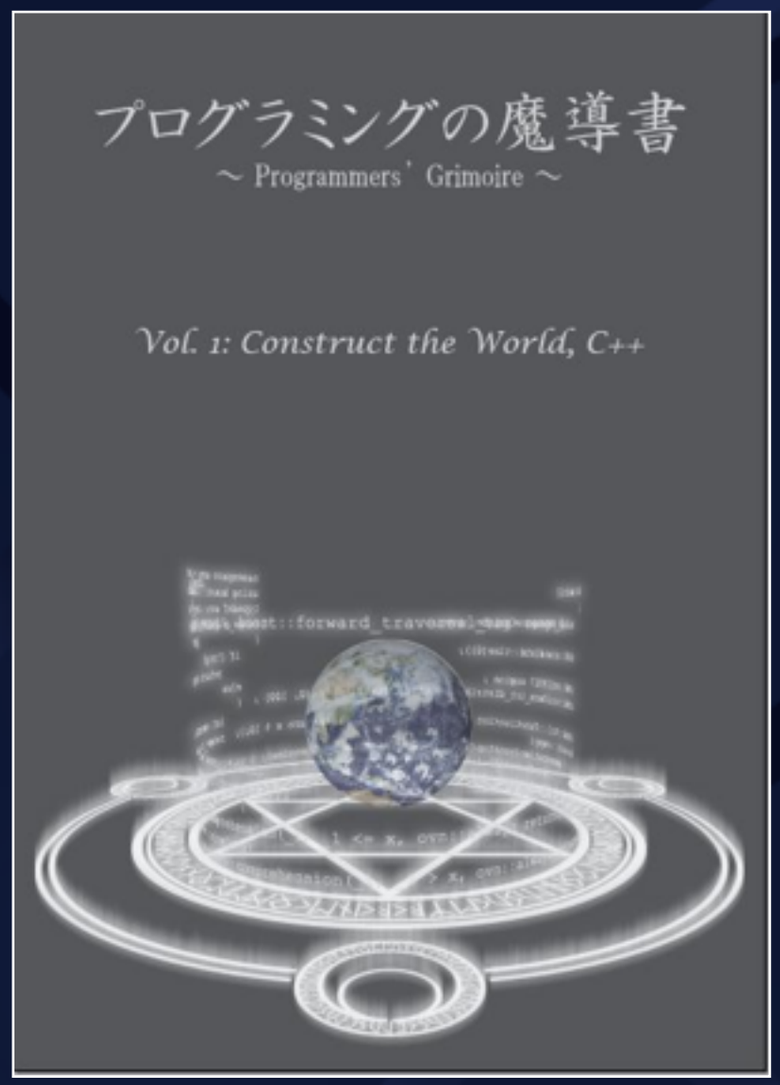
株式会社ロングゲート
高橋 晶(Akira Takahashi)
faithandbrave@gmail.com
2014/05/14(Wed) GREE Tech Talk #05

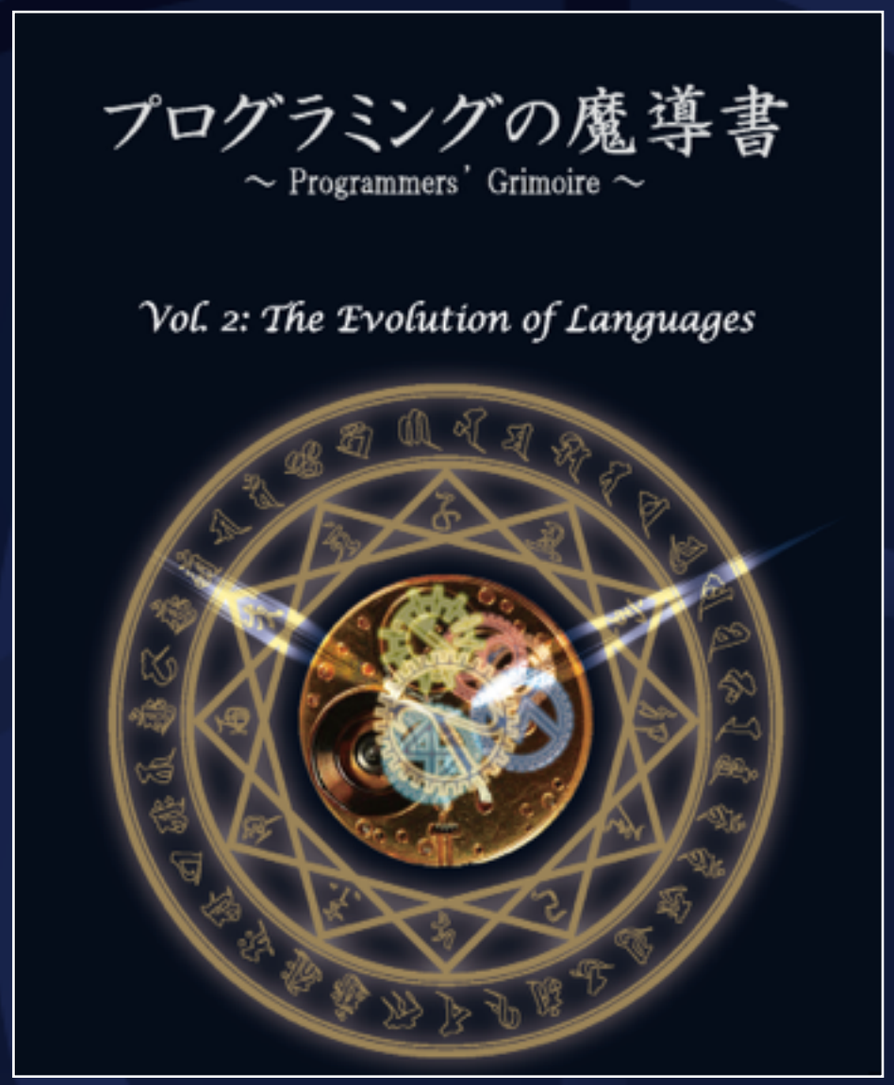
自己紹介

- 高橋 晶 (Akira Takahashi)
- 普段はC++をメインに使っていますが、
  言語マニアなのでScala、Haskell、Rubyなど、いろいろ使います。
- 『**プログラミングの魔導書**』 編集長
- Boost.勉強会 東京の主催者
- 著書：
  『C++テンプレートテクニック』
  『C++ポケットリファレンス』

プログラミングの魔導書

- 弊社ロングゲートで出版している、雑誌ライクな書籍。
- プログラミングの中・上級者向けの記事を提供している。





Vol.3のテーマは並行・並列・分散

- 2014年1月に出版したVol.3では、並行・並列・分散をテーマにしました。
- このテーマで深い領域に飛び込みたい方は、ぜひ読んでみてください。

並行プログラミングは難しい

- 難しい理由は、スレッドを使ったプログラムが「**非決定的である(non deterministic)**」という特性を持っていることです。
- 原理と原則を正しく理解して書かないと、数週間に一度しか起きないバグに悩まされ、痛い目に会います。
- どんなことを理解すれば、正しいプログラムを書けるのかを学んでみましょう。

並行と並列

- 複数データ、複数タスクを同時に扱う手法として、
  並行と並列という、2つの分野がある。
  どちらも、マルチスレッドや複数のCPUコア、複数の
  GPUコアを使用して同時処理を行う。

- **並行(Concurrent)**は、複数タスクを同時に処理する
  際に、タスク間の調停を必要とする計算分野。

- **並列(Parallel)**は、複数の計算資源を使用することで、
  計算を高速化する分野。

主な使用目的

- **関心の分離**として使う
  - 関係しない処理同士、もしくは機能性の異なる領域の処理を、別々に実行する。
  - アプリケーションがブロッキングしないよう、**バックグラウンドで処理を動かす**のも、この「関心の分離」に含まれる。

主な使用目的

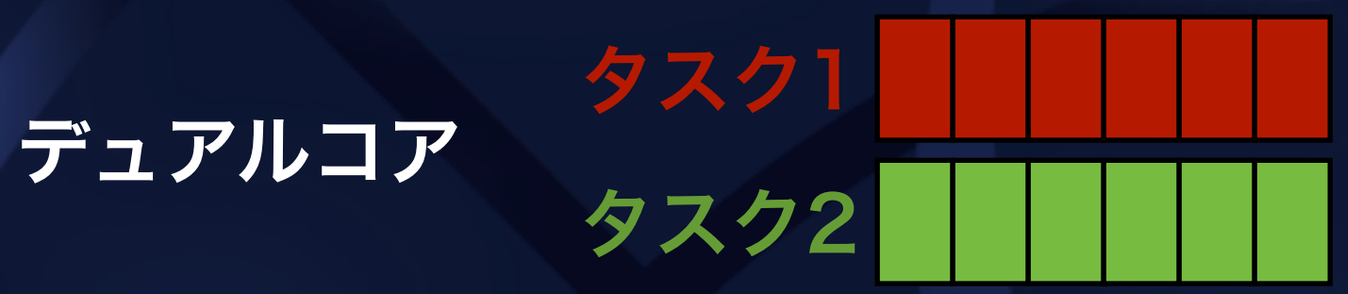
- **パフォーマンス**のために使う
  - 大量のデータ、大量のタスクを、複数同時に処理する。
  - 複数コアを持つCPUであれば、コンテキストスイッチのコストなく、効率的に複数同時処理を行える。

コンテキストスイッチ

- シングルコアCPUの場合を考える。
- このようなCPUでは、複数の処理をスレッドで同時に行う場合、「**少し実行して他の処理に切り替える**」ということをする。
- これを「**コンテキストスイッチ(context switch)**」と言う。これはOSのスケジューラによって自動的に行われる。

ハードウェア並行性

- 複数コアを持つCPUでは、コンテキストスイッチを必要とせず、複数の処理を**真に同時実行**できる。
- ハイパースレッディングという機構を持ってるCPUであれば、1コアで2つの処理を同時に実行できたりもする。
- このような、ハードウェア性能によって真に同時実行できる性質を「**ハードウェア並行性(hardware concurrency)**」と言う。



非決定性

- 並行プログラミングは、OSのスケジューラが行うコンテキストスイッチによって、複数処理を切り替えながら実行する。
- そのため、実行結果が毎回同じにならないことがある。
- この性質を「**非決定性(non-deterministic)**」と言う。

非決定性で起こる問題

- 非決定性は、複数スレッドが**共通のオブジェクトを操作**するときに問題が起こる。
  - 一方のスレッドがオブジェクトxにAを代入する。
    もう一方のスレッドがxにBを代入する。
  - このプログラムは、OSのスケジューラによって実行タイミングが変わるため、実行結果が「**どちらになるかわからない**」としか言えなくなる。

非決定性を飼い慣らすには

- 並行プログラミングでは、安定しない実行結果を飼い慣らし、自分が望む結果を常に得られるようにする必要がある。
- そのためにできる基本的なこと：
  - 排他制御する。
  - タスクの独立性を高めて、できるだけ共有オブジェクトを操作しない。
  - 並行プログラミングのデザインパターンを適用する
  - より抽象的なプログラミング機構(言語拡張やライブラリ)を活用する。

排他制御する

- **ミューテックス(mutex：MUTual EXclusion)**は、ひとつのリソースに複数スレッドが同時アクセスしないよう、排他制御する仕組み。
- 以下の機構も採用するといいだろう：
  - ミューテックスをより高級にした条件変数(condition variable)やFuture／Promise。
  - 排他制御を内部的に行ってくれる並行コンテナ
  - さらなるパフォーマンスのためのアトミックオブジェクトとロックフリーアルゴリズム。

タスクの独立性を高める設計

- 並行プログラミングを意識せずに作られた、タスクの独立性が高くないプログラムは、並行に処理することが難しい。
- 「**関数は引数のみに依存して処理を行う**」という原則に従ってプログラムを書くように心がけると、タスクの独立性が高くなる。
- HaskellやOCaml、F#といった言語で**関数型プログラミング**を学べば、そういったプログラムを自然に書けるようになるだろう。

デザインパターン

- 並行プログラミングで頻出する状況を、うまく解決するデザインパターンが存在する。
- 多くの状況では、これらのパターンを適用することで、設計バグを未然に防ぐことができる。
  - Producer-Consumerパターン
  - メッセージパッシング (Actorモデル)
  - Proactorパターン

抽象的なプログラミング機構

- プログラミング言語の機能、もしくは言語拡張で並行プログラミングをしやすくするものがある。
  - トランザクショナル・メモリー
  - OpenMP、Intel Cilk Plus (C++)
  - synchronizedブロック (Java)

避けては通れない道

- 並行プログラミングは、もはや避けて通ることはできません。
- 今回はキーワードのみを示したものも多かったですが、これらを調べ、原理・原則に従ったプログラミングを心がけ、事故をできる限り減らしてください。

オススメ書籍

- Java並行処理プログラミング
- 並行コンピューティング技法
- The Art of Multiprocessor Programming
- 構造化並列プログラミング
- C++ Concurrency in Action