非同期処理の書き方

# 概要

非同期処理に関連するライブラリや言語構文はいろいろあって、どういう場面でどれを使えばいいのか、慣れないうちは迷うと思います。

C#/.NET Frameworkでも、非同期処理は、バージョン1.0がリリースされてからもっとも進歩が著しかった分野です。C# 5.0/.NET 4.5でようやく落ち着きましたが、それまでの変遷もあって、なおのことわかりにくくなっていると思います。

そこでまず、本項では、C# 5.0/.NET 4.5以降で推奨される書き方の例と、その根拠を大まかに説明します。

詳しくは、次項以降で説明していきます。全体を通して、どうして推奨されるのか、基礎的な原理まで理解してもらうことで、他の言語・環境への応用や、将来もっと発展的な技術が生まれたときの良し悪しの判断にも活かせると思います。

# ThreadよりもTask

## 書き方の例

|  |  |
| --- | --- |
| 悪い例 | 良い(まだマシな)例 |
| for (int i = 0; i < num; i++) {     var t = new Thread(\_ =>          b[i] = F(a[i])         ); } | for (int i = 0; i < num; i++) {     Task.Run(() =>          b[i] = F(a[i])         ); } |

悪い例では、Threadクラスを使って、データの数(num)だけ同時にスレッドを立てています。一方、良い例では、Taskクラスを使っています。

特に、多数のデータに対して掛けたい処理(この例の場合はメソッドF)が小さな処理の場合、この2例の差は顕著になります。Threadクラスは、細々とした処理を大量に実行するのには向いていません。

## 解説

.NETにはスレッドを使うためのクラス/メソッドはいくつかあります。

* Threadクラス(System.Threading名前空間)
* Delegateクラス(System名前空間)のBeginInvoke/EndInvokeメソッド
* ThreadPoolクラス(System.Threading名前空間)
* Taskクラス(System.Threading.Tasks名前空間)

大まかに分けると、Threadクラスとその他3つに分かれます。

Threadクラスは、OSが提供するスレッド機能そのものを操作するものです。低レイヤー過ぎて、よほど非同期処理に詳しい人以外は直接使ってもまず性能の良いコードにはなりません。

スレッドは、新たに立てるのも、立てたものを維持し続けるのも結構大きなコストになります。そこで通常は、少数のスレッドを効率的に使いまわす仕組みとして、スレッド プールという機能を使います。Threadクラス以外の3つはこちらのスレッド プールを使うものです。Taskクラスは.NET 4以降でしか使えませんが、後発だけあって一番使いやすく、今となっては他の2者(Delegeta.BeginInvokeとThreadPool)は使いません。

## 補足

この例のような処理の場合、以下のように、Parallelクラス(System.Threading.Tasks名前空間)を使う方が簡単です(それにおそらく、この用途ならこちらの方がパフォーマンスもいい)。

Parallel.For(0, num, i =>  
    b[i] = F(a[i])  
);

ちなみに、Parallelクラスも、内部的にはTaskクラスを使っています。

## (次節以降で説明する)ポイント

* スレッドのコスト
* スレッド プール

# 非同期I/O

## 書き方の例

|  |  |
| --- | --- |
| 悪い例 | 良い例 |
| using (var r = new StreamReader("some.txt")) {     var t = Task.Run(() => r.ReadToEnd());     Console.WriteLine(await t); } | using (var r = new StreamReader("some.txt")) {     var t = r.ReadToEndAsync();     Console.WriteLine(await t); } |

多くの場合というか、つい最近まで(.NET 4世代くらいまで)、ファイル読み書きなどのいわゆるI/O処理には、同期版と非同期版の2種類のAPIが用意されていました。

悪い例では、自分でスレッドを立てて、その中で同期版のI/Oメソッド(ReadToEnd)を呼んでいます。一方、良い例では、最初から非同期版のI/Oメソッド(ReadToEndAsync)を呼んでいます。

最近では(.NET 4.5世代以降)、非同期版のメソッドしかAPI提供しないことが増えています。

## 解説

非同期処理は2種類に分類されます。

* CPU-bound: 「CPUに性能を束縛される」という意味。計算に時間がかかる(＝ CPUを専有しちゃう)処理のこと。
* I/O-bound: 「I/Oに性能束縛される」という意味。CPUの外の世界とのやり取り(＝I/O、Input/Output)を待っている時間が長い処理のこと。

この例で出したファイル処理はI/O-boundな処理の典型になります。

スレッドは、複数の処理でCPUをシェアして使うための仕組みです。一方、I/O-boundな処理は、CPUを全く使う必要はないので、このためにスレッドを1つ立てて専有するのはいろいろと無駄が多いです。

I/O用の非同期APIが正しく作られていれば、スレッドを専有せずにI/Oの完了を待つ作りになっています(OSの内部にI/O完了ポートと呼ばれる機能があって、これを使った作りになります)。「APIが正しく作られていれば」という前提は必ずしも守られてはいないかもしれませんが、少なくとも、自分で1つスレッドを立てて同期的にI/O完了を待つよりはマシなパフォーマンスが得られるでしょう。

## 補足1

「パフォーマンスを求めるならば非同期I/Oを使え」というのは、それこそ.NETよりもはるか前の時代から言われていることです。しかし、長らくの間、非同期I/Oを使うためにはソースコードがかなり複雑化しがちで、「多少パフォーマンスを落としてもわかりやすいコードを書く方がいい」という結論に至ることの方が多かったです。

今改めて非同期版のAPIが推奨されるのは、C# 5.0でawait/async構文が追加されて、非同期処理を書きやすくなったからです。

## 補足2

ちなみに、逆に、CPU-boundな処理は、必ず同期的なAPIだけを提供すべきです。必要に応じて、利用者側で(Task.Runなどを使って)非同期処理化します(非同期API提供しようとしても、結局内部でTask.Runを呼ぶだけのコードになる)。

## (次節以降で説明する)ポイント

* I/O
* I/O完了ポート

# データ競合

## 書き方の例

|  |  |
| --- | --- |
| 悪い例 | 良い(まだマシな)例 |
| var count = 0; Parallel.For(0, num, i => {     ++count; }); | var count = 0; var sync = new object(); Parallel.For(0, num, i => {     lock (sync)         ++count; }); |

これは、悪い例の側はそもそも正しく動作しません。問題を解決するには、良い例のように、lockステートメントを使います。

## 解説

複数のスレッドから同じ変数を読み書きする場合、競合を起こしてデータが狂うことがあります。インクリメントのような単純に見える演算ですら、実際には読み込み、計算、書き戻しという3つの処理にわかれます。これらの処理の合間に、別のスレッドで同じ処理が走ることで問題が起きます。

これを回避するために、複数のスレッドが同時に同じ処理を実行しなくする機構としてロック(lock: カギをかける)というものが用意されています。あるい範囲のコード(この例の場合は++count)にロックをかけて、その場所を抜けるまで、他のスレッドが同じ場所を実行できなくします。

C#では、lockステートメントという構文を使ってロックをかけることができます。

## 補足

Parallelクラスはループを数万程度ずつ区切って複数のスレッドに配分するので、要素数(この例の場合num)が数千程度であれば単一スレッドで実行されて、問題が出なかったりします(問題が100%起きるわけでないというのが、デバッグ時に問題に気づきにくくてなおのこと状況を悪くしたりします)。

また、この例はまだわかりやすく競合を起こしやすいコードを書いていますが、実際には、ごくごくまれにしか競合を起こさない場合もあります。再現方法もわからず、どこが原因なのか特定できない非常につらいバグになりがちです。

## (次節以降で説明する)ポイント

* データ競合
* ロック

# スレッドごとに別計算

## 書き方の例

|  |  |
| --- | --- |
| 悪い例 | 良い例 |
| var count = 0; var sync = new object(); Parallel.For(0, num, i => {     lock (sync)         ++count; }); | var count = 0; Parallel.For(0, num,     () => 0,     (i, state, localCount) => localCount + 1,     localCount => count += localCount     ); |

悪い例は、前節の「まだマシな例」そのままで、ロックをかけて競合回避しています。一方で、良い例の方は、少しコードは複雑ですが、ロックを使わずに集計を行っています。

## 解説

ロックをかければとりあえずデータの競合は回避できますが、ロックをかける処理は非常にコストがかかります。(Parallelクラスを使いたくなるのは、だいたい並列実行によってパフォーマンスを上げたい場合なわけですが)せっかく並列処理をしていても、ロックのコストが高すぎて、逆に性能を著しく落とすことも多いです。並列処理をしたければ、そもそもあまりロックを必要としない(＝ 同じデータの読み書きをしない)アルゴリズムを考えなければなりません。

この例の場合は実はそう難しくありません。整数の足し算なら、スレッドごとに部分集計したうえで、最後にスレッドごとの結果を集計すれば、そもそも競合(＝ 同じデータの読み書き)は起こしません。こういう考え方を、スレッドごとに独立したデータ読み書きをするという意味で、スレッド ローカル(thread local: スレッド内でだけ局所的に使う)な処理といいます。

.NETのParallelクラスのように、並列処理による高速化を目的とするライブラリには、通常、スレッド ローカルな処理(+ 最後にスレッドごとの結果の集計)を行うためのオーバーロードが用意されています(ない場合はライブラリ作成者が未熟か、手抜きです)。これを使ったコードが、良い例の方になります。

## 補足

ちなみに、ロックをかける以外にも、interlocked命令というものを使うことでも競合回避ができる場合があります。この例の場合は以下のようにも書けて、ロックをかけるよりは、この方がまだ幾分かよいパフォーマンスが出ます(それでも、良い例のようにスレッド ローカルな処理を心がけるのと比べると、数十倍は遅いです)。

var count = 0;  
Parallel.For(0, num, i =>  
{  
    Interlocked.Increment(ref count);  
});

ちなみに、.NETでinterlocked命令を使いたい場合にはInterlockedクラス(System.Threading名前空間)を使います。

## (次節以降で説明する)ポイント

* ロックやデータ共有のコスト
* スレッド ローカル
* interlocked命令

# ロック削減と自動実装イベント

C#のイベントの自動実装(参考: イベントの自動実装)は、スレッド安全なコードを生成します。C#の言語仕様上はスレッド安全であることだけを規定していて、実際どうやって保証するかは決まっていません。そして実際、C# 4.0で実装方法が変更されました。変更の理由は、主にパフォーマンスの改善です。

|  |  |
| --- | --- |
| C# 3.0以前(改善前) | C# 4.0以降(改善後) |
| [method:MethodImpl(MethodImplOptions.Synchronized)] add {     \_x = (EventHandler<XArgs>) Delegate.Combine(\_x, value); } | add {     EventHandler<XArgs> e1 = \_x, e2;     do     {         e2 = e1;         var value2 = (EventHandler<XArgs>)Delegate.Combine(e2, value);         e1 = Interlocked.CompareExchange(ref \_x, value2, e2);     }     while (e1 != e2); } |

改善前のコードはSynchronizedオプションというものを使っていて、これは、最終的にはlockステートメント相当のコードに展開されます。一方、改善後のコードはInterlockedクラス(System.Threading名前空間)を使った同期アルゴリズムで書かれています。

## 解説

前節で説明した通り、ロックは結構重たい処理で、可能な限り避けたいものです。いずれ説明しますが、ロックはソフトウェア割り込み(特権モード移行)を必要としたり、過剰にスレッドのスイッチが発生したりといった問題があります。旧実装は、もろにこの問題に直面します。

ロックの負担の問題はかなり昔から知られていて、ロックを避けつつスレッド安全性を保証する仕組みがいろいろと考えられています。例えば、interlocked命令と呼ばれるような特殊なCPU命令を使うことで、ロックを使うよりはだいぶ小さな負担で済む同期アルゴリズムがあります。その1つが新実装のようなコードです(Interlockedクラスのメソッドはinterlocked命令を使っています)。

## (次節以降で説明する)ポイント

* ロックやデータ共有のコスト
* interlocked命令