C#
非同期・並列プログラミング入門

北山洋幸◉著

Task、async/await、Invokeの要諦を学ぶ

■サンプルファイルのダウンロードについて

本書掲載のサンプルファイルは、下記URLからダウンロードできます。

```
https://----------------------------
```

ファイルサイズを小さくするため、コンパイル後のバイナリファイルは削除しています。

ソリューションファイルやプロジェクトファイルなどは、そのまま利用できることを保証するものではありません。

ダウンロードファイルの提供は、ソースコードを入力する手間を省くことを目的としています。

・本書の内容についてのご意見、ご質問は、お名前、ご連絡先を明記のうえ、小社出版部宛文書（郵送または E-mail）でお送りください。
・電話によるお問い合わせはお受けできません。
・本書の解説範囲を越える内容のご質問や、本書の内容と無関係なご質問にはお答えできません。
・匿名のフリーメールアドレスからのお問い合わせには返信しかねます。

本書で取り上げられているシステム名／製品名は、一般に開発各社の登録商標／商品名です。本書では、™ および ® マークは明記していません。本書に掲載されている団体／商品に対して、その商標権を侵害する意図は一切ありません。本書で紹介している URL や各サイトの内容は変更される場合があります。

はじめに

本書はC# の並列処理や非同期処理について記述した書籍です。一般的に並列処理は性能向上を目的にすることが多いのですが、本書はユーザーインターフェース（UI）の改善についても多くのページを割きました。

C# 4.0 および .NET Framework 4 で、タスク並列ライブラリ（TPL: Task Parallel Library）が追加されました。これによって、並列プログラムを容易に、かつ柔軟に記述できるようになりました。これは、.NET Framework が従来から持つ生産性の高さに加えて、性能とスケーラビリティの向上も同時に達成する大きな力となります。さらには、Task クラスと紐付いた async/await が C# 5.0 で導入され、UI を Task と簡単に統合して記述できるようになりました。そして、C# 7.1 では、Main メソッドに async なども指定できるようになり、非同期プログラムの開発に関する拡張が行われてきました。ほかにも、.NET 4.5 で Dispatcher クラスに InvokeAsync メソッドが追加されるなど、TPL 導入時から多くの拡張が行われています。

本書では、常に話題となる async/await やそれに絡むデッドロックや例外の捕捉などについてまとめた書籍です。本書が C# の非同期プログラミングや並列プログラミングの理解の一助となることを祈念いたします。

本書の対象読者は，以下のような人を対象としています.

- C# で非同期プログラミングを習得したい人
- C# でより良いユーザーインターフェースを実現したい人
- C# で並列プログラミングを習得したい人

本書を参考に C# における並列プログラミングと非同期プログラムの開発に役立ててください。微力ながら本書が学習の助けになれば幸いです。

謝辞

出版にあたり、開発環境である Visual Studio を無償公開している米 Microsoft 社、参考文献や参考サイトで情報公開されている団体・個人に深く感謝いたします。ならびに、株式会社カットシステムの石塚勝敏氏にも深く感謝いたします。

2022 年初夏　都立東大和南公園にて 北山洋幸

参考文献、参考サイト、参考資料

1. “C# reference”, https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/
2. “C# documentation”, https://docs.microsoft.com/en-us/dotnet/csharp/
3. “.NET API browser”, https://docs.microsoft.com/en-us/dotnet/api/?view=net-6.0
4. “System Threading Tasks”, https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks?view=net-6.0
5. “Async/Await - Best Practices in Asynchronous Programming”, https://docs.microsoft.com/en-us/archive/msdn-magazine/2013/march/async-await-best-practices-in-asynchronous-programming
6. “非同期プログラミングのベストプラクティス”, https://docs.microsoft.com/ja-jp/archive/msdn-magazine/2013/march/async-await-best-practices-in-asynchronous-programming
7. @acple@github, “Task を極めろ！ async/await 完全攻略”, https://qiita.com/acple@github/items/8f63aacb13de9954c5da
8. “Nullable value types (C# reference)”, https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-value-types
9. “High DPI support in Windows Forms”, https://docs.microsoft.com/en-us/dotnet/desktop/winforms/high-dpi-support-in-windows-forms?view=netframeworkdesktop-4.8
10. “Threading in C#”, https://www.albahari.com/threading/
11. 北山 洋幸 , “C# による Windows システムプログラミング”, カットシステム
12. 北山 洋幸 , “Parallel プログラミング—in .NET Framework 4.0”, カットシステム
13. 北山 洋幸 , “パーソナルな環境で実践的に学ぶ並列コンピューティングの基礎”, カットシステム

本書の使用にあたって

開発環境、および、実行環境を説明します。

■ Windows バージョン

Windows のバージョンへ依存するとは思えませんが、本書のプログラムの開発および動作確認は Windows 10 Pro（64 ビット）で行いました。

■ Visual C# のバージョンとエディション

無償の Visual Studio Community 2022 を使用します。「.NET デスクトップ開発」をデフォルトでインストールしてください。

■ .NET バージョン
本書のプログラムは .NET 6.0 を使用して開発および動作確認を行いました。.NET Framework4.7.2 以降でも問題ないと思われます。細かな C# コンパイラや .NET Framework のバージョン依存は、本文に注釈を入れています。

■ URL
URL の記載がある場合、執筆時点のものであり変更される可能性もあります。リンク先が存在しない場合、キーワードなどから自分で検索してください。

用語

用語の使用に関して説明します。

■ カタカナ語の長音表記
「メモリー」や「フォルダー」など、最近は語尾の「ー」を付けるのが一般的になっていますので、なるべく「ー」を付けるようにします。ただ、開発環境やドキュメントなどに従来の用語を使用している場合も多く、参考資料も混在して使用しているため、本書では、語尾の「ー」は統一していません。

■ ユーザーインターフェース
ユーザーインターフェースを UI と省略する場合があります。

■ クラスとオブジェクト
クラスとオブジェクトはなるべく使い分けています。クラス、オブジェクト両方に適用できる内容については、クラス、オブジェクトを省いている場合もあります。

■ クラスとインスタンス
クラスとインスタンスもなるべく使い分けています。クラス、インスタンス両方に適用できる内容については、クラス、インスタンスを省いている場合もあります。

■ ソースリストとソースコード
基本的に同じものを指しますが、ソースリストと表現する場合ソース全体を、ソースコードあるいはコードと表現する場合ソースの一部を指します。

■ フォームとウィンドウ
同じものを指しますが、デザイン時にフォームと呼び、実行時にウィンドウと呼ぶことがあります。

■ 非同期呼び出しと非同期処理
非同期呼び出しを非同期や非同期処理と省略する場合があります。正確な表現は文脈から判断してください。分かりにくい個所では明示的に使い分けています。

■ .NET と .NET Framework
本書で紹介するプログラムは、.NET Framework と .NET のどちらで開発しても構わないものが多いです。このため、.NET Framework と .NET を特別区別せず、.NET と表現している場合があります。

■ Task とタスク
ある一定の作業を一般名詞のタスク（作業）、Task クラスのことを Task と表記することが多いですが、明確に使い分けない場合もあります。何を指すかは文脈から判断してください。

■ タスク（Task）とスレッド
C# では、Task クラスを使用してスレッドを起動するときがあります。このため、スレッドとタスク、そして Task を混在して使用します。どれも同じものを指しますが、スレッドをタスクと表現する場合があり、その逆もあります。

■ メインスレッドとワーカースレッド
メインスレッドとワーカースレッドという用語を多用しています。正確には、元スレッドと新規に生成したスレッドのことです。説明を行いやすいようにメインスレッドとワーカースレッドを使用します。

■ 変数とオブジェクト
本来オブジェクトと表現するのが適切なものを、変数と表現している場合があります。

目次

はじめに iii

第 1 章　並列と非同期概論 1

1.1　並列化概論 1
1.2　逐次と並列 2
●逐次と並列のフロー ……2 ／●逐次と並列のスレッド ……3
1.3　プロセス・スレッドによる並列 3
1.4　並列化する目的 4
1.5　並列化の課題 7
●オーバーヘッド ……7 ／●アクセス競合 ……7 ／●複雑化 ……7
●スケーラビリティの喪失 ……8 ／●ポータビリティの喪失 ……8
1.6　タスク並列ライブラリ（TPL）概論 9
1.7　C# の並列と同期 10
●単純なスレッド（戻り値も引数もない）……10 ／● async/await ……10
●戻り値のあるスレッド ……12 ／●引数のあるスレッド ……13
●任意のスレッドと同期 ……13 ／●全てのスレッドと同期 ……15
●任意のスレッドと同期 ……16 ／●まとめ ……16

第 2 章　並列処理 17

2.1　スレッドの基本 17
●シンプルスレッド（Task クラスで記述）……17 ／● Task.Factory.StartNew で記述 ……20
●明示的に起動 ……22 ／● Thread クラスで記述 ……23
2.2　引数 24
●スレッドに引数 ……24 ／●明示的に起動 ……26 ／●引数・起動時に評価 ……27
●並列処理をメソッドで記述 ……28 ／● Task.Factory.StartNew ……29
● Thread クラスで記述 ……30
2.3　戻り値 32
●戻り値 ……32 ／● Thread クラス ……35 ／● Task 戻り値 ……39
2.4　タスク配列 40
●単純なタスク配列 ……40 ／●タスク配列と戻り値参照 ……41
●同じ処理を行うタスク配列（1）……43 ／●同じ処理を行うタスク配列（2）……44

2.5 タスク継続 47
●単純なタスク継続 ……47 ／●多段タスク継続 ……49 ／●複数タスクの継続 ……50
2.6 入れ子タスクと子タスク 51
●入れ子タスク ……52 ／●子タスク ……54

第 3 章 Task と非同期 55
3.1 同期処理と非同期処理 56
●同期処理 ……56 ／●非同期処理 ……56 ／●非同期処理と同期 ……57
● Task と async/await ……57 ／●同期処理のコード例 ……58
●非同期処理のコード例 ……59 ／●非同期処理と同期のコード例 ……59
3.2 Task と非同期 60
●待ちのある処理 ……66
3.3 フリーズする例 68
3.4 例外発生 71
3.5 結果誤り 74
3.6 ラムダ式でなくメソッドで記述 76
3.7 戻り値 77
3.8 非同期と Task の Wait メソッド 79
● Windows フォームアプリの例 ……83 ／● Task.Result でデッドロック ……84
3.9 非同期プログラミングガイドライン 88
● async void を使用しない ……88 ／●すべて非同期 ……89
●コンテキスト ……91 ／●ガイドラインのまとめ ……94

第 4 章 Task と UI 更新 99
4.1 Task と async/await 99
4.2 Task クラスと Invoke 100
● Windows フォームアプリの例 ……103
4.3 Thread クラスと Invoke メソッド 105
●ラムダ式をデリゲートへ ……107 ／● Invoke メソッドの必要性判断 ……108
4.4 Task クラスと BeginInvoke メソッド 110
●同期する ……112 ／● Windows フォームアプリ ……114
4.5 Task クラスと InvokeAsync メソッド 115
4.6 頻繁に UI 更新 116
● async/await を使って頻繁に UI 更新 ……120

4.7 BackgroundWorker で頻繁に UI 更新 121
●null 許容値型 ……124 ／●冗長性の排除 ……126
4.8 Queue クラスで頻繁に UI 更新 128

第 5 章 Task と例外 133
5.1 async/await で例外処理 134
●ラムダ式をメソッドで記述 ……135
5.2 Wait で捕捉 136
5.3 捕捉できない 137
●タスク内で捕捉 ……138 ／●Thread クラスで捕捉 ……140
5.4 戻り値 142
5.5 UI と例外 143
●ラムダ式をメソッド化 ……145 ／●Wait メソッドで捕捉 ……146
5.6 複数タスクと例外 148
●複数タスクを await ……150 ／●Wait メソッドで全例外を捕捉 ……151
5.7 配列タスクと例外 152
●戻り値を参照 ……154

第 6 章 排他処理 157
6.1 Interlocked クラス 158
●ラムダ式をメソッドで記述 ……161
6.2 Monitor クラス 162
6.3 lock 文 164
6.4 AutoResetEvent クラス 167
6.5 Mutex クラス 169
6.6 Semaphore クラス 170
6.7 コンカレントコレクション 172
●ConcurrentQueue クラス ……172 ／●ConcurrentStack クラス ……176
●ConcurrentBag クラス ……178
6.8 コンカレントコレクション応用 179

第7章　Parallel クラス 185

●C# とデータ並列概論 ……185

7.1　単純な Parallel.For 186

7.2　単純な Parallel.ForEach 188

7.3　Stop メソッドで脱出 189

7.4　IsStopped プロパティを監視 192

7.5　Break メソッドで脱出 193

7.6　LowestBreakIteration プロパティ 195

7.7　スレッドローカル変数と Parallel.For 198

7.8　スレッドローカル変数と Parallel.ForEach 201

7.9　ループ取り消し 203

7.10　Partitioner クラス 206

7.11　分割数の指定 208

7.12　Parallel.Invoke 209

●もう少し複雑な例 ……211

付 録 213

付録 A　Visual Studio のインストール 213

付録 B　プロジェクト作成 220

索 引 225

第1章 並列と非同期概論

1

本章は、一般的な並列概論、C# 特有なタスク並列ライブラリ、そして少し具体的な C# の並列と同期について、図やコードを示し解説します。

1.1 並列化概論

並列とは、同時に複数の処理を行うことです。並列処理と対立する概念が逐次処理で、逐次処理においては、複数の処理が順番に実行されます。並列処理は、時間時で観察した場合、物理的に複数の処理装置を持ち本当に並列処理するものと、処理装置が並列数より少なく、リソースを時分割で利用し、並列処理するものがあります。ただ、広義には、どちらも同じ並列処理です。並列は細かく分類すると、プロセス、スレッドによるもの、コプロセッサーを用いてオフロードするもの、MPI などのようにネットワークで分散するものなど多様です。また、処理を並列するかデータを並列するかで分類することもできます。本書は、C# の Task クラスなどに着目しますので、処理をスレッドで並列するものを解説します。

1.2 逐次と並列

逐次処理は、ある特定の瞬間でシステムを観察したとき、1つの処理しか行っていません。並列処理は、ある瞬間を観察すると、同時に2つ以上の処理を行っています。言葉が示すように、2つ以上のことを並んで処理します。

これを詳しく観察すると、人間のように低速なデバイスには同時に2つの処理を行っているように見えても、実際は高速なデバイスであるCPUが瞬間的にいくつもの作業を時分割で掛け持ちし、並列に実行しているように見せかけている場合があります。このような場合、狭義には並列処理している訳ではありません。しかし、広義にはこのような場合も並列処理と呼びます。例えば、CPUを1つしか搭載していないコンピュータが、同時に2つ以上の処理を行う場合がそれにあたります。本書で扱う並列処理は広義の並列です。

逐次と並列のフロー

逐次処理は、言葉が示すように、いくつかの処理を順序良く一つずつ処理することです。並列処理とは、並列に処理できる部分を同時に並行実行します。この例では、処理Bが並列化可能な処理であるとします。

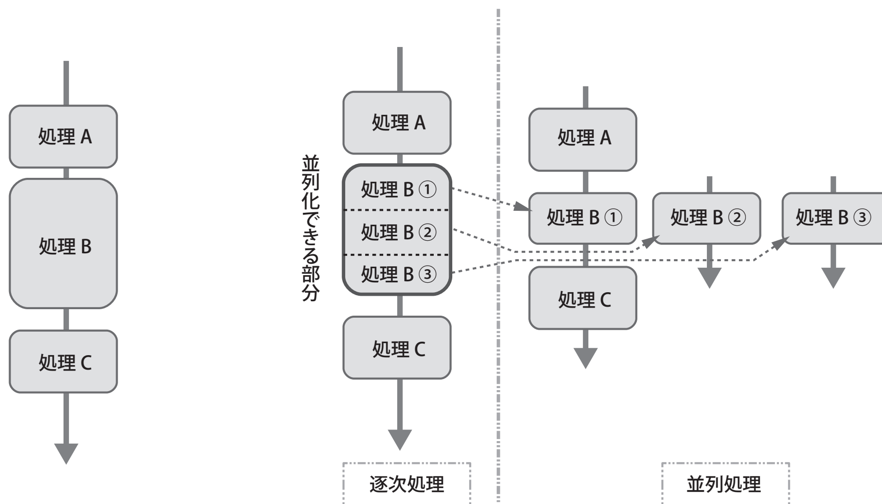

図1.1●逐次のフロー

図1.2●並列のフロー

逐次と並列のスレッド

逐次処理は、ひとつのスレッドで全処理を順序良く処理することです。並列処理とは、並列に処理できる部分をスレッドに分割して並列実行します。

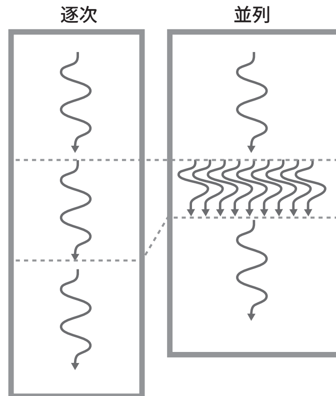

図1.3●逐次と並列のスレッド

1.3 プロセス・スレッドによる並列

並列化をプロセスとスレッドから分類します。スレッドはコンテキストを切り替えず同一プロセス内で並列処理を実現します。つまり、1 つのプログラム内で並列化を実現します。

プロセスを並列化させる方法は、完全に分離された複数のプログラムが協調しながら 1 つの目的を達成します。プロセスを分離すると、プログラムは論理的に分離された空間で動作するため、並列化された部分は疎結合となり競合などの問題が低減されます。プロセスは、メッセージを使ってデータやコマンドを交換するため、スレッドに比べデータ交換速度は低速ですが、プロセスが同一コンピュータ内に存在する必要はありません。このため、スレッドによる並列化に比べスケーラビリティは高く、自由度は高くなります。

スレッドによる並列化は、同一プロセス内で並列処理を行います。同一コンテキストで動作するため情報交換は高速です。また、同一メモリ空間で動作するため高速な並列処理が実現できます。ただし、データアクセスで競合などが起こります。プロセスによる並列化に比べて同期処理が煩雑になるでしょう。本書で扱うのはスレッドによる並列化です。プロセスによる並列化と、スレッドによる並列化の概念図を次に示します。

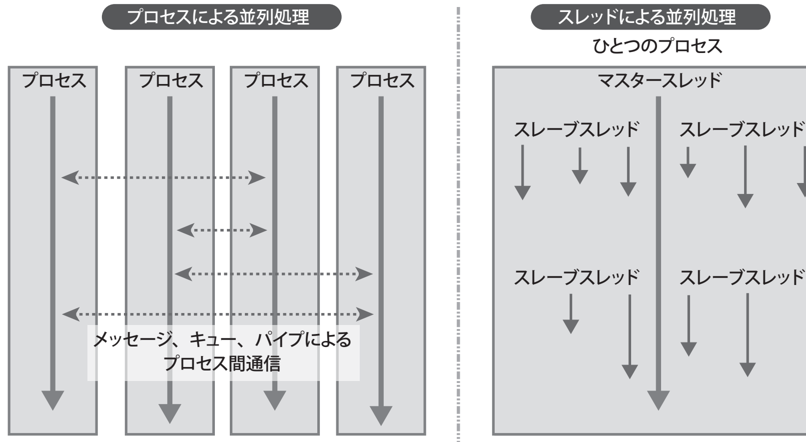

図1.4●プロセスによる並列化とスレッドによる並列化の概念図

1.4 並列化する目的

プログラムを並列化する理由は 1 つではありません。以降に代表的な目的を箇条書きにします。

- UI（ユーザーインターフェース）の向上
- CPU ブロック防止
- 性能向上

C# で良く使われる並列化は、UI（ユーザーインターフェース）の向上や CPU ブロックの防止です。シングルスレッドのプログラムは、1 つの作業を行っている際に別の要求を受け付けられません。これではプログラムの使い勝手の悪いシステムになってしまいます。このため、並列処理を導入し、UI がフリーズするのを回避します。あるいは、ネットワークや外部 I/O など、応答の遅い外部アクセスが CPU をブロックしてしまい、プログラムがフリーズするような挙動を示す場合があります。これも複数のスレッドを起動し、それぞれの処理を行わせることによって優れた UI を持つシステムを提供します。または、単純な外部 I/O 待ちだけである場合は、当該処理の完了をコールバックやシグナルで受け取る非同期処理の導入で解決する場合もあります。本書は、上記の両方のケースを網羅します。

さらに別の目的は、性能向上を目指した並列化です。一昔前まで、コンピュータの CPU は基本的に 1 つでした。正確には、大型コンピュータでは複数の CPU を搭載したものは古くからあり、デスクトップコンピュータでも複数の CPU を搭載したものは存在しましたが、それらはほんのわずかでした。しかし、現在では、一般の人々が使用するデスクトップコンピュータやノート PC にも複数の CPU（CPU コア）が搭載されています。このような環境で、すべての CPU を有効に活用できれば処理速度を驚異的に向上できます。

これらの様子を図にして示します。

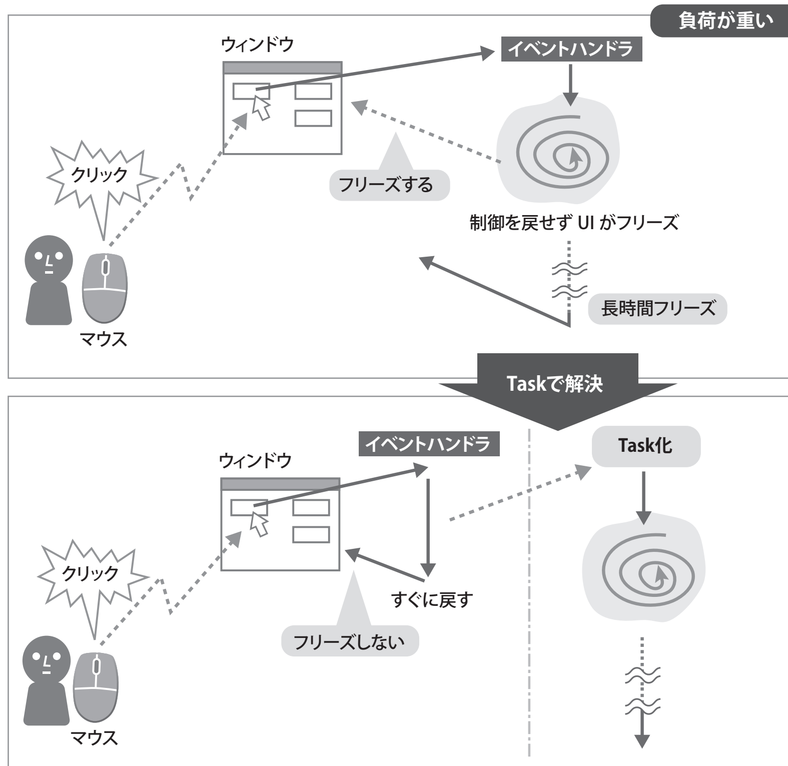

図1.5●UIの向上・負荷が重い場合

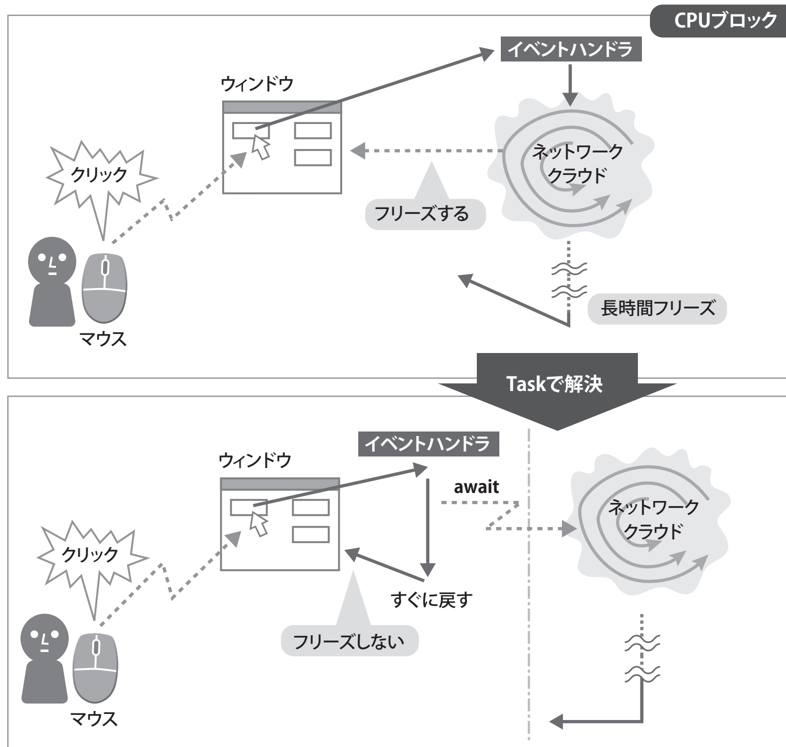

図1.6●CPUブロック

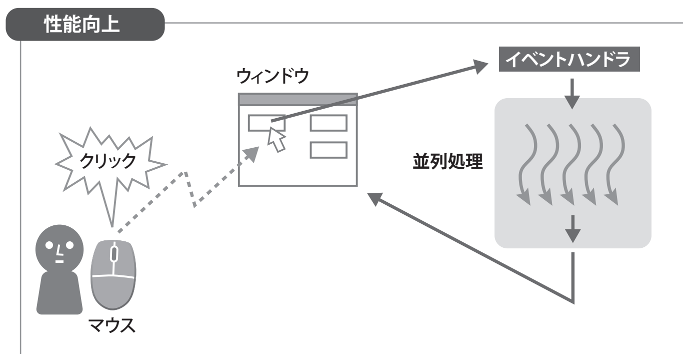

図1.7●性能向上

1.5 並列化の課題

並列化によるメリットはすでに述べた通りです。プログラムの高速化はもちろん、使い勝手も良くなります。さらには、消費電力を増大させることなく性能を向上できます。並列化は、多くの利点を持つ高速手法ですが、良いことばかりではありません。本節では並列化の課題を示します。単にデメリットではなく、並列化による影響についても述べます。

■ オーバーヘッド

プログラムを並列化するには、そのための準備が必要になります。これを一般的に、並列化のオーバーヘッドと呼びます。あまりにもオーバーヘッドが大きいと、並列化による速度向上分が相殺され、その結果、並列化したことでかえって遅くなってしまうことさえあります。

■ アクセス競合

プログラムを並列化するとさまざまな問題が起こります。その 1 つとして、並列化された部分からデータアクセスすると、複数の部分が同時に動作するためデータアクセスの衝突が発生します。これによって、正常な結果が得られない場合があります。

逐次プログラムでは何の問題も起きない処理が、並列化したために問題を引き起します。プログラムを単純に並列化する場合、性能向上を云々する以前に、正しく処理が行われるか検証する必要があります。例えば、並列に処理する部分で共通のデータを使用する場合、次に示すような方法で処理やデータアクセスが競合しないようにする必要があります。

- 並列化された各コードから排他的にデータアクセスする
- データを複製し、各並列化部を隔離する
- 並列化コードを順番に動作させる

本書では、データアクセス競合についても解説します。

■ 複雑化

基本的に、人間は物事を時系列に捉えるのは得意というか自然なことです。並列処理では、時間の捉え方を変えなければなりません。プログラムコードが、記述されたように上から下へ、順番に実行されると考えてはなりません。さらに、同一時間に複数のコードが動作するため、データアクセスの競合も気を付けなければなりません。処理順に依存性がある場合やデータ間に依存

がある場合、同期処理が必要です。

逐次プログラムの場合、プログラムは制御が移った順に処理されます。このため、課題を理論通り処理するだけです。ところが、並列化したプログラムでは、複数のブロックが同時に動作するため、処理順に依存性がある場合はそれらを制御しなければなりません。このように、逐次プログラムでは不要だった諸々の同期処理や排他制御などが必要です。これを誤ると、正常な結果は得られません。

並列化を利用するとシステムを使いやすく、また高速化できますが、「データアクセス競合」や「同期処理」を適切に実装しないと、使い勝手や性能以前に処理結果が正常でないという、まったく意味のないことになってしまいます。これらは逐次プログラムでは、まったく必要のなかった処理です。これだけでなく、並列化部分の通信や、前処理や後処理も必要になります。

基本的に、並列処理は逐次処理に比べ、はるかに複雑度が増します。結果、不具合が増えるだけでなく、開発工数の増大を招きます。開発増大は、開発期間、ひいては開発コストの増大も招きかねません。

スケーラビリティの喪失

一般的に、スレッド分割などをプログラマが意識してプログラミングすると、システムを変更したときに最適な分割にならない場合があります。

ただ、.NET の機能を使用して開発したプログラムはスケーラビリティを持ちます。たとえば、CPU が 2 つあるパソコンで並列化したプログラムを実行すると、CPU が 1 つのパソコン上で実行した場合の 2 倍の性能を示します。さらに CPU が 16 個のパソコンで並列化したプログラムを実行すると、CPU が 1 つのパソコン上で実行した場合の 16 倍の性能を示します。ただし、これは理想的な場合であり、実際にはまずこれほどの性能向上は達成できません。

もっとも、プログラム全体では、ほとんどの場合で CPU 数に比例して高速化することはありません。この原因の最も大きな理由は、並列化されているコードがプログラム全体の一部に留まるためです。これを示す理論が、有名なアムダールの法則です。また、並列化したコードであっても、オーバーヘッドやデータ競合などによってプログラムの性能は低下する場合があります。

ポータビリティの喪失

並列化プログラムは、CPU 依存やコンパイラ依存が存在します。どの程度、汎用性を持たせ、ポータビリティを向上させるかを考えなければなりません。性能を限界まで最適化すると、自ずとシステム依存せざるを得ません。これが並列化の欠点の 1 つです。

ただ、.NET の機能や C# を用いたプログラムは、抽象化が高いため、一般の並列化に比べポータビリティを喪失する可能性は高くありません。それでもポータビリティをまったく喪失しないとは言い切れません。

このような、システム依存のあるプログラムは、他の環境へ移行する場合、何らかの手を加えなければなりません。運が悪いと、単に命令やインターフェースを書き換えるだけでなく全体を作り直す必要もあります。ポータビリティと性能のバランスを、どの程度でバランスさせるかは重要な課題です。

1.6 タスク並列ライブラリ（TPL）概論

タスク並列ライブラリ（Task Parallel Library、TPL）は、.NET Framework 4 で追加されました。これまでもタスクやスレッドに対応したクラスは存在しましたが、.NET Framework 4 以降では、マルチスレッドや並列処理プログラムの開発では TPL の利用を推奨します。

TPL は、.NET Framework 4 の `System.Threading` 名前空間および `System.Threading.Tasks` 名前空間におけるパブリック型と API のセットです。TPL は、並列処理プログラムをより容易に、かつ柔軟に記述できるように用意され、プログラマの生産性とプログラムの性能、スケーラビリティを向上させます。TPL は、並列化すべきかどうかの判断および並列化数を自動で決定し、プロセッサの処理能力を有効に活用します。プログラマは決められた作法にのっとって TPL を利用することで、最適なプログラムを開発できます。

TPL によって、タスク並列では記述や起動負荷を軽減し、データ並列ではスケーラビリティや粒度の小さな並列化へ容易に対応できます。

スレッドセーフ

.NET Framework のすべての `public static` なメソッド、プロパティ、フィールド、およびイベントは、マルチスレッド環境における同時アクセスをサポートします。このため、.NET Framework のすべての静的メンバは 2 つのスレッドから同時に呼び出せます。競合状態、デッドロック、またはクラッシュが発生することはありません。

.NET Framework のすべてのクラスと構造体については、リファレンスドキュメントでスレッドセーフに関するセクションを調べ、スレッドセーフであるか確認してください。スレッドセーフでないクラスをマルチスレッド環境で使用する場合、必要な同期構造を実現するコードでそのクラスのインスタンスをラップする必要があります。

コントロールは一般的にスレッドセーフでないため、フォームを生成したスレッド以外からのアクセスには注意が必要です。これについては後述します。

1.7 C# の並列と同期

以降に、並列処理と同期の代表的な組み合わせなどを分かりやすく整理して示します。C# は並列処理をいくつかのクラスで記述できますが、ここでは主に使われている Task クラスで説明します。

単純なスレッド（戻り値も引数もない）

まず、戻り値も引数もないスレッドを紹介します。

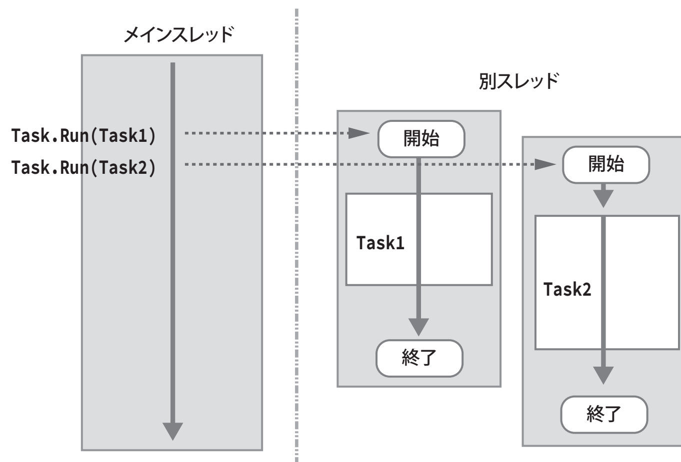

図1.8●戻り値も引数もないスレッドを起動する例

非同期に動作するため、Task1、Task2、そしてメインスレッドの実行順に相関はありません。言いかえると相関のある処理を非同期、かつ終了監視も行わずスレッド化するのは危険です。単純なプログラム以外、このような方式が使用されることは稀です。

async/await

await を指定し、そのスレッドが終了するまで以降の処理を待機させます。await を指定したブロックが終了するまで、await 以降の処理はブロックされます。ただし、メインスレッドはブロックされず、そのほかの処理を継続できます。Wait メソッドや戻り値の参照と違い、呼び出し元のスレッドは待機に入らず、await 以降の処理はスレッド終了後、非同期に呼び出されます。C# で並列処理と、非同期処理を実現したい場合、ほとんどのケースを、この方法で記述できます。

async/await は、CPU を占有する処理、あるいは CPU をブロックするような処理に応用できます。まず、負荷が重く CPU 占有で UI がフリーズするのを避ける様子を示します。

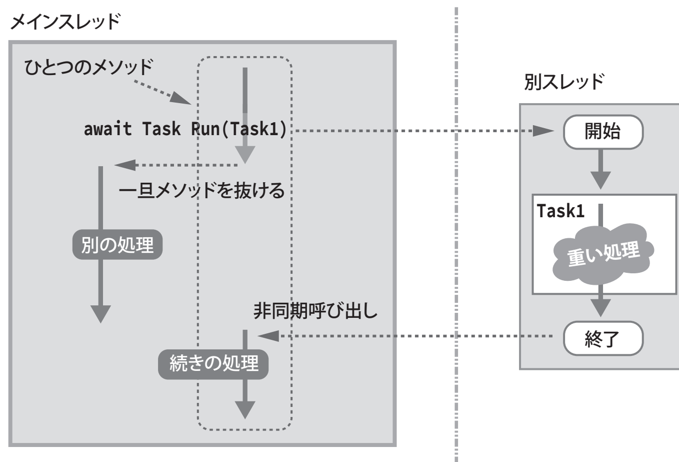

図1.9●CPU占有を避ける

以降に、サーバーとクライアントで通信するようなブロッキング処理で、CPU がブロックされるのを避ける例を示します。

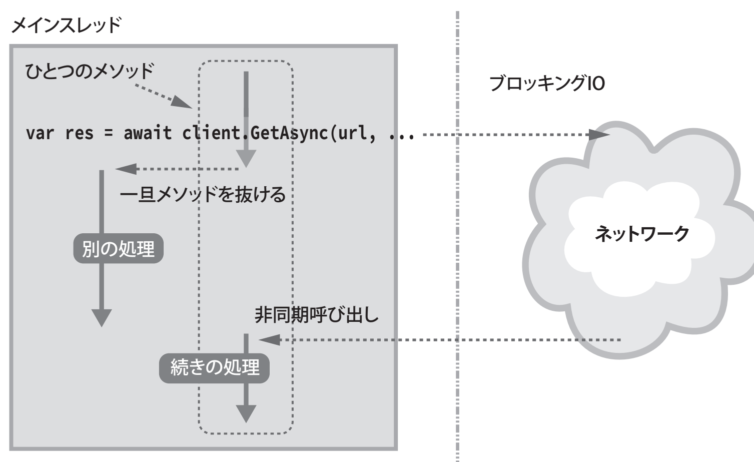

図1.10●CPUブロックを避ける

async/await を利用したメソッドは、同期的に動作します。Task.Run などで起動したスレッドが完了したときに、これと同期しなければならない上記の「続きの処理」が突然非同期に呼び出されるだけであって、メソッド自体は秩序を持って同期的に動作します。