アジョブジ星通信

進捗が出た頃に更新されるブログ。

2019-05-26

いまさら使う TPL Dataflow

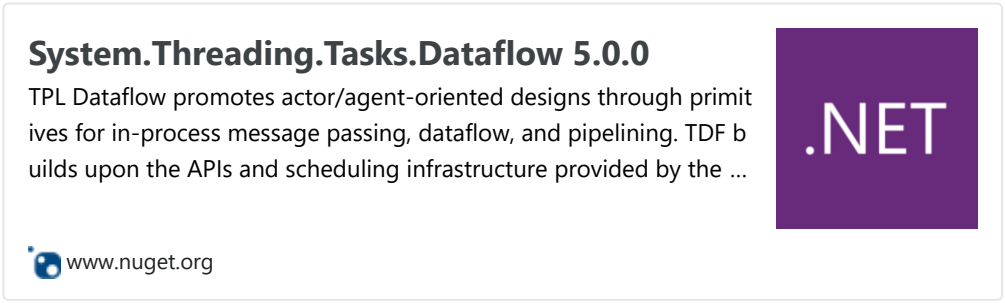
C#

みなさん、並列処理、書いてますか？ つらい、つらいよね、わかるよ。

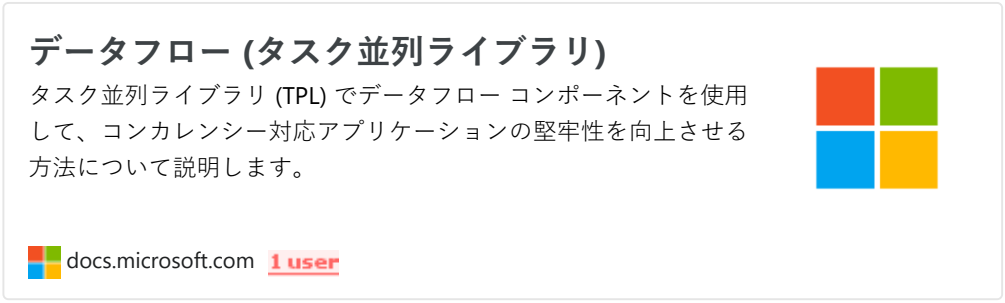
ある部分は高速化のために並列で実行し、ある部分は整合性を取るために直列で実行し、入力ジョブのキューが膨大にならないように流量を調節しながらジョブを入力して……。こんなプログラムをバグなく書くなんて、それはもう大変で、コーナーケースからぼろが出てくる出てくるって感じになります。

さて、そんな面倒な並列プログラミングを支えてくれるライブラリが、結構前（2014 年の NuGet パッケージが存在している。概念自体は 2011 年からある？）から提供されています。名前だけはご存知（個人的な感想）

System.Threading.Tasks.Dataflow、通称 TPL Dataflow です。 .NET Core では標準で含まれていますし、そうでない環境では NuGet からダウンロードできます。強いですね。



公式ドキュメントも比較的しっかり書かれています。日本語翻訳がまともかというと何とも言えないですが、翻訳が崩壊しているページよりはまともです。



今まで、イマイチ使いどころがわからない & 複雑で学習コストが高そうだと感じ、避けてきたのですが、触ってみたらかなり便利ということがわかったので、各種オプションによってどのような挙動をするのかをまとめて、今後使うときのメモとしておこうというのが、この記事の目的です。前半では、この目的通り、使い方を説明します。後半では、同じく TPL(Task Parallel Library) という枠で提供されている Parallel クラスや、似たような概念を提供する Reactive Extensions、System.Threading.Channels との比較を行います。

環境は、 .NET Core 2.2 を想定しています。 System.Threading.Tasks.Dataflow の古いバージョンでは、存在しない機能があるかもしれません。

TPL Dataflow の使い方

インターフェイスの紹介

TPL Dataflow では、データを加工したり、データを入力して処理を行ったりする単位を、データフローブロックと呼んでいます。このデータフローブロックをインスタンス化して、それぞれを接続することで、並列処理を行うシステムを構成するというのがコンセプトです。そんな壮大なシステムを考えなくても、ブロックひとつだけを使っても、役に立つユースケースもあるので、恐れずいきましょう。

まずは、登場人物の紹介です。データフローブロックは IDataflowBlock というインターフェイスで表されます。このインターフェイスは、データフローブロックであるということ以外何も表していないので、もっと詳しく性質を表したインターフェイスを見ていきましょう。

ISourceBlock<TOutput>

TOutput 型のデータを出力するブロック

ITargetBlock<TInput>

TInput 型のデータを入力するブロック

IPropagatorBlock<TInput, TOutput>

`ITargetBlock<TInput>` と `ISourceBlock<TOutput>` 両方の性質を持ち、 TInput 型のデータを入力し、 TOutput 型に加工して出力するブロック

このようなシステムでは、データを出力（作成、送信）する側を Producer、データを受信する側を Consumer と呼んだりしますね。というのを図に表したのが、次の図です。

インターフェイスを紹介しましたが、これらのインターフェイスを自分で実装することはたぶんないと思います。メッセージの再送とかを実装する必要があり、お作法が結構大変です。既存のブロックの組み合わせで、 IPropagatorBlock を作成する方法については、公式ドキュメント「チュートリアル: カスタム データ フロー ブロックの型の作成」にあるので、必要があれば参照してください。

「いや、データを最初に送信する部分は自分で作る必要があるでしょ」というツッコミもあるかと思いますが、次に説明するデータの入力方法を使えば、ブロックとして作成する必要はないです。

最初の例: 入力データに対して処理を直列に行う

もっとも簡単かつ、これだけを単体で使っても価値があるブロックである ActionBlock<TInput> を最初の例として紹介します。 ActionBlock は TInput 型のデータを入力し、指定したデリゲートを実行する ITargetBlock です。

ActionBlock のコンストラクタの第1引数には、 `Action<TInput>` または `Func<TInput, Task>` を指定することができます。つまり async なメソッドでも良いということです。

ITargetBlock にデータを入力するには、 Post メソッドまたは SendAsync メソッドを使用します。これらは拡張メソッドとして提供されています。 Post と SendAsync の違いについては「バッファー上限と SendAsync」節で説明しますが、戻り値として `true` が返ってきたら、ブロックがデータを受理したということを表します。逆に `false` が返ってきたら、入力データが条件を満たしていない、ブロックがすでに完了状態のため新たなデータを受け付けない、といった理由で受理できなかったことを表します。

とにかく、 ActionBlock にデータを入力して、挙動を見てみましょう。次の例では、ActionBlock に 1 から 5 の整数を Post メソッドを使って入力しています。ActionBlock に渡すデリゲートは、入力値をログに残しつつ、 500 ミリ秒待機する処理です。

このコードでは、すべての入力値を入力し終わった後に Complete メソッドを呼び出しています。このメソッドを呼び出すことで、これ以上入力値はないということをブロックに伝えています。

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

var stopwatch = Stopwatch.StartNew();
void Log(string message) => Console.WriteLine($"[{stopwatch.ElapsedMilliseconds}

var actionBlock = new ActionBlock<int>(async i =>
{
    Log($"Start {i}");
    await Task.Delay(500);
    Log($"End {i}");
});

// 1〜5 を入力してみる
for (var i = 1; i <= 5; i++)
{
    bool postResult = actionBlock.Post(i);
    Log($"Post {i} => {postResult}");
}
```

```
// すべて入力しきったことを通知する
actionBlock.Complete();

// Completion プロパティは、すべての処理が完了したら結果が返ってくる Task
Task completion = actionBlock.Completion;
completion.Wait();
Log("すべて完了");
```

実行結果

```
[21ms] Post 1 => True
[28ms] Post 2 => True
[28ms] Post 3 => True
[28ms] Post 4 => True
[28ms] Post 5 => True
[31ms] Start 1
[533ms] End 1
[535ms] Start 2
[1035ms] End 2
[1035ms] Start 3
[1536ms] End 3
[1536ms] Start 4
[2036ms] End 4
[2036ms] Start 5
[2538ms] End 5
[2541ms] すべて完了
```

Wandbox で実行

実行結果から、 Post メソッドは呼び出してすぐに返ってきていること、処理は入力した順番に直列に行われていることがわかります。このことから、このような ActionBlock の使い方をすることで、入力値に対する処理を直列に行うというコードが簡単に書けることがわかると思います。

極端な例を挙げると、このようにすることで、あらゆる処理を直列に行うブロックの出来上がりです。

```
var actionBlock = new ActionBlock<Action>(x => x());

actionBlock.Post(() => { /* お好きな処理 */ });
```

データを変換する TransformBlock とリンク

次の基本的なブロックは、データの変換を行うブロックの TransformBlock<TInput,TOutput> です。 TInput 型のデータを入力し、 TOutput 型のデータを出力するデリゲートを、各入力データに対して実行してくれます。 LINQ の Select みたいなものです。ブロックの分類としては、入出力両方を行うので、IPropagatorBlock です。

ブロックの作り方は、デリゲートを渡すだけです。

```
new TransformBlock<int, int>(i => i * 100)
```

指定するデリゲートは、 `Func<TInput, TOutput>` の同期的なメソッドでも、 `Func<TInput, Task<TOutput>>` の非同期メソッドでも構いません。

TransformBlock へデータを入力するには、 ActionBlock と同様に、 Post メソッドや SendAsync メソッドを使用します（ITargetBlock 共通の拡張メソッドなので）。データを入力すると、そのうち（非同期なので入力した瞬間ではなく、そのうちです）出力が得られるようになります。出力を取得するには ReceiveAsync メソッドを使う方法もありますが、多くの場合、変換結果は次の処理へつなげることになると思います。つまり出力を、次のブロックの入力に転送してあげるように設定してあげることになります。このようにブロックの出力を、別のブロックの入力に接続することをリンクと呼び、 LinkTo メソッドを使って、リンクを設定します。 `sourceBlock.LinkTo(targetBlock)` で、 *sourceBlock* の出力が *targetBlock* の入力になります。

それでは、また 1〜5 の整数を TransformBlock に入力してみましょう。そして、その出力を ActionBlock に接続し、コンソールに表示します。

```
using System;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks.Dataflow;
```

```
var stopwatch = Stopwatch.StartNew();
void Log(string message) => Console.WriteLine($"[{stopwatch.ElapsedMilliseconds}

var transformBlock = new TransformBlock<int, int>(input =>
{
    // 入力を 100 倍して出力する
    var output = input * 100;
    Log($"Transform {input} => {output}");
    return output;
});

var actionBlock = new ActionBlock<int>(i =>
{
    // 表示するだけ
    Log($"Action {i}");
});

// transformBlock -> actionBlock という接続
transformBlock.LinkTo(actionBlock);

// 1〜5 を transformBlock に入力
for (var i = 1; i <= 5; i++)
    transformBlock.Post(i);

// すべて入力しきったことを通知する
transformBlock.Complete();

// 完了を待つ
Thread.Sleep(1000);

// 完了した？
Log(transformBlock.Completion.IsCompleted
    ? "transformBlock is completed"
    : "transformBlock is not completed");

Log(actionBlock.Completion.IsCompleted
    ? "actionBlock is completed"
    : "actionBlock is not completed");
```

実行結果

```
[46ms] Transform 1 => 100
[56ms] Transform 2 => 200
[56ms] Transform 3 => 300
[56ms] Transform 4 => 400
[56ms] Transform 5 => 500
[62ms] Action 100
[62ms] Action 200
[62ms] Action 300
[62ms] Action 400
[62ms] Action 500
[1040ms] transformBlock is completed
[1040ms] actionBlock is not completed
```

Wandbox で実行

出力については、予想通りに動きましたね。

さて、このプログラムでは、最後に各ブロックの `Completion.IsCompleted` をチェックしてみました。結果としては、 Complete メソッドを呼び出した *transformBlock* は完了状態になりましたが、そうでない *actionBlock* は完了状態になっていません。つまり、「完了した」ということは、このリンクでは伝わっていないということがわかります。

じゃあ、ソース側の完了をターゲット側にも通知したいときはどうするの？ というと、 LinkTo メソッドの引数にオプションを指定することで、完了したという情報もリンクされます。

```
transformBlock.LinkTo(actionBlock,
    new DataflowLinkOptions() { PropagateCompletion = true });
```

PropageteCompletion、名前通りですね。このように書き換えて実行すると、*actionBlock* も完了状態になることが確認できます。

完了、キャンセル、例外の伝搬

PropageteCompletion によって、完了をリンク先に通知できることがわかりました。しかし、ブロックは正常に完了するとは限らず、キャンセルが発生したり、指

定したデリゲート中で例外が発生したりすることもあります。そのときには、どのように通知されるのでしょうか？ 実験を行ってみたところ、次のような挙動をすることがわかりました。

ソースの完了状態	PropagateCompletion = false のときのターゲットの完了状態	PropagateCompletion = true のときのターゲットの完了状態
キャンセル	完了しない	ターゲットに同じ CancellationToken を指定 → キャンセル それ以外 → 正常に完了
例外	完了しない	例外

まとめると、 PropageteCompletion = true のとき、キャンセルや、例外を含めて伝搬します。ただし、ソースとターゲットの CancellationToken が異なる場合は、キャンセルではなく正常な完了として伝搬します。

ブロック中の SynchronizationContext

ActionBlock や TransformBlock のように、デリゲートを指定することができるブロックのコンストラクタには、 ExecutionDataflowBlockOptions を指定することで、実行方法のオプションを与えることができます。 ExecutionDataflowBlockOptions の TaskScheduler プロパティは、デリゲートを実行する Task をどの TaskScheduler で実行するかを表します。デフォルトは TaskScheduler.Default なので、 Task.Run と同様にスレッドプール上でデリゲートが実行されます。

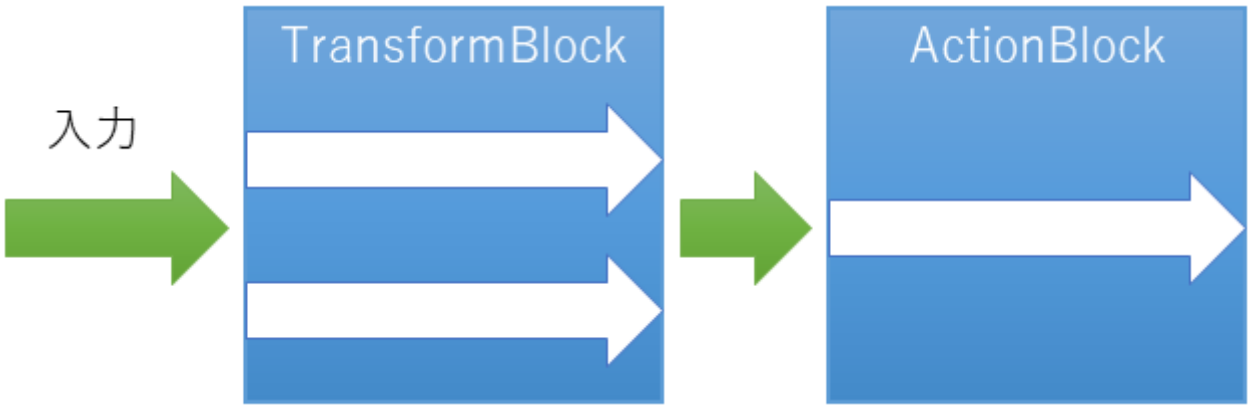
明示的に TaskScheduler を指定しない限り、デフォルトスケジューラで実行されるので、デリゲートが実行される環境の SynchronizationContext は null と想定することができます。これは、ブロックに渡すデリゲートで await を行うときに、ConfigureAwait のことは気にしなくてよいということです。

並列化

これまで直列に処理する例を見てきました。確かに、直列実行を行うだけの ActionBlock も使い道はあるのですが、 Task Parallel Library という名前の通り、並

列処理に使ってなんぼなわけです。というわけで、入力データに対して、並列に処理を行ってみましょう。

ここでは、 TransformBlock に並列実行を許可して、入力データに対して並列に処理を行ってもらいましょう。例えば、入力データが 5 件あり、それぞれの処理に 0.5 秒かかる場合、直列で実行したら 2.5 秒かかりますが、2つ同時に並列に実行したら 1.5 秒（$0.5 \times \lfloor 5 \div 2 \rfloor$）で終わるはずです。その結果を、直列実行する ActionBlock に転送し、コンソールに表示します。この例では、 TransformBlock にのみ並列実行を許可しますが、同様のやり方で並列実行する ActionBlock も作成することができます。



TransformBlock に並列実行を許可するには、コンストラクタの第2引数に渡す ExecutionDataflowBlockOptions で、 MaxDegreeOfParallelism プロパティ に並列実行数（2 以上）の値を指定します。もし、無制限に並列を許可（入力データがバッファーにある限り Task を作成する）するなら、 DataflowBlockOptions.Unbounded を指定します。つまりこんな感じ。

```
new TransformBlock<int, int>(i => i * 100,
    new ExecutionDataflowBlockOptions() { MaxDegreeOfParallelism = 2 })
```

それでは、実際に以上の設定で、試してみましょう。

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

var stopwatch = Stopwatch.StartNew();
void Log(string message) => Console.WriteLine($"[{stopwatch.ElapsedMilliseconds}
```

```
var transformBlock = new TransformBlock<int, int>(
    async input =>
    {
        Log($"Transform Enter {input}");

        // 処理に 0.5 秒かかるということに
        await Task.Delay(500);

        // 入力を 100 倍して出力する
        var output = input * 100;
        Log($"Transform Return {input} => {output}");

        return output;
    },
    new ExecutionDataflowBlockOptions()
    {
        // 並列数 2
        MaxDegreeOfParallelism = 2
    });

var actionBlock = new ActionBlock<int>(i => Log($"Action {i}"));

transformBlock.LinkTo(actionBlock, new DataflowLinkOptions() { PropagateCompleti

// 1〜5 を transformBlock に入力
for (var i = 1; i <= 5; i++)
    transformBlock.Post(i);

transformBlock.Complete();
actionBlock.Completion.Wait();
```

実行結果

```
[48ms] Transform Enter 1
[48ms] Transform Enter 2
[557ms] Transform Return 1 => 100
[557ms] Transform Return 2 => 200
[570ms] Transform Enter 3
```

```
[570ms] Transform Enter 4
[575ms] Action 100
[575ms] Action 200
[1070ms] Transform Return 4 => 400
[1070ms] Transform Return 3 => 300
[1073ms] Transform Enter 5
[1073ms] Action 300
[1077ms] Action 400
[1574ms] Transform Return 5 => 500
[1574ms] Action 500
```

Wandbox で実行

ほぼ同じ時刻に2つの「Transform Enter」が出力されていることから、2つ同時に処理が開始されていることが確認できました。そして、予想通り 1.5 秒で完了しました。

出力データの順番

さて、直前の例では、 ActionBlock のほうは直列で実行しましたが、ログには 100, 200, 300, 400, 500 と出力されています。つまり、 TransformBlock に入力した順番と同じ順番で ActionBlock に入力されているようです。これは順番が保証されているからでしょうか？ それとも偶然でしょうか？ 答えは、**デフォルトでは順番が保証されています。**

実際に実験して確認してみましょう。入力値 1, 2, 3, 4, 5 に対して、 1000, 800, 600, 400, 200 ミリ秒待機するような関数を指定し、並列数を無制限とすることで、必ず逆順で処理が完了するような TransformBlock を作成してみます。

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

var stopwatch = Stopwatch.StartNew();
void Log(string message) => Console.WriteLine($"[{stopwatch.ElapsedMilliseconds}

var transformBlock = new TransformBlock<int, int>(
    async input =>
```

```
    {
        Log($"Transform Enter {input}");

        /* 前回の例からの変更点 */
        // 入力値が大きいほど、待ち時間を短くする
        await Task.Delay(1200 - (input * 200));

        // 入力を 100 倍して出力する
        var output = input * 100;
        Log($"Transform Return {input} => {output}");

        return output;
    },
    new ExecutionDataflowBlockOptions()
    {
        /* 前回の例からの変更点 */
        // 無制限に並列実行を許可する
        MaxDegreeOfParallelism = DataflowBlockOptions.Unbounded
    });

var actionBlock = new ActionBlock<int>(i => Log($"Action {i}"));

transformBlock.LinkTo(actionBlock, new DataflowLinkOptions() { PropagateCompleti

// 1〜5 を transformBlock に入力
for (var i = 1; i <= 5; i++)
    transformBlock.Post(i);

transformBlock.Complete();
actionBlock.Completion.Wait();
```

実行結果

```
[49ms] Transform Enter 1
[49ms] Transform Enter 3
[49ms] Transform Enter 2
[55ms] Transform Enter 4
[55ms] Transform Enter 5
[255ms] Transform Return 5 => 500
```

```
[455ms] Transform Return 4 => 400
[652ms] Transform Return 3 => 300
[852ms] Transform Return 2 => 200
[1052ms] Transform Return 1 => 100
[1069ms] Action 100
[1069ms] Action 200
[1069ms] Action 300
[1069ms] Action 400
[1069ms] Action 500
```

Wandbox で実行

「Transform Return」は入力と逆の順なのに、「Action」は入力と同じ順番になっていることが確認できました。したがって、 TransformBlock での処理が完了したタイミングとは関係なく、入力順でブロックの出力が決まるということがわかりました。

この挙動は便利なときもあれば、非効率的なこともあります。順番を気にしなくてよいときには、前の入力の処理が完了を待つ時間が無駄ですからね。順番を気にせず、完了した順に出力する場合は、 TransformBlock のコンストラクタに渡すオプションで、 EnsureOrdered プロパティに `false` を指定します（デフォルトは `true`）。前の例に `EnsureOrdered = false` オプションを追加すると、このような出力になります。

```
[48ms] Transform Enter 2
[48ms] Transform Enter 3
[48ms] Transform Enter 1
[55ms] Transform Enter 4
[55ms] Transform Enter 5
[256ms] Transform Return 5 => 500
[271ms] Action 500
[452ms] Transform Return 4 => 400
[452ms] Action 400
[653ms] Transform Return 3 => 300
[653ms] Action 300
[849ms] Transform Return 2 => 200
[849ms] Action 200
[1049ms] Transform Return 1 => 100
[1049ms] Action 100
```

Wandbox で実行

EnsureOrdered はあらゆる種類のブロックのオプションとして設定できますが、効果があるのは今のところ TransformBlock と TransformManyBlock だけのようです。

バッファー上限と SendAsync

これまでの例では、ブロックに整数 1〜5 を Post メソッドを使って入力してきました。「最初の例」からもわかるように **Post メソッドは、入力を行い（ブロックの入力キューに積み）、すぐに返ってきます**。この動作は、簡単な例には持って来いですが、実際に利用する場面では、不便だったりします。

例えば、 Twitter のツイートに対して、何か処理を行うターゲットブロック *processTweetBlock* があるとします。このブロックに対して、タイムラインを遡りながらツイートを入力していこうとすると、このようなコードになります。

```
// 例であり、動作するプログラムではありません。
// また、 現在の Twitter API は page ではなく max_id を使用します。
for (var page = 0; ; page++)
{
    IEnumerable<Tweet> tweets = await GetTimelineAsync(page);
    foreach (var tweet in tweets)
        processTweetBlock.Post(tweet);
}
```

このとき、 Post メソッドはすぐに返ってくるので、ブロックの処理がどこまで完了しているかに関わらず、次のページを取得しに行ってしまいます。データソースからプル型で取得してくる場合、ブロックの処理状況に応じたペースで取得しに行くほうが、合理的です。

今までの例では、 Post メソッドを呼び出せば、必ずブロックがその入力を受け入れてくれました。しかし、このように流量制限を行いたい場合、ブロックが「まだダメだ。待ってくれ」と言ってくれると都合がよいです。そのように入力できるデータ数に制限をかけるのが、 BoundedCapacity オプションです。デフォルトは無制限（`DataflowBlockOptions.Unbounded`）になっており、ブロックはメモリが許す限りいくらでも入力を受け付けますが、正の整数を指定することで、制限を設定できま

す。例えば、入力を 1 件に制限した ActionBlock を作成するには、次のように指定します。

```
new ActionBlock<T>(x => { },
    new ExecutionDataflowBlockOptions() { BoundedCapacity = 1 })
```

BoundedCapacity は、そのブロックの **入力データ数 + 処理中データ数 + 出力データ数** に対する上限です。入力データ数とは、 Post、SendAsync による入力や、リンクされたブロックから入力され、並列数の制限により、まだ処理が開始されていないデータの件数を指します。出力データ数とは、リンク先ブロックが BoundedCapacity の上限に達していて、まだ送信できていないデータの件数を指します（リンク先がないソースブロックの場合には、 Receive を呼び出さない限り出力データが溜まります）。

BoundedCapacity には処理中のデータ数が含まれます。つまり、 BoundedCapacity を設定するとき **BoundedCapacity ≧ MaxDegreeOfParallelismでなければ、せっかく並列数を指定しても意味がない**ということに、気を付けてください。

BoundedCapacity による上限を超えたとき、 Post メソッドで値を入力すると、`false` が返ってきます。つまり、入力に失敗します。キューに積まれるということではなく、単に受け付けないのです。

例えば、入力に対して 0.5 秒かかる処理を行う、 BoundedCapacity が 1 の ActionBlock に、 2 件のデータを連続して入力すると、 2 件目は一切処理されません。

```
using System;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

var actionBlock = new ActionBlock<int>(
    async i =>
    {
        Console.WriteLine($"Action {i}");
        await Task.Delay(500);
    },
    new ExecutionDataflowBlockOptions() { BoundedCapacity = 1 });
```

```
for (var i = 1; i <= 2; i++)
{
    bool postResult = actionBlock.Post(i);
    Console.WriteLine($"Post {i} => {postResult}");
}

actionBlock.Complete();
actionBlock.Completion.Wait();
```

実行結果

```
Post 1 => True
Post 2 => False
Action 1
```

Wandbox で実行

では、ブロックが新たなデータを受け入れられるようになるまで待機し、そして入力を行うようにするにはどうすればよいのでしょうか？ これをうまくやってくれるのが SendAsync メソッドです。 SendAsync メソッドの戻り値は `Task<bool>` で、この Task はブロックが入力を受け入れたときに完了します。前に挙げたツイートを処理する例は、 *processTweetBlock* に BoundedCapacity を指定し、次のように書き換えれば、適切に流量制限をかけることができます。

```
for (var page = 0; ; page++)
{
    IEnumerable<Tweet> tweets = await GetTimelineAsync(page);
    foreach (var tweet in tweets)
        await processTweetBlock.SendAsync(tweet);
}
```

ブロックが受け入れるまで待機するのに、それでも戻り値が bool なのは、BoundedCapacity による制限以外にも失敗する理由があるからです。それは主に、ブロックが完了状態である場合です。

ググって出てくる TPL Dataflow の情報では、 Post と SendAsync の違いについて、あまり丁寧に説明されていないので、これだけは覚えて帰ってください:

Post メソッドは、 BoundedCapacity による上限が設定されている場合、または後で説明する最短一致モードの場合、入力に失敗することがあります。もし、ブロックが受け入れてくれるまで待機する必要があるならば、 SendAsync メソッドを使用します。

複数のブロックに同じデータを送信する

あるブロックの出力を、複数のブロックに送信したいということもあると思います。このとき、単に LinkTo を 2 回呼び出せばよいのでしょうか？ 残念ながら、これでは最初にリンクしたブロックにだけデータが送信されます。例えば、次の例では *fooBlock* にのみ送信されていることが確認できます。

```
using System;
using System.Threading.Tasks.Dataflow;

var transformBlock = new TransformBlock<int, int>(
    i => i * 100 // 入力を 100 倍して出力する
);

var fooBlock = new ActionBlock<int>(i => Console.WriteLine($"Foo {i}"));
var barBlock = new ActionBlock<int>(i => Console.WriteLine($"Bar {i}"));

// transformBlock から fooBlock と barBlock 両方にリンク
transformBlock.LinkTo(fooBlock, new DataflowLinkOptions() { PropagateCompletion
transformBlock.LinkTo(barBlock, new DataflowLinkOptions() { PropagateCompletion

for (var i = 1; i <= 5; i++)
    transformBlock.Post(i);

transformBlock.Complete();
fooBlock.Completion.Wait();
barBlock.Completion.Wait();
```

実行結果

```
Foo 100
Foo 200
Foo 300
```

```
Foo 400
Foo 500
```

Wandbox で実行

プログラムが終了することから、 *fooBlock*、*barBlock* 共に完了の伝搬だけはできているようです。このように、最初のリンクにのみデータが送信されるのには理由があり、リンクには条件を指定できるからです。条件を満たさなかったデータは、次のリンクに送信されます。詳しくは、 LinkTo メソッドの条件を引数に取るオーバーロードや、リンクオプションの MaxMessages プロパティについて調べてみてください。

話は戻って、複数のブロックに同じデータを送信したいという需要に応えてくれるのが、 BroadcastBlock<T> です。このブロックだけは、リンクが特殊な挙動をし、すべてのリンクに入力データを送信します。ただし、ドキュメントには「一度に最大で 1 個の要素を格納し、各メッセージを受信した次のメッセージで上書きするバッファーを提供します。」とあります。**最大 1 個の最新の要素を格納するということは、リンク先がデータを受け入れるか否かに関係なく、 BroadcastBlock が持つデータは入力によって更新されてしまう**ということです。したがって、リンク先の BoundedCapacity が無制限ならうまく機能しますが、そうでなければ予想と反する挙動を示すでしょう。

したがって、前の例は、 BoundedCapacity が無制限なので、 BroadcastBlock を用いて、次のように書き換えることができます。

```
using System;
using System.Threading.Tasks.Dataflow;

var transformBlock = new TransformBlock<int, int>(
    i => i * 100 // 入力を 100 倍して出力する
);

// BroadcastBlock の第1引数は、各リンク先にデータを送信するときに、クローンを行うための関
// ディープコピーを行いたいとか特殊な事情がない限り、気にする必要はないので、 null を指定
var broadcastBlock = new BroadcastBlock<int>(null);

var fooBlock = new ActionBlock<int>(i => Console.WriteLine($"Foo {i}"));
var barBlock = new ActionBlock<int>(i => Console.WriteLine($"Bar {i}"));
```

```
// transformBlock -> broadcastBlock
transformBlock.LinkTo(broadcastBlock, new DataflowLinkOptions() { PropagateCompl

// broadcastBlock -> {fooBlock, barBlock}
broadcastBlock.LinkTo(fooBlock, new DataflowLinkOptions() { PropagateCompletion
broadcastBlock.LinkTo(barBlock, new DataflowLinkOptions() { PropagateCompletion

for (var i = 1; i <= 5; i++)
    transformBlock.Post(i);

transformBlock.Complete();
fooBlock.Completion.Wait();
barBlock.Completion.Wait();
```

実行結果

```
Bar 100
Foo 100
Bar 200
Foo 200
Bar 300
Foo 300
Bar 400
Foo 400
Bar 500
Foo 500
```

Wandbox で実行

しかし、前述の通り、流量制限を行うには BroadcastBlock は使用できません。そこで次のような拡張メソッドを作成して、それっぽい挙動を再現してみます。

```
static IDisposable LinkMany<T>(this ISourceBlock<T> source, params ITargetBlock<
{
    IDisposable linkDisposable = null;

    var linkBlock = new ActionBlock<T>(
        async item =>
```

```
    {
        // targets すべてに対して SendAsync して、送信し終わるのを待つ
        var tasks = targets.Select(target => target.SendAsync(item));
        var results = await Task.WhenAll(tasks).ConfigureAwait(false);

        // すべてに失敗したならば、これ以上受け入れる必要がない
        if (results.All(x => x == false)) linkDisposable?.Dispose();
    },
    new ExecutionDataflowBlockOptions()
    {
        // ターゲット側の流量制限にまかせて、ここでは1つしか値を持たない
        BoundedCapacity = 1
    });

linkDisposable = source.LinkTo(linkBlock, new DataflowLinkOptions() { Propag

// 完了を伝搬
linkBlock.Completion.ContinueWith(
    sourceCompletion =>
    {
        Exception exception = null;
        if (sourceCompletion.IsFaulted)
        {
            AggregateException aex = sourceCompletion.Exception;
            exception = aex.InnerExceptions.Count == 1
                ? aex.InnerException
                : aex;
        }

        foreach (var target in targets)
        {
            if (exception == null)
                target.Complete();
            else
                target.Fault(exception);
        }
    },
    // 忘れがちだけど、 ContinueWith に TaskScheduler を指定しないと Current にな
    TaskScheduler.Default);
```

```
    return linkDisposable;
}
```

この拡張メソッドを利用して、 BoundedCapacity が設定された 2 つのブロックにデータを送信する例は、こちらをご覧ください。

入力データをまとめて配列にする BatchBlock

入力データが何件か溜まったら、一気に処理を行う、みたいな構成にすることで、効率的に行うことができる処理もあります。そこで、入力データを何件か溜めることを行ってくれるのが BatchBlock<T> です。 BatchBlock は `IPropagatorBlock<T, T[]>` を実装しており、出力は配列になります。コンストラクタで、何件まとめるのかを指定します。

次の例では、入力データを 3 件ずつまとめています。入力データを 1〜5 の整数とすると、 1, 2, 3 がまず出力されます。 4, 5 については、これ以上入力データがないとわかったら出力されます。

```
using System;
using System.Threading.Tasks.Dataflow;

// 3 件まとめる
var batchBlock = new BatchBlock<int>(3);

// カンマ区切りで表示
var actionBlock = new ActionBlock<int[]>(
    inputs => Console.WriteLine(string.Join(", ", inputs)));

batchBlock.LinkTo(actionBlock, new DataflowLinkOptions() { PropagateCompletion =

// 1〜5 を入力
for (var i = 1; i <= 5; i++)
    batchBlock.Post(i);

batchBlock.Complete();
actionBlock.Completion.Wait();
```

実行結果

```
1, 2, 3
4, 5
```

Wandbox で実行

また、 TriggerBatch メソッドを呼び出すことで、指定した件数の入力データが溜まっていなくても、 1 件以上あれば、出力を行います。

BoundedCapacity を指定する場合、入力データ数は BoundedCapacity の計算に含まれることから、コンストラクタで指定した件数以上の数値を指定しなければいけません（コンストラクタが例外をスローします）。

最短一致モード、最長一致モード

公式ドキュメントを読んでいると、 BatchBlock、JoinBlock、BatchedJoinBlock において、最短一致モード(non-greedy mode)、最長一致モード(greedy mode)という表現が出てきます。これは、 GroupingDataflowBlockOptions.Greedy プロパティのことを指しています。デフォルトは `true` で最長一致モードになっています。

デフォルトの最長一致モードでは、 BoundedCapacity が許す限り入力を受け入れます。おそらくこれが、普通予想される挙動だと思います。

一方、最短一致モードでは、基本的に入力を延期します。つまり Post すると `false` が返ってきます。ブロック内部では、延期した入力を記録しています。そしてその記録がバッチとしてまとめる件数分だけ溜まって、初めて入力を要求します。（正直、 BatchBlock で最短一致モードを使うことはないと思います。JoinBlock ならあるかな？ そもそも JoinBlock 自体の良い使い道が思い浮かんでいないので、この記事では扱わない予定ですが。）

SendAsync を使って、入力が受け入れられるタイミングを見てみましょう。次のプログラムでは、 SendAsync を呼び出した時刻と、入力が受け入れられた時刻を記録することで、タイミングを調べられるようになっています。

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
```

```
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

GreedyTest(true);
GreedyTest(false);

void GreedyTest(bool greedy)
{
    Console.WriteLine($"Greedy = {greedy}");

    var stopwatch = Stopwatch.StartNew();
    void Log(string message) => Console.WriteLine($"[{stopwatch.ElapsedMilliseco

    // 引数によって Greedy を変える
    var batchBlock = new BatchBlock<int>(3,
        new GroupingDataflowBlockOptions() { Greedy = greedy });

    var actionBlock = new ActionBlock<int[]>(
        inputs => Log("Action " + string.Join(", ", inputs)));

    batchBlock.LinkTo(actionBlock, new DataflowLinkOptions() { PropagateCompleti

    var sendTasks = new List<Task>();

    // 0.1 秒ずつ 1～5 を送信する
    for (var i = 1; i <= 5; i++)
    {
        sendTasks.Add(SendToBatchBlockAsync(i));
        Thread.Sleep(100);

        async Task SendToBatchBlockAsync(int item)
        {
            // 送信開始と送信完了の時刻をログに残す
            Log($"Sending {item}");
            bool sendResult = await batchBlock.SendAsync(item);
            Log($"Sent {item} => {sendResult}");
        }
    }
```

```
    // 入力終わり
    Log("Complete");
    batchBlock.Complete();

    actionBlock.Completion.Wait();
    Task.WaitAll(sendTasks.ToArray());

    Console.WriteLine();
}
```

最長一致モードの結果

```
Greedy = True
[14ms] Sending 1
[27ms] Sent 1 => True
[127ms] Sending 2
[127ms] Sent 2 => True
[227ms] Sending 3
[230ms] Sent 3 => True
[234ms] Action 1, 2, 3
[331ms] Sending 4
[331ms] Sent 4 => True
[431ms] Sending 5
[431ms] Sent 5 => True
[533ms] Complete
[537ms] Action 4, 5
```

最短一致モードの結果

```
Greedy = False
[8ms] Sending 1
[116ms] Sending 2
[217ms] Sending 3
[234ms] Action 1, 2, 3
[235ms] Sent 2 => True
[235ms] Sent 3 => True
[235ms] Sent 1 => True
[320ms] Sending 4
[420ms] Sending 5
```

```
[520ms] Complete
[525ms] Sent 4 => False
[525ms] Sent 5 => False
```

Wandbox で実行

最長一致モードでは、「Sending」と「Sent」がほぼ同時で、すぐに受け入れられていることがわかります。一方、最短一致モードでは、 3 件まとめた配列が *actionBlock* に入力されるのと同時に「Sent」が表示されています。このことから、必要な数の入力が集まるまで延期する、ということの意味がわかると思います。また、最短一致モードでは、最後の 2 件は *actionBlock* に送信されていないのも特徴的です。

とにかく、最短一致モードは癖が強いよ、ということです。

ちょっと大規模なサンプル: マネージドキューサービスからジョブを取り出し、実行するワーカー

経緯

この記事は、 TPL Dataflow をもっと早く知っていれば良かったという後悔によって書かれています。以前、仕事で、マネージドキューサービスである Amazon SQS を使ったことがありました。経緯としては、ジョブが溜まるペースの予想がつかないので、とりあえずマネージドキューサービスにジョブを溜め、そのジョブを実行するワーカーは、キューの溜まり具合を見ながら増やしていこうという作戦でした。そこで、ワーカー側の実装を行ったのですが、ワーカー 1 プロセス内で並列に処理を行えるようにするために、 Microsoft.VisualStudio.Threading を駆使して、なかなか渋いコードを書きました。 TPL Dataflow を学んで、このコードをもっと安全に書き直せるなと思いました（下手に自作するとミスりがちな部分がライブラリとして提供されているので）。というわけで、このようなシステムを題材にサンプルを用意しました。

Amazon SQS は、リクエストを投げると、キューの中身（メッセージ）が返ってくる、いわゆるプル型のサービスです。あるワーカーがメッセージをキューから取得すると、そのメッセージは他のワーカーからは一定時間取得できなくなるので、メッセージを取得したワーカーは、他のワーカーからそのメッセージが見えなくなっ

ている間に処理を行い、処理が完了したら「完了したからキューから削除してくれ」とリクエストを送信します。キューからメッセージの取得、およびメッセージの削除は 10 件までまとめて行うことができます。こんな感じのマネージドキューサービスを想定して、そこからメッセージを取得して、並列に処理を行うプログラムを作成します。

マネージドキューサービスのモデル

僕個人では AWS のアカウントを持っていないので、ここでは実際に AWS へアクセスせず、それっぽいものを用意します。

それっぽければなんでもいいの精神で雑にやっていきます。メッセージはとりあえず ID だけ付与できるようにしました。

```
public class QueueMessage
{
    public int MessageId { get; }
    public QueueMessage(int messageId) => this.MessageId = messageId;
}
```

キューサービスは、メッセージの取得（ReceiveMessagesAsync）と削除（DeleteMessagesAsync）ができます。

```
public class QueueService
{
    private static readonly Random s_rng = new Random();
    private int _messageId;

    public async Task<IReadOnlyList<QueueMessage>> ReceiveMessagesAsync(Cancella
    {
        // 取得に 100〜600ms かかるものとする
        await Task.Delay(100 + s_rng.Next(500), cancellationToken).ConfigureAwai

        // 1〜10 件取得できる
        var results = new QueueMessage[s_rng.Next(10) + 1];
        for (var i = 0; i < results.Length; i++)
            results[i] = new QueueMessage(++_messageId);
        return results;
```

```
    }

    public Task DeleteMessagesAsync(IEnumerable<QueueMessage> messages)
    {
        // 削除に 100ms かかるものとする
        return Task.Delay(100);
    }
}
```

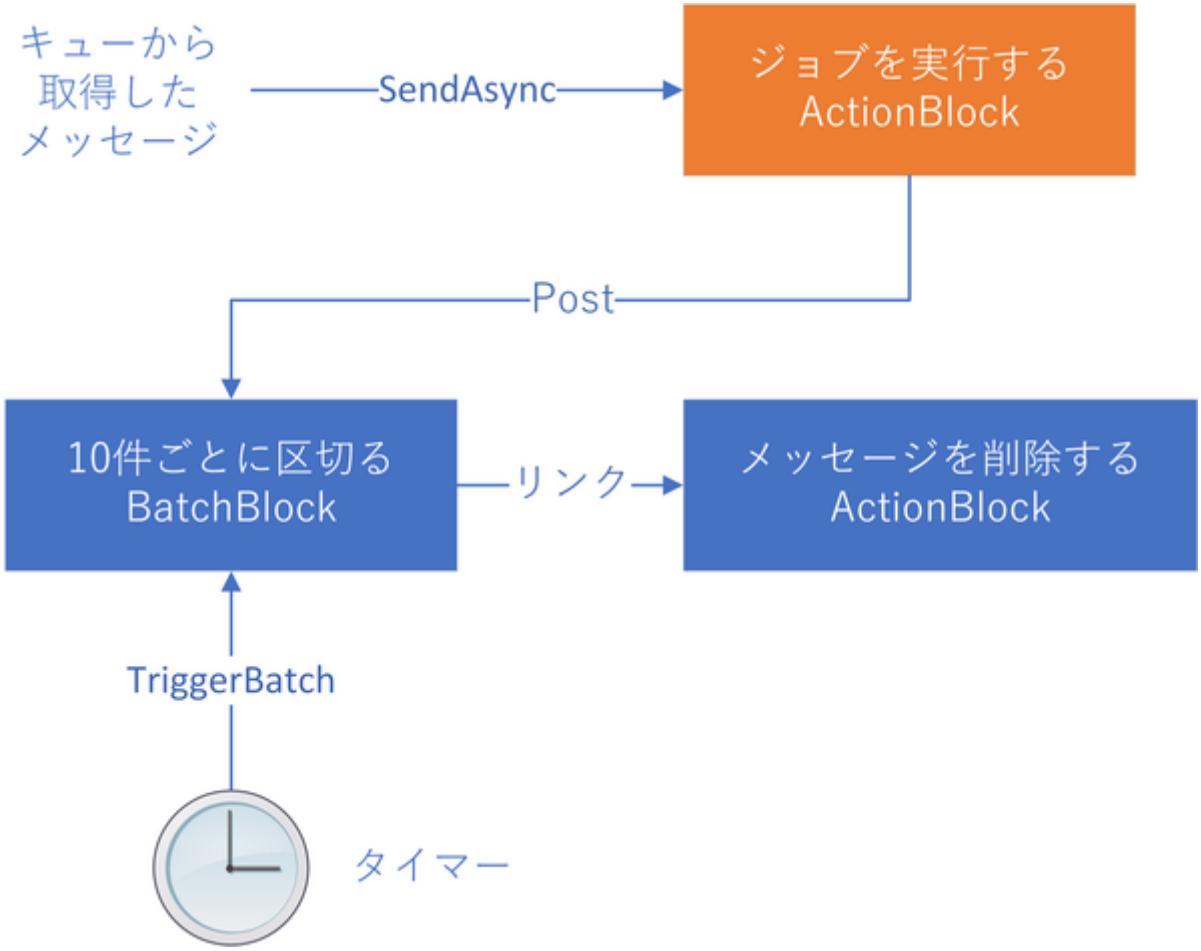
このモデルを相手に、メッセージを並列で処理していけるようにします。

データフローブロックを組んでいく

メッセージを処理する部分のブロックは非常に簡単で、並列数を指定した ActionBlock を作ればよいだけです。キューからの取得については、プル型なので、Twitter の例で示したものと同じことをすればよいのです。 BoundedCapacity は並列数の 2 倍とします。並列数分だけ余分に入力を許可することで、キューから取得しにいくのが遅すぎて、処理効率が落ちるのを防ぐためです。

問題は、完了したメッセージの削除です。 Amazon SQS では 10 件まとめて削除ができますし、 API コール回数は少ないほうが安くなります。というわけで 10 件を溜めたいわけですが、 10 件溜まるのってどのくらい時間がかかるのでしょうか？ 10 件溜まるまでに時間がかかりすぎて、他のワーカーが同じメッセージを取得してしまったら、同じ処理を行ってしまうので効率が非常に悪くなります*1。そこで、 5 秒ごとに、必ず削除を行うようにします。さすがにタイマーは TPL Dataflow の範囲ではないので、自分で頑張る必要があります。しかし、先程紹介したように、BatchBlock.TriggerBatch メソッドという便利なものがあるので、なんとかなりそうという気はします。

そんな感じで、必要なデータブロックをまとめたのが、次の図です。



さて、まだ考えないといけないことがあります。それは、キャンセルや例外によって、終了するときです。キューからの取得部分や、メッセージを処理する部分に対しては CancellationToken を用いることで、キャンセルを通知できます。しかし、メッセージの削除処理については、キャンセルが発生して、処理中のジョブがすべて終わったときに、メッセージの削除キューを流しきる必要があります。つまり、図でいう「ジョブを実行する ActionBlock」の Completion を監視して、この ActionBlock が成功しようが失敗しようが、メッセージ削除のための BatchBlock を Complete にして、実際にメッセージの削除が完了するまで待機する必要があります。ざっくり書くと、以下のような感じですが、実際には例外処理を含めて、もう少し複雑になります。

```
Task completion = workBlock.Completion
    .ContinueWith(async workBlockCompletion =>
    {
        // すべての削除が完了するのを待つ
        deleteMessageBatchBlock.Complete();
        await deleteMessageWorkBlock.Completion.ConfigureAwait(false);

        return workBlockCompletion;
```

```
    }, TaskScheduler.Default)
    .Unwrap().Unwrap();
```

完成形

完全なソースコードと実行結果は Wandbox にあります。並列数の制御とか ResetEvent を使ったりとかせずに、キューからの取得の流量制御や、並列実行が実現できているということを感じていただければ幸いです。

類似品との比較

Parallel, PLINQ

Parallel クラスおよび PLINQ は、（主観ですが） TaskAsync が普及する前に書かれたプログラムを並列化するためのライブラリです。例えば `for (int i = 0; i < len; i++)` において、 `i` のときの計算が `i - 1` のときの計算結果に依存しないならば、並列化出来るよね、とか、 LINQ の Select に渡すデリゲートが参照透過性を持っているならば並列化できるよね、とかそういう感じです。

Parallel, PLINQ は、内部では Task を使用し、スレッドプールに計算をスケジュールしています。しかし、それらを行った結果が Task で返ってくるわけではありません。つまり、計算が完了するまで、呼び出し元スレッドをブロックします*2。コンソールアプリのメインスレッドや、自分で作成した生の Thread 上で動くコードから呼び出す分には、ブロッキングは問題ではありません。しかし、スレッドプール上で動くプログラムでブロッキングが発生するのは、スレッドプールの利用効率上問題があります。 TaskAsync が多用される現在のプログラミングスタイルにおいて、どのスレッドで呼び出されるかわからないプログラムが、スレッドをブロックする可能性があるのは、良い設計ではありません。したがって、現在 Parallel, PLINQ を利用できる場面は限られていると思います。

また、 Parallel, PLINQ では、引数に渡すデリゲートを async メソッドにすることはできません。これは、同様に TaskAsync 多用時代にマッチした設計ではありません。 IAsyncEnumerable が本格的に用いられるようになったら、 PLINQ が改良されるかもしれません。

まとめると、どのスレッドで呼び出されるかわからないプログラムで Parallel, PLINQ を使用するのはおすすめできません。 TPL Dataflow は PLINQ ほどの簡単さを持ってはいませんが、できれば TPL Dataflow を使用することをおすすめします。

Reactive Extensions

TPL Dataflow において、 `ISourceBlock<T>` は `IObservable<T>` に、 `ITargetBlock<T>` は `IObserver<T>` に、それぞれ AsObservable 拡張メソッド、AsObserver 拡張メソッドを用いて変換できます。

見かけ上変換はできるのですが、 TPL Dataflow と Rx は異なる概念であることを覚えておいてください。 **Rx はプッシュ型であるのに対して、 TPL Dataflow は、プッシュ型とプル型両方の性質を持っています。**

Rx では、データの作成は完全に Producer に任されており、新たなデータがどんどんプッシュされてきます。一方 TPL Dataflow では、プッシュされてきたデータの受信を延期し、後で受信者が要求するという手段を用意しています。この延期という概念を導入することによって「バッファー上限と SendAsync」節で紹介した流量制御を、スレッドのブロッキングなしに実現することができています。

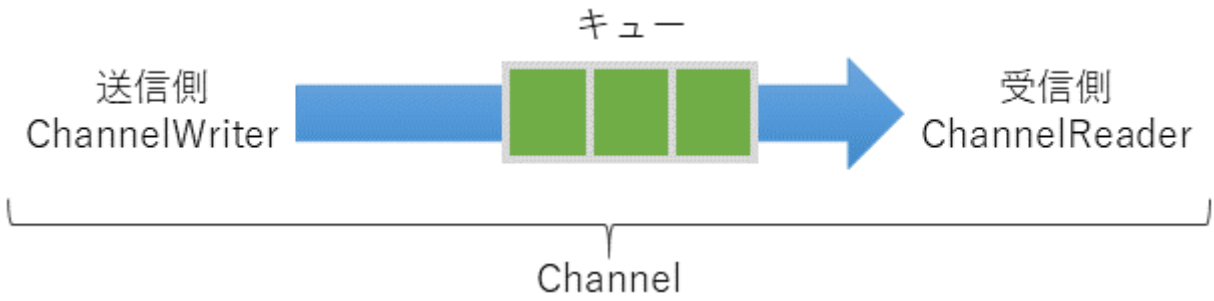
Rx と TPL Dataflow を両方利用した運用は、有効な手段です。並列化と流量制限が必要な部分をデータフローブロックを用いて作成し、その最後のブロックを AsObservable メソッドで IObservable に変換し、その後の処理を Rx で行うことで、すっきりしたコードになるかもしれません。ただし、 Rx で作成したデータを、AsObserver したブロックに入力するのには注意が必要です。もし、そのブロックに BoundedCapacity が設定されている（流量制限がある）場合、 IObserver の OnNext メソッドは SendAsync が完了するまでブロッキングを行います。

System.Threading.Channels

Channel<T> クラスは、 TPL Dataflow における BufferBlock<T> と同じ概念です。BufferBlock は、基本的ブロックすぎて、逆に例として示すのが難しかったので、ここまでこの記事に登場していませんが、言ってしまえば何もしない TransformBlock、すなわち `new TransformBlock<T, T>(x => x)` と同じ動作をします。

ここで、軽くチャンネルの説明をしておきます。チャンネルは、並行・並列処理において、安全にデータの受け渡しを行うための、データの通り道です。キューとロ

ックを組み合わせて実現されています。



送信側がチャンネルにデータを書き込むと、キューにデータが積まれます。受信側がチャンネルにデータを要求すると、キューからデータがデキューされます。

特徴的なのは、キューに一切データがないとき、またはキューが一杯（設定した上限に達した）ときの動作です。キューに一切データがないときに、受信しようとすると、何か送信されるまで待機します。ただし System.Threading.Channels は TaskAsync 世代のライブラリなので、待機をブロッキングではなく Task で表現します。逆に、キューが一杯の時に、送信しようとすると、受信側がデキューしに来るまで待機します。

チャンネルは、受信側から見たら、プル型の概念のです。送信側から見た場合は、プッシュ型ですが、もし、キューが一杯ならば待機を行う点で、少し特殊です。これは Rx との比較のときに TPL Dataflow の特徴として挙げた点と同じです。したがって、チャンネルとは TPL Dataflow を支える概念ということです。そして、System.Threading.Channels は、その部分だけをシンプルな形で取り出したものです。 TPL Dataflow とは、チャンネルの上に、並列処理を行うにあたって便利な機能をブロックという形で提供したライブラリである、と考えるとよいでしょう。

まとめ

ここまでずっと「並列」の話をしてきましたが、並列自体は Task によって簡単に表現できるようになりました。しかし、並列を支える周辺の「並行」の処理、すなわち、何が何を待機しなければいけないのか、みたいな制御をバグなく書き上げるのは大変な作業です。 TPL Dataflow は、そこをうまくライブラリとして提供することで、利用者は、ただ async/await でプログラムを書くだけでよくなります。安全に並列処理を書きあげるために、ぜひ TPL Dataflow を活用してみてください。

*1: 同じメッセージを複数のワーカーが処理してしまったら、不整合が起こるのでは？ と思うかもしれませんが、そもそも Amazon SQS では、メッセージが 1 度だけしか取得できないということは保証されておらず、複数回同じメッセージを受信するということを想定してシステムを作らなければいけません。

*2: 最初の 1 並列分だけは、呼び出し元スレッドで計算が行われます。しかし、それが完了してしまったら、後は他のスレッドが完了するのを待機します。

azyobuzin (id:azyobuzin) 1年前

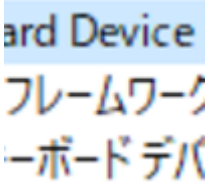
関連記事

2019-12-24
私、ValueTaskは限定的でって言ったよね！

メリークリスマス！ みなさんは、見てないアニメをネタにするな…

2019-11-12
比較的安全に Docker で Pleroma サーバーを建てる

みんな大好き「やってみた」、今回は Pleroma サーバー構築をや…



2018-07-26
はじめての仮想HID

半年ぶりです。崇高な計画を遂行するために、仮想 HID が欲しく…

2017-01-04
MySQL Connector/.NET の TreatTinyAsBoolean のバグと Dap…

MySQL Connector/.NET、長いので MySql.Data と呼びます。 バグ…

2015-12-12

非同期処理のデータ伝達を楽にする System.Threading.Tasks.Channels

corefxlab Advent Calendar 12 日目です。頓挫しかしてねえ！！…

long

TPL DataFlowは私も最近までチャレンジしていました。結局TPL DataFlow向けの1点もののコードを書く羽目になったのが大変でした。
何より期待外れだったのは、データフローをいい感じに出来る触れ込みなのに、ほとんど可視化が不可能な所にあります。結局レコード1件ずつを追跡する仕組みはなく、バッファの内部の個数を取得する事しかできませんでした。
LINQPadはそこそこ可視化できますが、実装を見るとかなり限界まで頑張っているみたいです。
何か良い可視化方法がありますでしょうか。

1年前 ★+

azyobuzin (id:azyobuzin)

> 結局TPL DataFlow向けの1点もののコードを書く羽目になったのが大変でした。

再利用できるものが書きやすいかというと、そんなことは全然ないですよね。

使い道としては、記事で挙げたように、一部並列、一部直列のように、並列数の制御が入り乱れるところに Dataflow を使うことで、楽に制御ができるという点で使うべきかと思います。

あとは、再利用を考えたとき、ひとつの DataflowBlock を作ると考えるよりは、`DataflowBlock.Encapsulate` した結果を返す関数として定義したほうが扱いやすくなると思います。

> LINQPadはそこそこ可視化できますが、実装を見るとかなり限界まで頑張っているみたいです。

LINQPad がここでも有能だとは知らなかった！

LINQPad の実装を読んできましたが、各ブロックのクラスには、 `DebugView` という子クラスが提供されていて、それでキューの状態を調べられるのですね。とすると、流れるデータを確認したいリンクの間に `TransformBlock` を挟んで、そのブロックの処理として、前後のブロックの `DebugView` の中身を読み出す、とすれば、いい感じにログになりそうな気はします。そこまで大掛かりにな仕掛けはしたくないですけどね。でも、大規模に使うならメトリクスとか取りたいですよね……。

1年前 ★+

コメントを書く

« バタフライ図も行列も見たくない人のため…

Kubernetesで隔離Mastodonネットワークを… »

プロフィール

azyobuzin (id:azyobuzin)

読者になる 56

検索

記事を検索

カテゴリー

Twitter (7)

ASP.NET (4)

Nemerle (2)

Android (17)

Go (2)

ネタ (5)

思想 (10)

はてなブログ (4)

JavaScript (4)

レビュー (6)

iOS (5)

Python (12)

Windows8 (6)

Xtend (2)

Krile (5)

Office (1)

LINE (1)

WPF (4)

羽毛 (1)

Visual Basic (3)

ウォーキング (9)

Sandcastle (5)

Git (1)

Scala (1)

月別アーカイブ

- 2020 (1)
  - 2020 / 2 (1)
- 2019 (7)
- 2018 (5)
- 2017 (8)
- 2016 (13)
- 2015 (27)
- 2014 (25)
- 2013 (55)
- 2012 (39)
- 2011 (35)
- 2010 (21)

 アジョブジ星通信