

まとめ

- 非同期で複数のデータを受け渡し
- C# 8.0の新機能
 - 非同期メソッドの拡張 : `await`と`yield`の混在
 - 非同期`foreach` : `await foreach (var x in asyncStream)`
- パフォーマンスへの配慮
 - `ValueTask`
 - 本当に非同期の時だけアロケーション
 - `IValueTaskSource`/`ManualResetValueTaskSourceCore`
 - 「1度に1人だけ`await`」な前提で最適化

C# 8.0 非同期ストリーム

++C++; // 未確認飛行 C

岩永 信之

今日の話

- C# 8.0の非同期ストリーム
- 非同期メソッドやTaskクラス周りの歴史
 - パフォーマンス改善

C# 8.0の他の機能

- Preview版の頃から割と安定していた機能は4月の登壇を参照
Visual Studio 2019 Launch

(<https://connpass.com/event/122145/>)

C# 8.0 Preview in Visual Studio 2019 (16.0)

(<https://www.slideshare.net/ufcpp/c-80-preview-in-visual-studio-2019-160>)

- null許容参照型は先月の登壇を参照

Visual Studio Users Community Japan #1

(<https://vsuc.connpass.com/event/143114/>)

C# 8.0 null許容参照型

(<https://www.slideshare.net/ufcpp/c-80-null>)

C# 8.0の他の機能

- Preview版の頃から割と安定していた機能は4月の登壇を参照
Visual Studio 2019 Launch
(<https://connpass.com/event/122145/>)
C# 8.0 Preview in Visual Studio 2019 (16.0)
(<https://www.slideshare.net/ufcpp/c-80-preview-in-visual-studio-2019-160>)
- null許容参照型は先月の登壇を参照
Visual Studio Users Community Japan #1
(<https://vsuc.connpass.com/event/143114/>)
C# 8.0 null許容参照型
(<https://www.slideshare.net/ufcpp/c-80-null>)

非同期ストリームとは

非同期で複数のデータを受け渡し

非同期ストリームとは

- 非同期で複数のデータを受け渡し

非同期ストリームとは

- 非同期で複数のデータを受け渡し
 - 非同期メソッドでyieldを使える = 複数のデータを非同期に返す

```
await Task.Delay(1);  
yield return 1;
```

awaitとyieldの混在

非同期ストリームとは

- 非同期で複数のデータを受け渡し
 - 非同期メソッドで `yield` を使える = 複数のデータを非同期に返す

```
await Task.Delay(1);  
yield return 1;
```

awaitとyieldの混在

- `foreach` の非同期版が使える = 複数のデータを非同期に受け取る

```
await foreach (var item in source)  
{  
}
```

await foreach構文

非同期ストリームの例

- gRPCを例にして非同期ストリームを紹介
 - サンプル: [NetCoreGrpc](#)
- 背景
 - gRPCは非同期ストリームを送受信する想定を持っている
 - `stream`キーワードを付けると非同期ストリームになる
 - ASP.NET Core 3.0はgRPCに対応
 - 既存のGoogle.Protobufパッケージを使っていそう
 - C# 8.0の非同期ストリームに対応した拡張メソッドも提供
 - gRPCの`stream`に対して`await foreach`を使える

非同期ストリームの例

- gRPCを例にして非同期ストリームを紹介
 - サンプル: [NetCoreGrpc](#)
- 背景
 - gRPCは非同期ストリームを送受信する想定を持っている
 - `stream`キーワードを付けると非同期ストリームになる
 - ASP.NET Core 3.0はgRPCに対応
 - 既存のGoogle.Protobufパッケージを使っていそう
 - C# 8.0の非同期ストリームに対応した拡張メソッドも提供
 - gRPCの`stream`に対して`await foreach`を使える

非同期ストリームの例(gRPC)

- サンプル(proto定義)
 - Microsoft独自なことはしておらず、普通のProtocol Buffers

```
service Sample {  
    rpc GetValues(stream SampleRequest) returns (stream SampleResponse);  
}  
  
message SampleRequest {  
    int32 bits = 1;  
    int32 length = 2;  
}  
  
message SampleResponse {  
    repeated int32 values = 1;  
}
```

`stream` って付けておくと
非同期に複数のデータを送れる

非同期ストリームの例(gRPC)

- サンプル(proto定義)
 - Microsoft独自なことはしておらず、普通のProtocol Buffers

```
service Sample {  
    rpc GetValues(stream SampleRequest) returns (stream SampleResponse);  
}  
  
message SampleRequest {  
    int32 bits = 1;  
    int32 length = 2;  
}  
  
message SampleResponse {  
    repeated int32 values = 1;  
}
```

streamって付けておくと
非同期に複数のデータを送れる

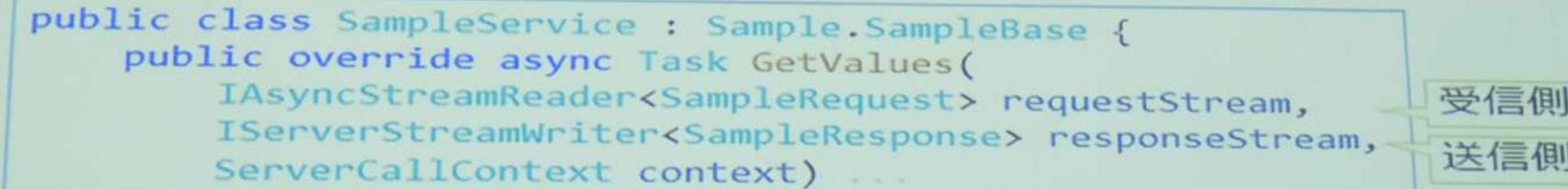
例として

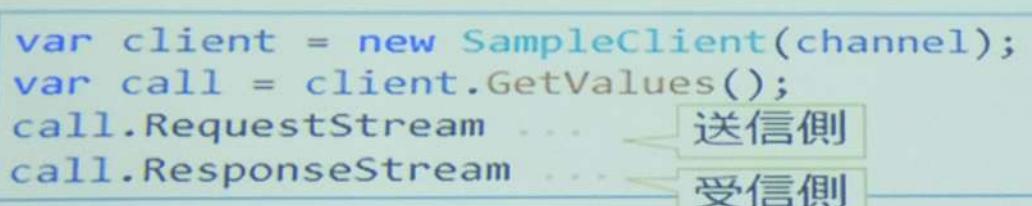
- bits桁の数値をlength個要求して
- リスト(repeated)で返してもらう

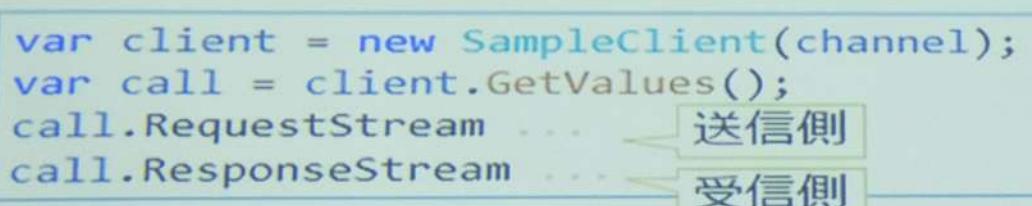
非同期ストリームの例(コード生成)

- protoからC#コードが生成される
 - サーバー側 (生成結果のクラスを派生して使う)

```
public class SampleService : Sample.SampleBase {  
    public override async Task GetValues(  
        IAsyncStreamReader<SampleRequest> requestStream,  
        IServerStreamWriter<SampleResponse> responseStream,  
        ServerCallContext context) ...
```


 - クライアント側 (生成結果のクラスをnewして使う)

```
var client = new SampleClient(channel);  
var call = client.GetValues();  
call.RequestStream ... 
```

```
var client = new SampleClient(channel);  
var call = client.GetValues();  
call.RequestStream ... 
```

非同期ストリームの例(送信側)

- 一定時間ごとにデータを要求

```
async Task sender()
{
    for (int i = 1; i < 32; i++)
    {
        await call.RequestStream.WriteAsync(
            new SampleRequest { Bits = i, Length = i });
        await Task.Delay(500); // とりあえずちょっと遅延を挟んでおく
    }
    await call.RequestStream.CompleteAsync();
}
```

複数のデータを送信
(stream)

とりあえずちょっと遅延を挟んでおく

もうこれ以上送るデータがないことを伝える

非同期ストリームの例(送信側)

- 一定時間ごとにデータを要求

```
async Task sender()
{
    for (int i = 1; i < 32; i++)
    {
        await call.RequestStream.WriteAsync(
            new SampleRequest { Bits = i, Length = i });
        await Task.Delay(500);
    }
    await call.RequestStream.CompleteAsync();
}
```

複数のデータを送信
(stream)

とりあえずちょっと遅延を挟んでおく

もうこれ以上送るデータがないことを伝える

非同期ストリームの例(サーバー)

- 要求が来るたびにリストを作って返す

```
var rand = new Random();
await foreach (var req in requestStream.ReadAllAsync())
{
    var mask = (1 << req.Bits) - 1;
    var len = req.Length;

    var res = new SampleResponse();
    for (int i = 0; i < len; i++)
        res.Values.Add(rand.Next() & mask); len要素のリストを作つて

    await responseStream.WriteAsync(res); そのリストを送る
}
```

1要求受け取るたびに

len要素のリストを作つて

そのリストを送る

非同期ストリームの例(受信側)

- 複数回非同期でリストが届いて、リストの1要素ずつ返す

```
async IAsyncEnumerable<int> receiver()
{
    await foreach (var res in call.ResponseStream.ReadAllAsync())
    {
        foreach (var value in res.Values)
        {
            yield return value;
        }
    }
}
```

複数のデータを受信
(stream)

受信のたびに複数の値
(repeated)

リストの1要素ずつ返す

非同期ストリームの例(受信側)

- 複数回非同期でリストが届いて、リストの1要素ずつ返す

```
async IAsyncEnumerable<int> receiver()
{
    await foreach (var res in call.ResponseStream.ReadAllAsync())
    {
        foreach (var value in res.Values)
        {
            yield return value;
        }
    }
}
```

複数のデータを受信
(stream)

受信のたびに複数の値
(repeated)

リストの1要素ずつ返す

非同期ストリームの例(受信側)

- 複数回非同期でリストが届いて、リストの1要素ずつ返す

```
async IAsyncEnumerable<int> receiver()
{
    昔だったら

- void receiver(IObservable<int> results)
  - foreachで使いにくい
- Task<IEnumerable<int>> receiver()
  - 1要素ずつ返せない(List<T>とかのバッファー必須)


}
```

C# 8.0の言語構文・ライブラリ的な説明

- 非同期メソッドの拡張
 - awaitとyieldの混在
 - async修飾子を付ける
 - 戻り値は`IAsyncEnumerable<T>`型

C# 8.0の言語構文・ライブラリ的な説明

- 非同期メソッドの拡張
 - `await`と`yield`の混在
 - `async`修飾子を付ける
 - 戻り値は`IAsyncEnumerable<T>`型
- 非同期`foreach`
 - `await foreach (var x in asyncStream)`
 - `IAsyncEnumerable<T>` (と同じ名前のメソッドを持つ型)を受け付ける

C# 8.0の言語構文・ライブラリ的な説明

- 非同期メソッドの拡張
 - awaitとyieldの混在
 - async修飾子を付ける
 - 戻り値は`IAsyncEnumerable<T>`型
- 非同期foreach
 - `await foreach (var x in asyncStream)`
 - `IAsyncEnumerable<T>` (と同じ名前のメソッドを持つ型)を受け付ける
- (おまけで)非同期using
 - `await using (var x = asyncResource)`
 - `IAsyncDisposable` (同じ名前のメソッドを持つ型)を受け付ける

要求されるフレームワーク

- 非同期ストリームには`IAsyncEnumerable<T>`型などが必須
 - .NET Standard2.1/.NET Core 3.0では標準提供
 - 古いフレームワーク向けのNuGetパッケージ提供あり
 - [Microsoft.Bcl.AsyncInterfaces](#)
 - netstandard2.0/net461でも使える(Unityでも使える)

要求されるフレームワーク

- 非同期ストリームには`IAsyncEnumerable<T>`型などが必須
 - .NET Standard2.1/.NET Core 3.0では標準提供
 - 古いフレームワーク向けのNuGetパッケージ提供あり
 - [Microsoft.Bcl.AsyncInterfaces](#)
 - netstandard2.0/net461でも使える(Unityでも使える)
- ※非同期ストリームがらみだけ異例
 - 最近の方針としてはC#と.NETの世代をそろえて使ってほしそう
 - 古いフレームワークで最新のC#を使うのは「わかってる人が勝手にやって」
 - `Range/Index`などは公式ライブラリ提供なし
 - 非同期ストリームがらみはそれだけ需要が高い

要求されるフレームワーク

- 非同期ストリームには`IAsyncEnumerable<T>`型などが必須
 - .NET Standard2.1/.NET Core 3.0では標準提供
 - 古いフレームワーク向けのNuGetパッケージ提供あり
 - [Microsoft.Bcl.AsyncInterfaces](#)
 - netstandard2.0/net461でも使える(Unityでも使える)
- ※非同期ストリームがらみだけ異例
 - 最近の方針としてはC#と.NETの世代をそろえて使ってほしそう
 - 古いフレームワークで最新のC#を使うのは「わかってる人が勝手にやって」
 - `Range/Index`などは公式ライブラリ提供なし
 - 非同期ストリームがらみはそれだけ需要が高い

IAsyncEnumerable<T>インターフェイス

- `IEnumerable<T>`の非同期版

```
public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator(
        CancellationToken cancellationToken = default);
}

public interface IAsyncEnumerator<out T> : IAsyncDisposable
{
    T Current { get; }
    ValueTask<bool> MoveNextAsync();
}
```

IAsyncEnumerable<T>インターフェイス

- **IEnumerable<T>**の非同期版

```
public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator(
        CancellationToken cancellationToken = default);
}

public interface IAsyncEnumerator<out T> : IAsyncDisposable
{
    T Current { get; }
    ValueTask<bool> MoveNextAsync();
}
```

同期版との差

- 名前に Asyncが入ってる
- ジェネリック版のみ

• `CancellationToken`をオプションで受け付ける

• 戻り値が `ValueTask`
• `Reset`メソッドはない

非同期foreach

- `IAsyncEnumerable<T>`に対するforeach
 - 仕組みは同期版とほぼ同じ

```
await foreach (var x in s)
{
    ...
}
```

- パターンベース
 - (同名のメソッドを持っていればインターフェイス実装は不要)



```
var e = items.GetAsyncEnumerator();
try
{
    while (await e.MoveNextAsync())
    {
        int item = e.Current;
        ...
    }
}
finally
{
    if (e != null) { await e.DisposeAsync(); }
}
```

- 呼ぶメソッドがAsyncで
- awaitが付く

IAsyncDisposableインターフェイス

- **IDisposable**の非同期版

```
public interface IAsyncDisposable
{
    ValueTask DisposeAsync();
}
```

• 戻り値が**ValueTask**

同期版との差

- 名前に**Async**が入ってる

IAsyncDisposableインターフェイス

- **IDisposable**の非同期版

```
public interface IAsyncDisposable
{
    ValueTask DisposeAsync();
}
```

- 戻り値が**ValueTask**

同期版との差

- 名前に**Async**が入ってる

IAsyncDisposableインターフェイス

- **IDisposable**の非同期版

```
public interface IAsyncDisposable
{
    ValueTask DisposeAsync();
}
```

- 戻り値が**ValueTask**

同期版との差

- 名前にAsyncが入ってる

非同期using

- `IAsyncDisposable`に対する`using`

```
await using (d)
{
    ...
}
```

非同期using

- `IAsyncDisposable`に対する`using`
 - これも仕組みは同期版とほぼ同じ

```
await using (d)
{
    ...
}
```



- (同期版と違つて)
パターンベース
 - (同名のメソッドを持って
いればインターフェイス
実装は不要)

```
try
{
    ...
}
finally
{
    await d.DisposeAsync();
}
```

- 呼ぶメソッドが`Async`で
- `await`が付く

非同期using

- `IAsyncDisposable`に対する`using`
 - これも仕組みは同期版とほぼ同じ

```
await using (d)  
{  
    ...  
}
```



```
try  
{  
    ...  
}  
finally  
{  
    await d.DisposeAsync();  
}
```

- 呼ぶメソッドが`Async`で
- `await`が付く

- (同期版と違つて)
パターンベース
 - (同名のメソッドを持って
いればインターフェイス
実装は不要)

非同期メソッドの拡張

- 非同期メソッド中にyieldを書けるように
 - これまでの非同期メソッド同様、`async`修飾子を付ける
 - これまでのイテレーター同様、メソッド中にyieldを書く
 - 戻り値は`IAsyncEnumerable<T>`である必要あり

```
async IAsyncEnumerable<int> AsyncIterator()
{
    await Task.Delay(1);
    yield return 1;
    yield break;
}
```

非同期メソッドの拡張

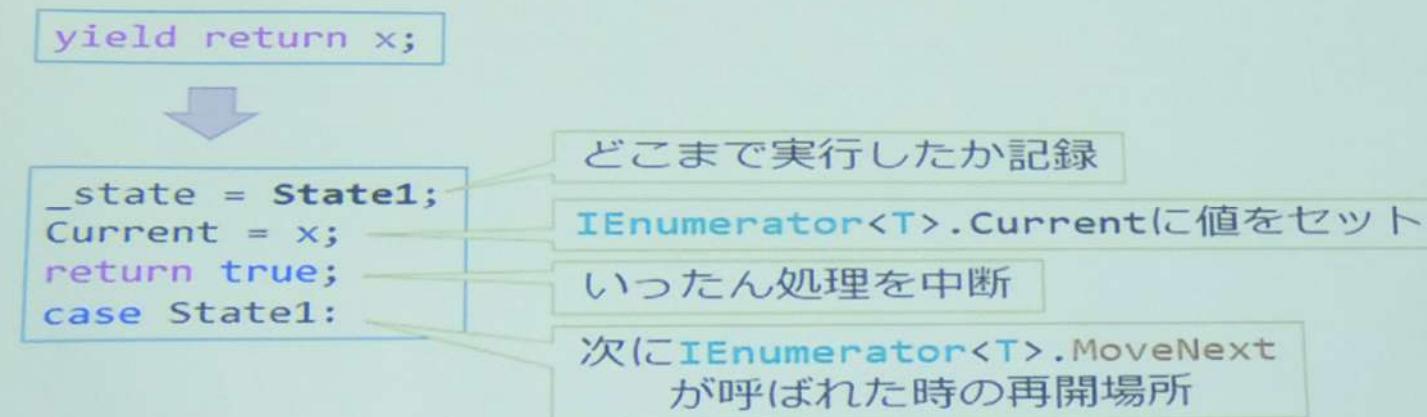
- 非同期メソッド中に `yield` を書けるように
 - これまでの非同期メソッド同様、 `async`修飾子を付ける
 - これまでのイテレーター同様、 メソッド中に `yield` を書く
 - 戻り値は `IAsyncEnumerable<T>` である必要あり

```
async IAsyncEnumerable<int> AsyncIterator()
{
    await Task.Delay(1);
    yield return 1;
    yield break;
}
```

要するに、 非同期メソッド
とイテレーターの混在

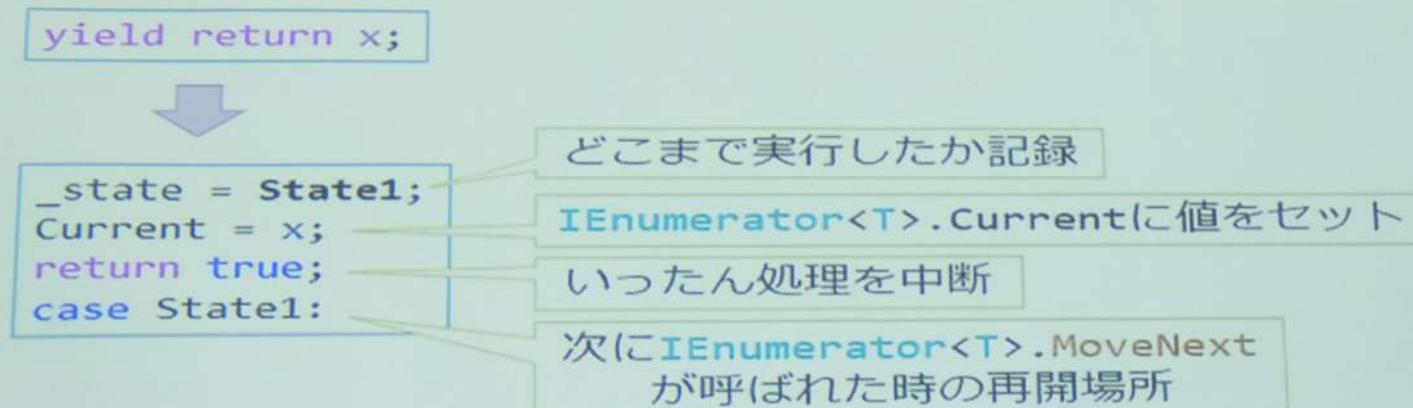
補足: (同期)イテレーター

- `yield return` の展開結果(疑似コード)
 - 状態記録 → 中断 → 再開



補足: (同期)イテレーター

- `yield return` の展開結果(疑似コード)
 - 状態記録 → 中断 → 再開



補足: await演算子

- awaitの展開結果(疑似コード)
 - 状態記録 → 中断 → 再開

```
var result = await task;
```



```
_awaiter = task.GetAwaiter();
if (!_awaiter.IsCompleted)
{
    _state = State1;
    _awaiter.OnComplete(自分自身);
    return;
}
case State1:
var result = _awaiter.GetResult();
```

どこまで実行したか記録

task完了時に呼びなおしてもらう

いったん処理を中断

OnCompleteが呼ばれた時の再開場所

結果の受け取り

補足: await演算子

- awaitの展開結果(疑似コード)
 - 状態記録 → 中断 → 再開

```
var result = await task;
```



```
_awaiter = task.GetAwaiter();
if (!_awaiter.IsCompleted)
{
    _state = State1;
    _awaiter.OnComplete(自分自身);
    return;
}
case State1:
var result = _awaiter.GetResult();
```

どこまで実行したか記録

task完了時に呼びなおしてもらう

いったん処理を中断

OnCompleteが呼ばれた時の再開場所

結果の受け取り

補足: await演算子

- awaitの展開結果(疑似コード)

- 状態記録 → 中断 → 再開

```
var result = await task;
```



```
_awaiter = task.GetAwaiter();
if (!_awaiter.IsCompleted)
{
    _state = State1;
    _awaiter.OnComplete(自分自身);
    return;
}
case State1:
var result = _awaiter.GetResult();
```

この辺りの構造がyield return
とまったく同じ

どこまで実行したか記録

task完了時に呼びなおしてもらう

いったん処理を中断

OnCompleteが呼ばれた時の再開場所

結果の受け取り

補足: await演算子

- **await**の展開結果(疑似コード)

- 状態記録 → 中断 → 再開

```
var result = await task;
```



```
_awaiter = task.GetAwaiter();
if (!awaiter.IsCompleted)
{
    state = State1;
    awaiter.OnComplete(自分自身);
    return;
}
case State1:
var result = awaiter.GetResult();
```

この辺りの構造がyield return
とまったく同じ

どこまで実行したか記録

task完了時に呼びなおしてもらう

いったん処理を中断

OnCompleteが呼ばれた時の再開場所

結果の受け取り

非同期イテレーター

- 非同期メソッド中の `yield return` の展開結果(疑似コード)
 - 状態記録 → 中断 → 再開(根底にある仕組みが同じだから混ぜれる)

```
var result = await task;  
yield return result;
```



```
var result = _awaiter.GetResult();  
_state = State1;  
Current = result;  
_promise.TrySetResult(true);  
return;  
case State1:
```

`await`の結果を受け取り

どこまで実行したか記録

`Current`に値をセット

1つ前の `MoveNextAsync` を完了させる

いったん処理を中断

次に `MoveNextAsync` が呼ばれた時の再開場所

非同期イテレーター

- ・非同期メソッド中の **yield return** の展開結果(疑似コード)
 - ・状態記録 → 中断 → 再開(根底にある仕組みが同じだから混ぜれる)

```
var result = await task;  
yield return result;
```



```
var result = _awaiter.GetResult();  
_state = State1;  
Current = result;  
_promise.TrySetResult(true);  
return;  
case State1:
```

awaitの結果を受け取り

どこまで実行したか記録

Currentに値をセット

1つ前のMoveNextAsyncを完了させる

いったん処理を中断

次にMoveNextAsyncが呼ばれた時の再開場所

パフォーマンス

- Q. なぜ非同期にしたいか?
 - A. パフォーマンスを改善したい
- Taskの生成などのオーバーヘッドがネックになってはいけない

パフォーマンス

- Q. なぜ非同期にしたいか?
 - A. パフォーマンスを改善したい
- 
- Taskの生成などのオーバーヘッドがネックになってはいけない
 - もしC# 5.0世代の技術でやろうとすると...

```
interface IAsyncEnumerator<out T>
{
    T Current { get; }
    Task<bool> MoveNextAsync();
}
```

MoveNextAsyncのたびに
Taskクラスのインスタンスができる

```
_promise = new TaskCompletionSource<T>();  
...  
_promise.TrySetResult(true);
```

TaskCompletionSourceクラスは
再利用ができないので毎回new

パフォーマンス

- Q. なぜ非同期にしたいか?
 - A. パフォーマンスを改善したい
- **Task**の生成などのオーバーヘッドがネックになってはいけない
 - もしC# 5.0世代の技術でやろうとすると...

```
interface IAsyncEnumerator<out T>
{
    T Current { get; }
    Task<bool> MoveNextAsync();
}
```

MoveNextAsyncのたびに
Taskクラスのインスタンスができる

```
_promise = new TaskCompletionSource<T>();  
...  
_promise.TrySetResult(true);
```

TaskCompletionSourceクラスは
再利用ができないので毎回new

パフォーマンスに関する前提

- 受信側サンプルより再掲:

```
await foreach (var res in call.ResponseStream.ReadAllAsync())
    foreach (var value in res.Values)
        yield return value;
```

- 本当に非同期処理が必要なことは少ない
 - ReadAllAsyncの方だけ非同期
 - Values側は同期
 - ReadAllAsyncもバッファリングが効けば同期になることがある

パフォーマンスに関する前提

- 受信側サンプルより再掲:

```
await foreach (var res in call.ResponseStream.ReadAllAsync())
    foreach (var value in res.Values)
        yield return value;
```

- 本当に非同期処理が必要なことは少ない
 - ReadAllAsyncの方だけ非同期
 - Values側は同期
 - ReadAllAsyncもバッファリングが効けば同期になることがある
- IAsyncEnumerator<T>.MoveNextAsyncの呼ばれ方
 - 1度に1人だけがawaitする
 - 戻り値のValueTaskは1回限りGetResultが呼ばれる

パフォーマンス改善のために

- `ValueTask<T>`構造体
 - C# 7.0/.NET Core 2.0世代
- `IValueTaskSource<T>`インターフェイス
 - C# 7.2/.NET Core 2.1世代
- `ManualResetValueTaskSourceCore<T>`構造体
 - C# 8.0/.NET Core 3.0世代

パフォーマンス改善のために

- `ValueTask<T>`構造体
 - C# 7.0/.NET Core 2.0世代
- `IValueTaskSource<T>`インターフェイス
 - C# 7.2/.NET Core 2.1世代
- `ManualResetValueTaskSourceCore<T>`構造体
 - C# 8.0/.NET Core 3.0世代



ちょっと歴史を振り返りつつ説明

Task周りの歴史

～クラス初導入から非同期ストリームに至るまで

時系列



課題: マルチコア化

- マルチコアCPUの普及
 - 高クロック化に限界 (サイズが量子力学的な世界に突入)
 - 単体で速くできないので並列に並べるしかない
- コア間のコミュニケーションコストが課題
 - 「N個あればN倍」とはならない
 - 他のコアの動作を邪魔をしてはいけない
 - 暇になっているコアをなくさないといけない



Work stealing

- 小さいタスクを大量にさばくにはスレッドは重たい
 - 新規作成・スレッド間のコンテキスト切り替えの負担が大きい
- コア数分のスレッドを事前に立てておく
 - 各スレッドにキューを持つ(ローカルキュー)
 - タスクは1度キューに詰める
 - ローカルでやることがなくなったら他のスレッドのキューからタスクを奪って(stealing)実行する
 - Work stealingって呼ぶ
 - 効率のいい実装方法がいろいろ研究されてる
 - (JavaとかGoとかでも同時期に)

Work stealing

- 小さいタスクを大量にさばくにはスレッドは重たい
 - 新規作成・スレッド間のコンテキスト切り替えの負担が大きい
- ↓
- コア数分のスレッドを事前に立てておく
 - 各スレッドにキューを持つ(ローカルキュー)
 - タスクは1度キューに詰める
 - ローカルでやることがなくなったら他のスレッドのキューからタスクを奪って(stealing)実行する
 - Work stealingって呼ぶ
 - 効率のいい実装方法がいろいろ研究されてる
 - (JavaとかGoとかでも同時期に)

当時の.NET周り

- Windows 7 (2009/9リリース)
 - User Mode Scheduler
 - Work stealing的なことをやってる並列処理API
- .NET Framework 4 (2010/4)
 - Taskクラス追加
 - Parallelクラス追加
 - ThreadPoolがWork stealing実装になって効率化
 - 当時の課題は「マルチコアを使い切ること」
 - CPUをフルに使うような計算主体

当時の.NET周り

- Windows 7 (2009/9リリース)
 - User Mode Scheduler
 - Work stealing的なことをやってる並列処理API
- .NET Framework 4 (2010/4)
 - Taskクラス追加
 - Parallelクラス追加
 - ThreadPoolがWork stealing実装になって効率化

- 当時の課題は「マルチコアを使い切ること」
- CPUをフルに使うような計算主体

Taskクラス(継続呼び出し)

- 同期だと `var result = x.M();` と書けるものがあったとして
 - Task以前: Begin/Endのペア

```
x.BeginM(state =>
{
    var result = x.EndM(state);
});
```

匿名関数がなかった頃は
もっと大変だった

- Task以後: ContinueWith(継続呼び出し)

```
x.MAsync().ContinueWith(t =>
{
    var result = t.Result;
});
```

Taskクラス(継続呼び出し)

- 同期だと `var result = x.M();` と書けるものがあったとして
 - Task以前: Begin/Endのペア

```
x.BeginM(state =>
{
    var result = x.EndM(state);
});
```

匿名関数がなかった頃は
もっと大変だった

- Task以後: ContinueWith(継続呼び出し)

```
x.MAsync().ContinueWith(t =>
{
    var result = t.Result;
});
```

C# 5.0世代 (2012年頃)

"await"元年

課題: I/O-bound

- CPUの高速化に対して...
 - 相対的にストレージアクセスが遅くなった
 - CPUより3・4桁遅い
 - ネットワークを使うことが多くなった
 - 秒単位で待たされることもざら
- ただ待ってる時間が増える
- 待ち時間にフリーズすると印象が悪い
 - 待ってる間に他のことをしたい
 - せめて、マウスは動いてほしいし、インジケーターを出したい

I/O-bound
(CPUの外の世界との
入出力がネック)

当時の.NET周り

- Windows 8 (2012/10)
 - WinRT (新API)は非同期前提
 - 同期処理でフリーズされるくらいなら、いっそ同期APIを用意しない
 - I/Oは軒並み非同期APIのみ提供
- .NET Framework 4.5
 - 非同期I/OにTaskクラスを使うように
 - `Stream.ReadAsync`, `HttpClient.GetAsync`, ...
- C# 5.0 (2012/8)
 - 非同期メソッド導入
 - この当時は戻り値にTaskクラスしか使えなかった

非同期メソッド

- 同期と非同期で同じような書き方ができる
 - I/Oを非同期にしたいだけあって、やりたいことは同期と同じ
 - やりたいことが同じなら書き方は同じでありたい

同期

```
void M()
{
    var result = x.N();
    ...
}
```

非同期

```
async Task MAasync()
{
    var result = await x.NAsync();
    ...
}
```

- 非同期処理の前にawaitを付けるだけ
- (習慣上)メソッド名にAsync語尾を付ける

非同期メソッド

- 同期と非同期で同じような書き方ができる
 - I/Oを非同期にしたいだけであって、やりたいことは同期と同じ
 - やりたいことが同じなら書き方は同じでありたい

同期

```
void M()
{
    var result = x.N();
    ...
}
```

非同期

```
async Task MAync()
{
    var result = await x.NAsync();
    ...
}
```



- 非同期処理の前に**await**を付けるだけ
- (習慣上)メソッド名に**Async**語尾を付ける

Taskの用途が変わった

- 先史時代: マルチコアをフルに使いたい
 - Task.WhenAllやParallel.Forが主力
- await後: むしろI/O待ちに使われるようにな
る
 - 1つのプログラムでCPUをフルに使いきることはあまりない
 - 書きたいコード自体は同期の頃と同じ
 - Taskの利用場面の大半がawaitに

余談: C# 5.0→6.0の開発体制 (2012～2015頃)

WindowsからAzureに
オープン化

C# 5.0開発進行上の成約

- WinRTのための非同期メソッド
 - Windows 8に合わせてC# 5.0を出すのが必須だった
 - Windowsがクローズだったので、C# 5.0もクローズなところがあった
 - 非同期メソッドの初期設計もオープンにできないことが多かった

C# 5.0開発進行上の成約

- WinRTのための非同期メソッド
 - Windows 8に合わせてC# 5.0を出すのが必須だった
 - Windowsがクローズだったので、C# 5.0もクローズなところがあった
 - 非同期メソッドの初期設計もオープンにできないことが多かった

締め切り的にもフィードバック的にもなかなか
自由が利かない中での`await`導入



- 汎用性を持たせる余裕なし
 - 戻り値は`Task`, `Task<T>`に限る
 - `await`は`GetAwaiter`メソッドを直呼び

C# 5.0開発進行上の成約

- WinRTのための非同期メソッド
 - Windows 8に合わせてC# 5.0を出すのが必須だった
 - Windowsがクローズだったので、C# 5.0もクローズなところがあった
 - 非同期メソッドの初期設計もオープンにできないことが多かった

締め切り的にもフィードバック的にもなかなか
自由が利かない中での`await`導入



- 汎用性を持たせる余裕なし
 - 戻り値は`Task`, `Task<T>`に限る
 - `await`は`GetAwaiter`メソッドを直呼び

C# 5.0開発進行上の成約

- WinRTのための非同期メソッド
 - Windows 8に合わせてC# 5.0を出すのが必須だった
 - Windowsがクローズだったので、C# 5.0もクローズなところがあった
 - 非同期メソッドの初期設計もオープンにできないことが多かった

締め切り的にもフィードバック的にもなかなか
自由が利かない中での`await`導入



- 汎用性を持たせる余裕なし
 - 戻り値は`Task`, `Task<T>`に限る
 - `await`は`GetAwaiter`メソッドを直呼び

C# 6.0開発体制

- MicrosoftのCEO交代で一気にオープンに
 - C#コンパイラーも2014/4にオープンソース化(当時はcodeplex)
 - 2015/1にGitHub移行
- コンパイラーを1から作り直してたので言語構文的には停滞期
 - .NET Coreも1.0は「まずは最低限動く」が目標
- 非同期メソッド周りもいったん据え置き
 - 計画としては早期からあった非同期ストリームの話も8.0までおあずけ
- ASP.NET方面からのフィードバックが強くなる

C# 7.0世代 (2017年)

Core 2.0

パフォーマンス向上が命題に

C# 7.0世代 (2017年)

.NET Core 2.0

徐々にパフォーマンス向上が命題に

パフォーマンス

- .NET Core 2.Xはパフォーマンス改善の積み重ねだった
 - ガベコレ前提・安全性と生産性優先の言語の中ではだいぶ速くなった
 - パフォーマンス的にはC, C++, Rustの次くらいの位置には来れてる
- ヒープ アロケーションができるかぎり避ける方針に
 - ガベコレは十分速いけど、「ヒープを使わない」方がもっと速い
 - なので、非同期メソッドのTask<T>のアロケーションも問題に

パフォーマンス

- .NET Core 2.Xはパフォーマンス改善の積み重ねだった
 - ガベコレ前提・安全性と生産性優先の言語の中ではだいぶ速くなった
 - パフォーマンス的にはC, C++, Rustの次くらいの位置には来れてる
 - ヒープアロケーションができるかぎり避ける方針に
 - ガベコレは十分速いけど、「ヒープを使わない」方がもっと速い
 - なので、非同期メソッドのTask<T>のアロケーションも問題に
- ↓
- ValueTask<T>構造体の導入
 - 最初はNuGetパッケージで提供
 - .NET Core 2.0から標準提供

ValueTask

- `ValueTask<T>` = `T` or `Task<T>` な共用体

構造

```
struct ValueTask<T>
{
    T _result;
    Task<T> _task;
}
```

用途の例

```
async ValueTask<int> MAsync()
{
    if (9割9分) return 1;           高確率で実際には同期処理 = T を返せばいい
    await Task.Delay(1);            低確率で非同期処理 = Task<T> が必要
    return 0;
}
```

ValueTask

- `ValueTask<T>` = `T` or `Task<T>` な共用体

構造

```
struct ValueTask<T>
{
    T _result;
    Task<T> _task;
}
```

用途の例

```
async ValueTask<int> MAsync()
{
    if (9割9分) return 1;
    await Task.Delay(1);
    return 0;
}
```

通信結果をキャッシュしたりバッファリングしたり、こういう場面は結構な頻度で起こる
(高確率で`Task<T>`のアロケーションを回避)

高確率で実際には同期処理

= `T` を返せばいい

低確率で非同期処理

= `Task<T>` が必要

非同期メソッド戻り値の汎用化

- C# 7.0で非同期メソッドの戻り値を汎用化
 - `Task<T>`, `Task`以外の型を非同期メソッドの戻り値にできるように
 - パターンベース(所定のメソッドさえ持つていれば型は問わない)
- ただ、結構複雑なパターンだし、自前で書くことはほとんどない
 - なので説明割愛
 - 気になる人は「`AsyncMethodBuilder`」で検索
 - (C# 8.0世代ですら)実用上はほぼ`ValueTask`のために入った機能

ValueTaskに掛かるオーバーヘッド

- コスト
 - Task<T>とTを両方持つ上に、追加でちょっとフラグを持つ
 - (Tによれど大体)16バイト以上になる
 - 常に非同期な場合は単なるオーバーヘッド
 - 結局Task<T>クラスが作られた上で大きめの構造体コピーが発生

ValueTaskに掛かるオーバーヘッド

- コスト
 - `Task<T>`と`T`を両方持つ上に、追加でちょっとフラグを持つ
 - (`T`によれど大体)16バイト以上になる
 - 常に非同期な場合は単なるオーバーヘッド
 - 結局`Task<T>`クラスが作られた上で大きめの構造体コピーが発生

C# 7.0世代での問題
後になってくる改善案が
`IValueTaskSource<T>`

C# 7.2世代 (2018年)

.NET Core 2.1

格的にパフォーマンス向上が命題に

非同期戻り値を汎用化するにあたって

- おさらい
 - Task/Task<T>のアロケーションを避けたい
 - await用途の場合いくつか前提を置ける
 - 1度に1人だけがawaitする
 - 1回限りGetResultが呼ばれる
 - AsyncMethodBuilderを作るのはそれなりに大変

非同期戻り値を汎用化するにあたって

- おさらい
 - `Task`/`Task<T>`のアロケーションを避けたい
 - `await`用途の場合いくつか前提を置ける
 - 1度に1人だけが`await`する
 - 1回限り`GetResult`が呼ばれる
 - `AsyncMethodBuilder`を作るのはそれなりに大変
- 結局、`ValueTask`に乗っかりたい

非同期戻り値を汎用化するにあたって

- おさらい
 - Task/Task<T>のアロケーションを避けたい
 - await用途の場合いくつか前提を置ける
 - 1度に1人だけがawaitする
 - 1回限りGetResultが呼ばれる
 - AsyncMethodBuilderを作るのはそれなりに大変
- 結局、ValueTaskに乗っかりたい
 - IValueTaskSourceインターフェイスを提供することに

IValueTaskSource<T>

- 非同期メソッドの戻り値の求められる要件を満たす型

```
public interface IValueTaskSource<out TResult>
{
    ValueTaskSourceStatus GetStatus(short token);
    void OnCompleted(Action<object?> continuation, object? state,
                     short token, ValueTaskSourceOnCompletedFlags flags);
    TResult GetResult(short token);
}
```

- ValueTask<T>の仕組みに乗っかれる
 - ValueTask<T>のコンストラクターに渡せる
 - AsyncMethodBuilderはValueTask<T>のものをそのまま使う

IValueTaskSource<T>

- やりたいこと自体はTask<T>と同じ

```
public interface IValueTaskSource<out TResult>
{
    ValueTaskSourceStatus GetStatus(short token);
    void OnCompleted(Action<object?> continuation, object? state,
                     short token, ValueTaskSourceOnCompletedFlags flags);
    TResult GetResult(short token);
}
```

```
public class Task<TResult>
{
    TaskStatus Status { get; }
    Task ContinueWith(Action<Task> continuation);
    TResult Result { get; }
}
```

IValueTaskSource<T>

- ・パフォーマンスへの配慮あり

```
public interface IValueTaskSource<out TResult>
{
    ValueTaskSourceStatus GetStatus(short token);
    void OnCompleted(Action<object?> continuation, object? state,
                     short token, ValueTaskSourceOnCompletedFlags flags);
    TResult GetResult(short token);
}
```

- 何回目の呼び出しかを弁別するための数値
- 1つのインスタンスを使いまわす前提

ValueTask<T>側の変化

- **ValueTask<T> = T or Task<T> or IValueTaskSource<T>**

```
struct ValueTask<T>
{
    T _value;
    object _task;
    short _token;
}
```

最初から完了しているときはTを直接渡す

is演算子でTask<T> or IValueTaskSource<T>に分岐

何回目の呼び出しかを弁別するための数値

ValueTask/IValueTaskSource (型引数なし)

- ValueTask = Task or IValueTaskSource or 完了済み

```
struct ValueTask
{
    object _task;
    short _token;
}
```

is演算子でTask or IValueTaskSourceに分岐
nullだったら完了済み

何回目の呼び出しかを弁別するための数値

```
public interface IValueTaskSource
{
    ValueTaskSourceStatus GetStatus(short token);
    void OnCompleted(Action<object?> continuation, object? state,
                    short token, ValueTaskSourceOnCompletedFlags flags);
    void GetResult(short token);
}
```

IValueTaskSourceを再利用

- インスタンスの再利用でアロケーションを減らす例

```
ValueTaskSource _source = new ValueTaskSource();

ValueTask M()
{
    _source.Reset();           // どこか別の場所でTrySetResultする
    return new ValueTask(_source, _source.Token);
}
```

- 1つのインスタンス(_sourceに格納)を使いまわす
- 1つ前のM()呼び出しが終わる前に重複でM()を呼ばない
 - ResetのたびにTokenを変える
 - TrySetResult時にTokenが不一致だと例外を出す

TaskよりValueTaskの方が汎用的に

- Task ... 具象クラス。あまり変更の余地なし
- ValueTask ... これ自体は変更の余地なし
 - IValueTaskSourceを渡せる
 - IValueTaskSourceは自由に実装できる
- どちらが推奨か

```
interface X
{
    Task M1();
    ValueTask M2();
}
```

インターフェイスみたいに
実装がどうなるかわからないとき、
TaskにすべきかValueTaskにすべきか

TaskよりValueTaskの方が汎用的に

- Task ... 具象クラス。あまり変更の余地なし
- ValueTask ... これ自体は変更の余地なし
 - IValueTaskSourceを渡せる
 - IValueTaskSourceは自由に実装できる
- どちらが推奨か

```
interface X
{
    Task M1();
    ValueTask M2();
}
```

インターフェイスみたいに
実装がどうなるかわからないとき、
TaskにすべきかValueTaskにすべきか

TaskよりValueTaskの方が汎用的に

- Task ... 具象クラス。あまり変更の余地なし
- ValueTask ... これ自体は変更の余地なし
 - IValueTaskSourceを渡せる
 - IValueTaskSourceは自由に実装できる
- どちらが推奨か
 - 昔: ValueTaskにもコストはあるし、構造体は扱いにくいし
 - Task優位
 - 今: ValueTaskの方が汎用になったし、実装次第では高パフォーマンス
 - ValueTask優位

TaskよりValueTaskの方が汎用的に

- Task ... 具象クラス。あまり変更の余地なし
- ValueTask ... これ自体は変更の余地なし
 - IValueTaskSourceを渡せる
 - IValueTaskSourceは自由に実装できる
- どちらが推奨か
 - 昔: ValueTaskにもコストはあるし、構造体は扱いにくいし
 - Task優位
 - 今: ValueTaskの方が汎用になったし、実装次第では高パフォーマンス
 - ValueTask優位

扱いにくさ

- ・後入りなので...
 - Task戻り値をValueTask戻り値に変えると破壊的変更になる
 - TaskとValueTaskが混在しているときにWhenAnyとかが面倒
- ・構造体なので...
 - box化に気を付けないといけない
 - ValueTask<T>とValueTaskに継承関係がない
 - それぞれに対して同じロジックを多重保守する必要あり
 - 組み合わせ爆発もあり得る

```
void M(ValueTask a, ValueTask b);
void M<T>(ValueTask a, ValueTask<T> b);
void M<T>(ValueTask<T> a, ValueTask b);
void M<T>(ValueTask<T> a, ValueTask<T> b);
```

引数の数nに対して
 2^n 個のオーバーロード

扱いにくさ

- ・後入りなので...
 - Task戻り値をValueTask戻り値に変えると破壊的変更になる
 - TaskとValueTaskが混在しているときにWhenAnyとかが面倒
- ・構造体なので...
 - box化に気を付けないといけない
 - ValueTask<T>とValueTaskに継承関係がない
 - それぞれに対して同じロジックを多重保守する必要あり
 - 組み合わせ爆発もあり得る

```
void M(ValueTask a, ValueTask b);
void M<T>(ValueTask a, ValueTask<T> b);
void M<T>(ValueTask<T> a, ValueTask b);
void M<T>(ValueTask<T> a, ValueTask<T> b);
```

引数の数nに対して
 2^n 個のオーバーロード

IValueTaskSourceってどう実装するの？

- **IValueTaskSource**を使えと言われましても...
 - 汎用にできるといつてもやることはほぼ常に同じ
 - スレッド安全でないとダメ
 - ちゃんと書くのは結構難しい

IValueTaskSourceってどう実装するの？

- IValueTaskSourceを使えと言われましても...
 - 汎用にできるといつてもやることはほぼ常に同じ
 - スレッド安全でないとダメ
 - ちゃんと書くのは結構難しい
- 共通処理を詰め込んだ構造体を提供
 - ManualResetValueTaskSourceCore<T>
 - 自前のクラスにこの構造体のフィールドを持たせて、IValueTaskSourceの実装に使える

ManualResetValueTaskSourceCoreの例

- `IValueTaskSource<T>` の処理をほぼ丸投げ

```
class ValueTaskSource : IValueTaskSource<T>
{
    private ManualResetValueTaskSourceCore<T> _core;

    public int GetResult(short token) => _core.GetResult(token);
    public ValueTaskSourceStatus GetStatus(short token) => _core.GetStatus(token);
    public void OnCompleted(Action<object> continuation, object state, short token, ...)
        => _core.OnCompleted(continuation, state, token, flags);

    public ValueTask<T> ValueTask => new ValueTask<T>(this, _core.Version);
}
```

Reset(再利用)とか TrySetResult(完了)とかの
タイミングだけ独自にコントロールしてやればOK

ManualResetValueTaskSourceCoreの例

- `IValueTaskSource<T>` の処理をほぼ丸投げ

```
class ValueTaskSource : IValueTaskSource<T>
{
    private ManualResetValueTaskSourceCore<T> _core;           フィールドに持つて

    public int GetResult(short token) => _core.GetResult(token);   ほぼ丸投げ
    public ValueTaskSourceStatus GetStatus(short token) => _core.GetStatus(token);
    public void OnCompleted(Action<object> continuation, object state, short token, ...)
        => _core.OnCompleted(continuation, state, token, flags);

    public ValueTask<T> ValueTask => new ValueTask<T>(this, _core.Version);
}
```

Reset(再利用)とか TrySetResult(完了)とかの
タイミングだけ独自にコントロールしてやればOK

非同期ストリームの中身

- ManualResetValueTaskSourceCore<T>を使って実装

再掲・抜粋

```
await foreach (var x in s)
{
    ...
}
```



```
while (await e.MoveNextAsync())
{
    int item = e.Current;
    ...
}
```

戻り値が ValueTask<T>

本当に非同期な時だけアロケーション

```
var result = await task;
yield return result;
```

非同期ストリームの中身

- ManualResetValueTaskSourceCore<T>を使って宝装

再掲・抜粋

```
await foreach (var x in s)
{
    ...
}
```

戻り値が ValueTask<T>

本当に非同期な時だけアロケーション

```
while (await e.MoveNextAsync())
{
    int item = e.Current;
    ...
}
```

```
var result = await task;
yield return result;
```

非同期ストリームの中身

- ManualResetValueTaskSourceCore<T>を使って実装

再掲・抜粋

```
await foreach (var x in s)
{
    ...
}
```

```
var result = await task;
yield return result;
```



```
while (await e.MoveNextAsync())
{
    int item = e.Current;
    ...
}
```



```
var result = _awaiter.GetResult();
_state = State1;
Current = result;
_promise.TrySetResult(true);
return;

```

戻り値が ValueTask<T>
本当に非同期な時だけアロケーション

ManualResetValueTaskSourceCore<T>を利用
何度も MoveNextAsync を呼んでも同じインスタンスを使いまわし

1度に1人だけがawaitする想定

- continuation (OnCompleteで呼ぶ処理)の管理がシンプルに (=高速)
 - 複数いると...
 - List<Action>なフィールドを持って
 - lockを書いてAdd/Remove
- 1度に1人だけだと
 - Actionなフィールドを持って
 - CompareExchangeで入れ替えするだけ

List自体のnewが発生

lockと比べると軽い処理

1度に1人だけがawaitする想定

- continuation (OnCompleteで呼ぶ処理)の管理がシンプルに (=高速)
- 複数いると...
 - List<Action>なフィールドを持って
 - lockを書いてAdd/Remove
- 1度に1人だけだと
 - Actionなフィールドを持って
 - CompareExchangeで入れ替えるだけ

List自体のnewが発生

lockと比べると軽い処理

ManualResetValueTaskSourceCore<T>
はこっちを採用

「1度に1人だけ」想定

- 普通は書かないようなコードを書くと例外が出る

```
async IAsyncEnumerable<int> A()
{
    await Task.Delay(10); yield return 1;
    await Task.Delay(10); yield return 2;
}

async Task B()
{
    var a = A();
    var e = a.GetAsyncEnumerator();
    var t1 = e.MoveNextAsync();
    var t2 = e.MoveNextAsync(); // t1 完了前に次の MoveNextAsync を呼ぶ
    await t1; // ここで例外が起こる
```

パフォーマンスとの引き換えで
意図的にそういう仕様