

C#の新機能勉強会

～ C#7、8の新機能を活用して速く安全なプログラムを書こう ～



2020/02/14
小島 富治雄

前提条件とゴール

- 前提条件

- C# 1～6

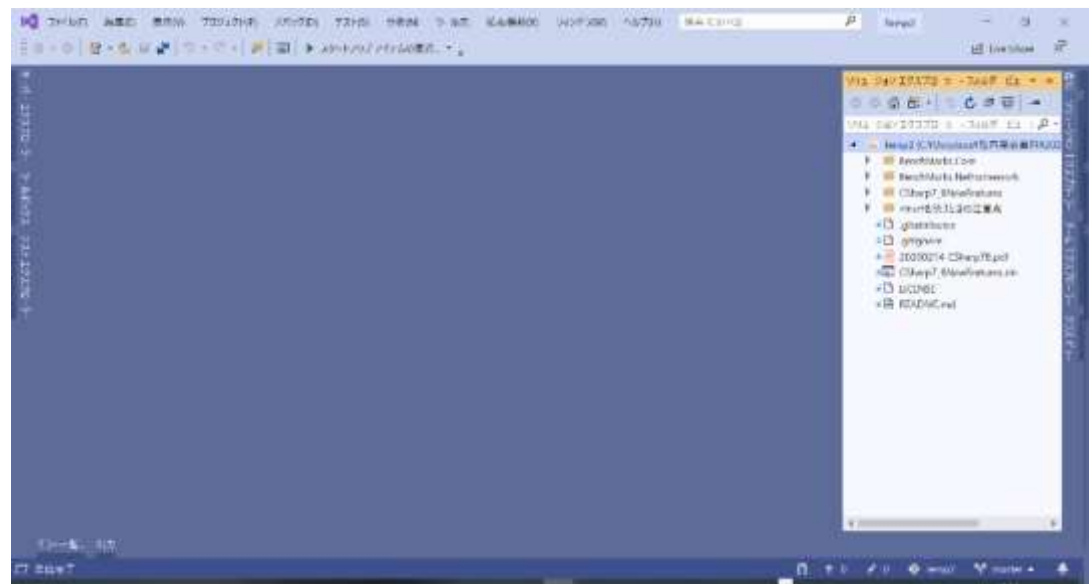
- ゴール

- C# 7～8 の新機能を使った
速く安全で快適なプログラミング



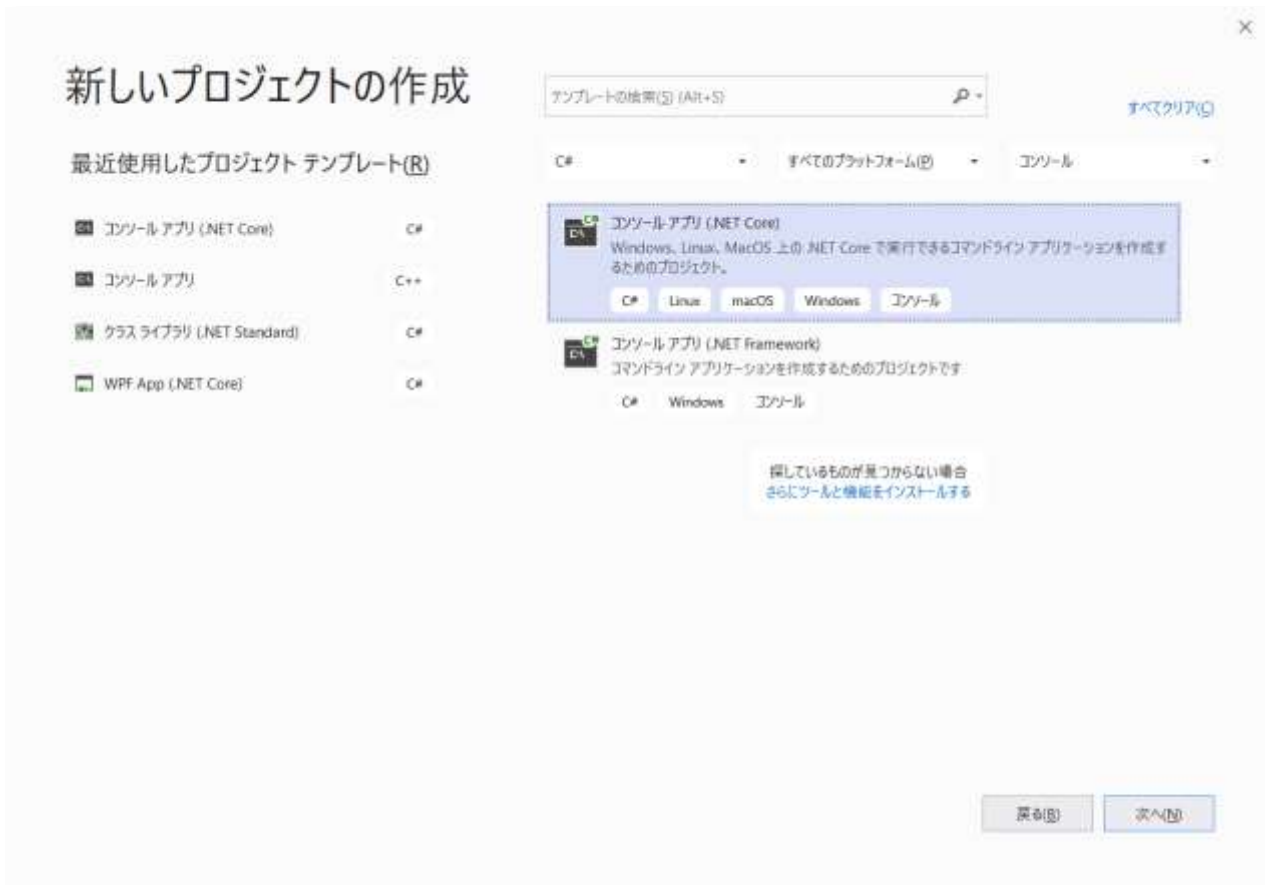
開元環境

- Visual Studio 2019 (Version 16.3以降)
- .NET Core 3.1
- Windows 7 SP1以降、8.1、10 Ver.1703以降



Visual Studio でプロジェクトを新規作成

- コンソール アプリ (.NET Core) C#

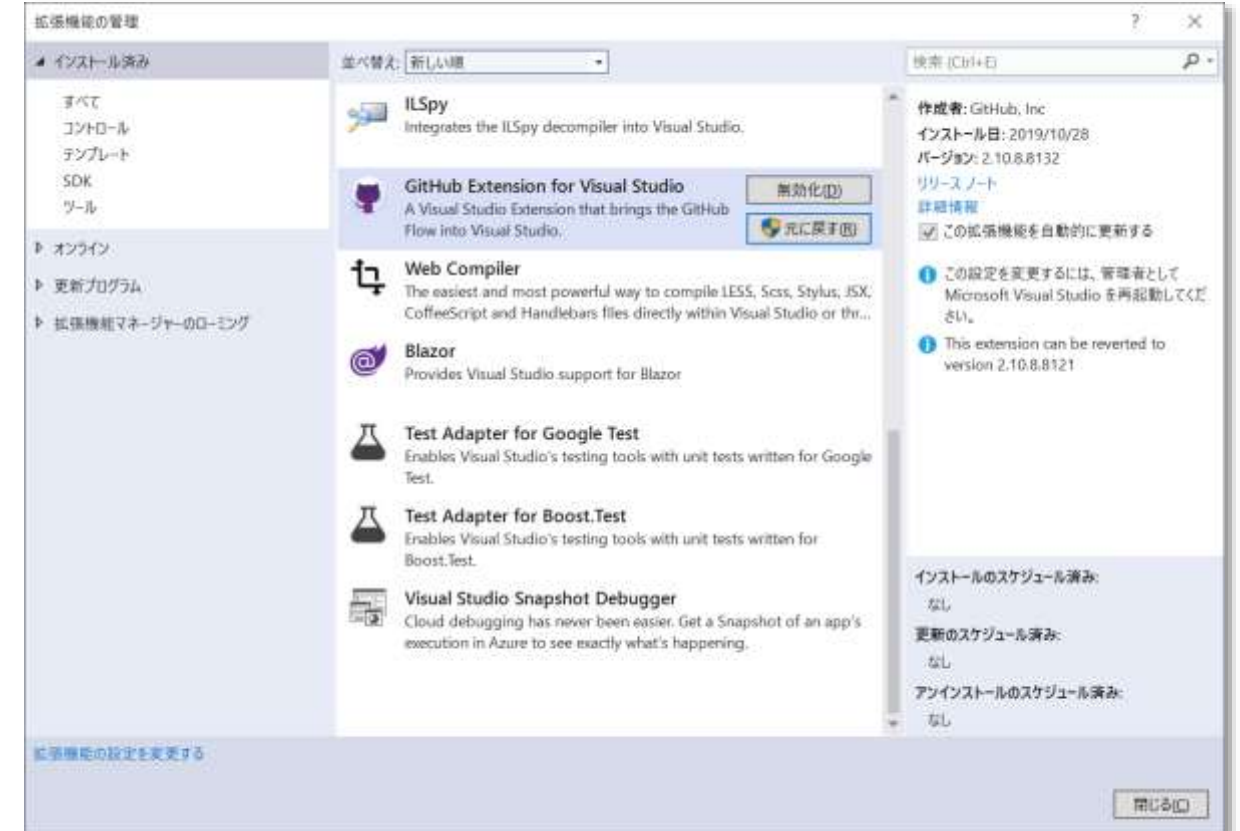
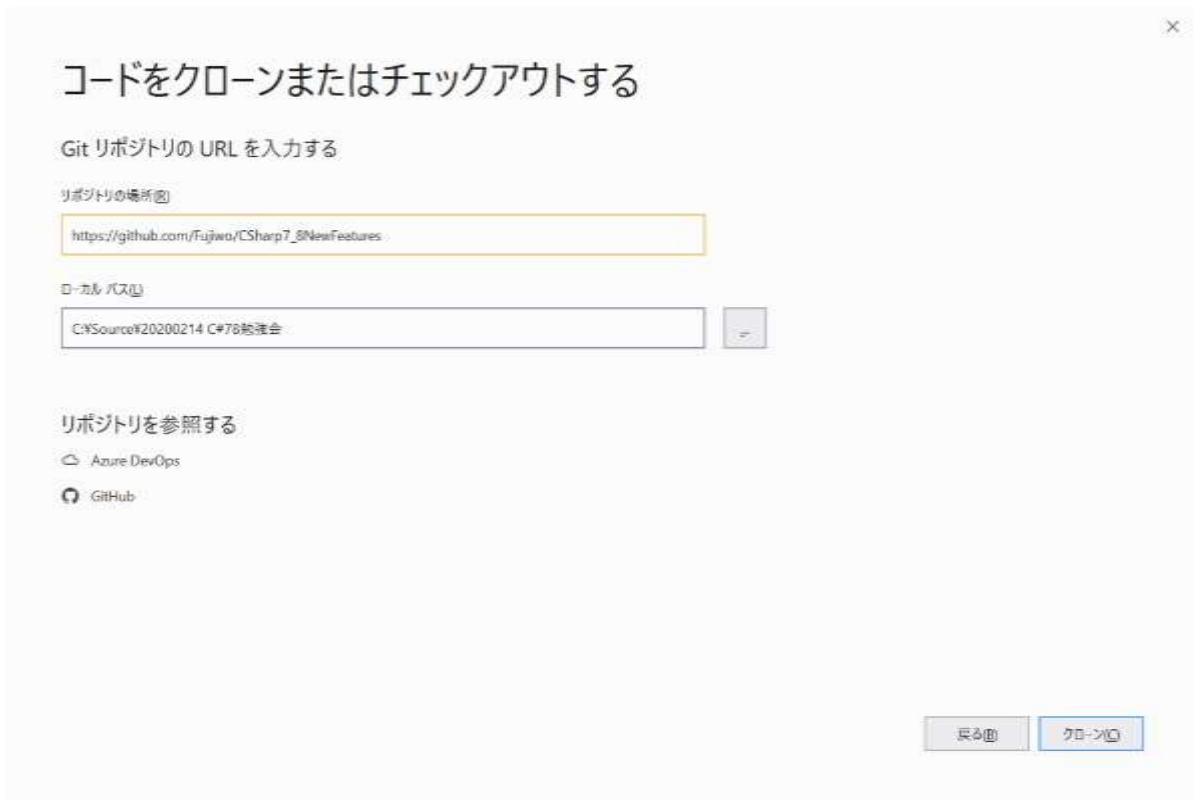


xxx.csproj

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

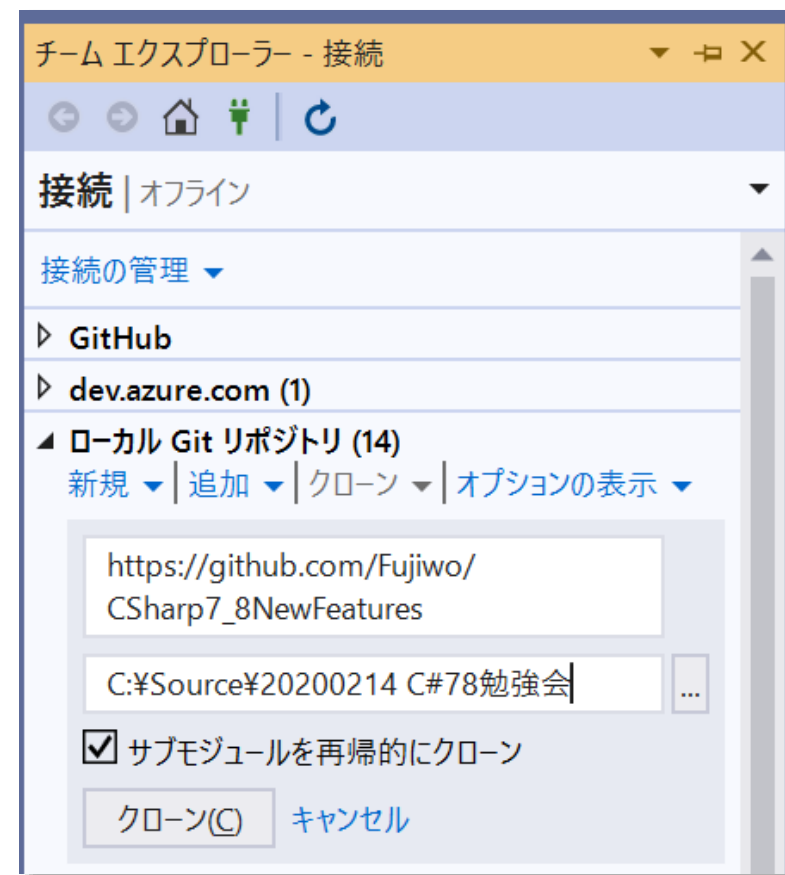
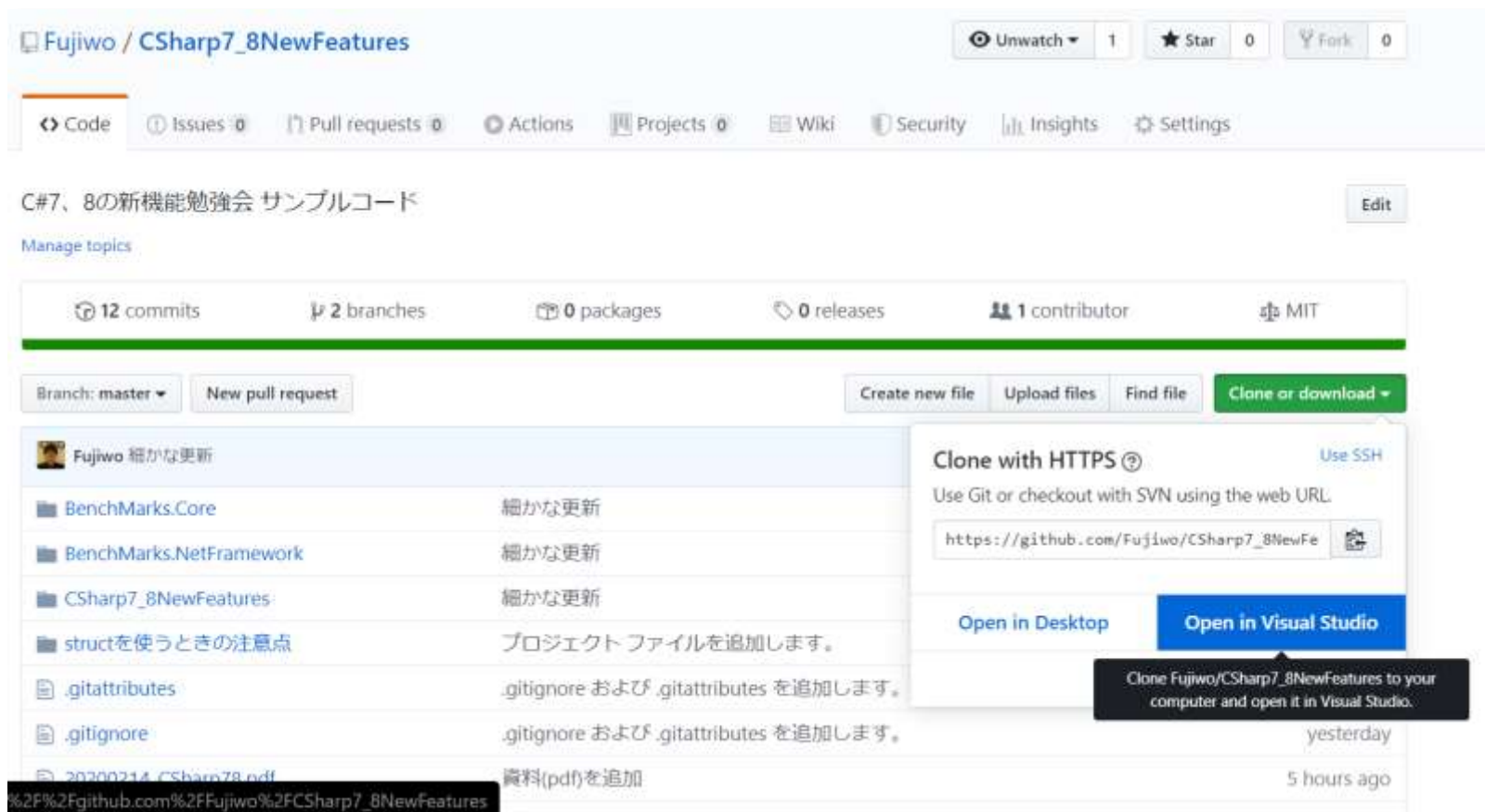
サンプル ソースコードや資料

- https://github.com/Fujiwo/CSharp7_8NewFeatures
- Visual Studio でクローン



サンプル ソースコードや資料

- https://github.com/Fujiwo/CSharp7_8NewFeatures
- GitHub からクローンする場合



アジェンダ

1. C#/.NETの今と近未来
2. C# の値型と参照型
3. C# プログラムの高速化
4. C# 7、8の新機能

AGENDA



1. C#/.NETの今と近未来



C# の歴史



C# Ver.	主な新機能	登場時期	.NET	Visual Studio
1.0, 1.1, 1.2	オブジェクト指向	2002年	.NET Framework 1.0,1.1	.NET, .NET 2003
2.0	ジェネリック	2005年	.NET Framework 2.0	2005
3.0	関数型	2007年	.NET Framework 2.0, 3.0, 3.5	2008, 2010
4.0	動的	2010年	.NET Framework 4	2010
5.0	非同期	2012年	.NET Framework 4.5	2012, 2013
6.0	Roslyn (コンパイラーをC#で実装しオープンソース化)	2015年	.NET Framework 4.6 .NET Core 1.0	2015
7.0, 7.1, 7.2, 7.3	パターン マッチング、値型に関する改良	2017年	.NET Framework 4.6.2, 4.7, 4.7.1, 4.7.2 .NET Core 2.0, 2.1, 2.2	2017
8.0	値型、参照型に関する改良	2019年	.NET Core 3.0	2019 Ver.16.3

C# 7~8



C# Ver.	Visual Studio
7.0	Visual Studio 2017
7.1	Visual Studio 2017 バージョン 15.3
7.2	Visual Studio 2017 バージョン 15.5
7.3	Visual Studio 2017 バージョン 15.7
8.0	Visual Studio 2019 16.3

C# 7~8



ターゲット フレーム	バージョン	C# 言語の既定のバージョン
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	all	C# 7.3

プロジェクト ファイルでの C#のバージョン指定

```
<Project .....>
  <PropertyGroup>
    <OutputType>.....</OutputType>
    <TargetFramework>.....</TargetFramework>
    <!--.....中略.....-->

    <LangVersion>8.0</LangVersion>

    <Nullable>enable</Nullable>
  </PropertyGroup>
  <!--.....中略.....-->
</Project>
```

LangVersion	説明
preview	最新プレビュー バージョン
latest	最新リリース バージョン (マイナー バージョンを含む)
latestMajor	最新リリースの メジャー バージョン
8.0	C# 8.0

C# 8.0

- C#の最新を全部使えるのは、.NET Core 3 以降
と .NET Standard 2.1 以降
- (.NET Framework では一部使用不可)



参考: .NET Framework と .NET Core

- .NET Framework と .NET Core

どちらを使えばよい？

.NET Framework



.NET Core



参考: 今の .NET (2019年9月以降)



.NET Framework 4.8.X

- WPF
- Windows Forms
- ASP.NET

.NET Core 3.X

- WPF (Windows)
- Windows Forms (Windows)
- ASP.NET

Xamarin

- iOS
- Android
- Windows
- MacOS

.NET Standard Library

参考: 近未来の .NET (2020年11月予定)



.NET Framework 4.8.X

- WPF
- Windows Forms
- ASP.NET

保守フェーズに

.NET 5.0

- WPF (Windows)
- Windows Forms (Windows)
- UWP (Windows)
- ASP.NET

Xamarin

- iOS
- Android
- Windows
- MacOS

.NET Standard Library

C# 7~8の新機能の例

- 分解と Deconstruct **タプル**
- タプル (ValueTuple)
- ValueTask
- 参照戻り値
- in 引数
- readonly struct
- ref readonly
- Span **値型 (struct)**

- null 許容参照型 **参照型 (class)**
- 型 switch
- switch 式 **パターン マッチング**
- インターフェイスのデフォルト実装
- ローカル関数 **その他**
- 非同期ストリーム
/非同期イテレーター
/非同期foreach

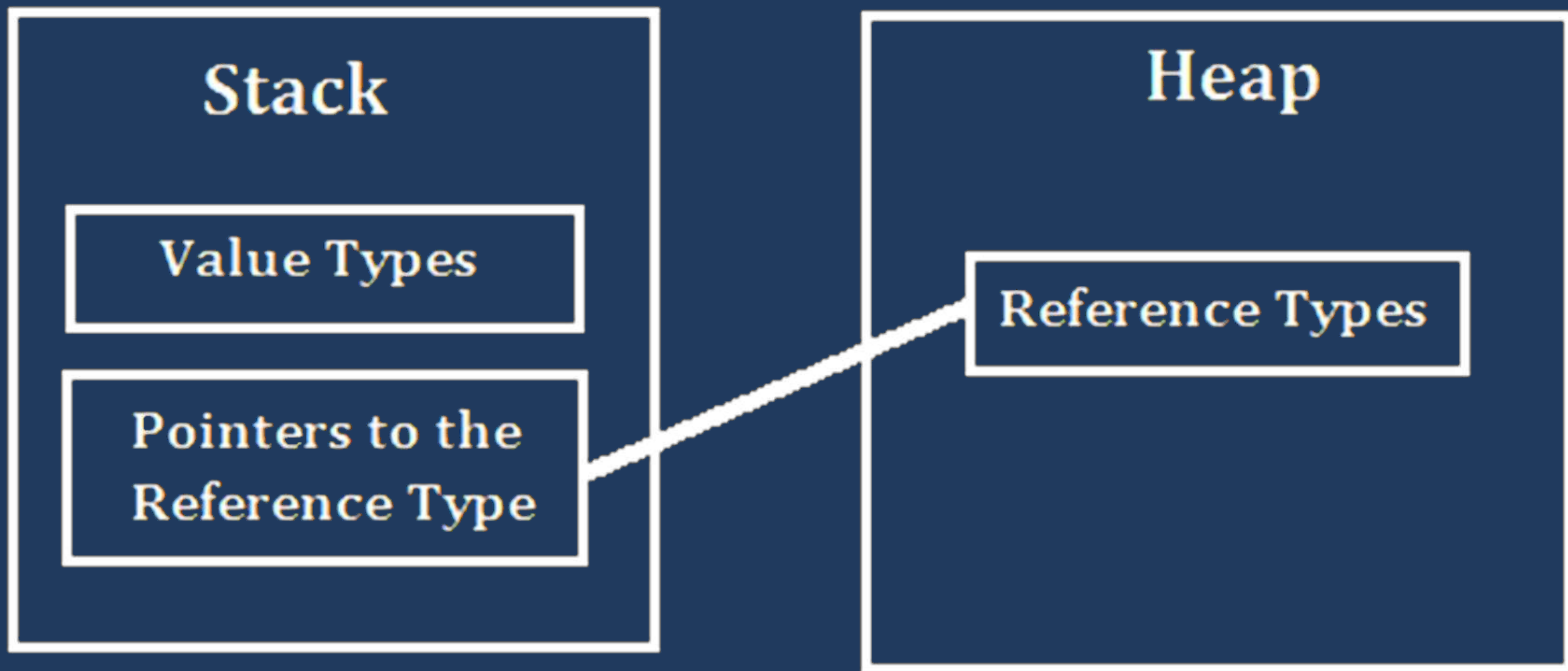
C# 7～8の新機能

- 値型 (struct) や参照型 (class) に関する
改良がたくさん

- 速く
- 安全に
- 快適に



2. C# の値型と参照型



C#に潜むstructの罠？

- [C#に潜むstructの罠 – KAYAC engineers' blog](#)
- 「お急ぎの方のために結論を申しあげますと、
structを使うなとなります。」



お急ぎの方のために
結論を申しあげますと、
そんなわけありません。

なんで C# に値型が
あるかが分かっていない。



遥佐保 @hr_sao · 2019年4月9日

なんだこれ😞この人C#のこと、よく知らないだけでは?罠と思ったのは仕方ない。でも言語には言語の文化があるんやから、そこを尊重した書き方にしないと、これではただのディスり😞気分悪いわ。structのせいにするな



じんぐる @xin9le · 2019年4月9日

途中で読むの止めた。

Unity で全力で性能出すための努力をしないで「struct は罠が多いから使うな」という結論、会社名義の技術ブログの記事としてはちょっと悲しいと思う。

復習: structを使うときの注意点

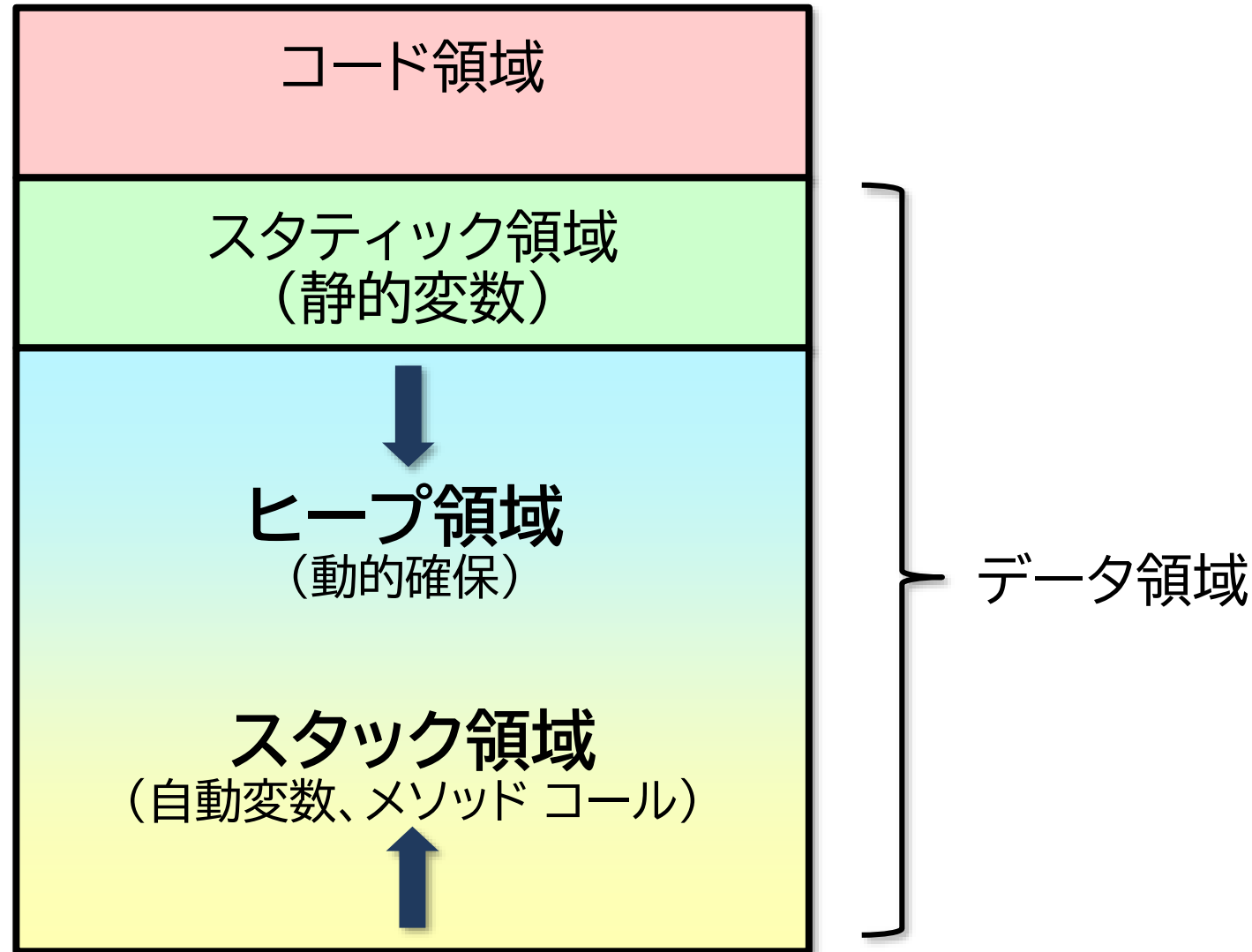
- メソッドやプロパティで内部の状態を変えるときは注意



コードを参考

- https://github.com/Fujiwo/CSharp7_8NewFeatures
 - [サンプルコード](#)

復習: スタック領域とヒープ領域



復習: スタック領域

```
int Add(int x, int y) // C++, x64、最適化なし
```

```
{
```

```
00007FF765AD1000  mov          dword ptr [rsp+10h],edx
```

```
00007FF765AD1004  mov          dword ptr [rsp+8],ecx
```

```
00007FF765AD1008  sub          rsp,18h
```

```
    auto answer = x + y;
```

```
00007FF765AD100C  mov          eax,dword ptr [y]
```

```
00007FF765AD1010  mov          ecx,dword ptr [x]
```

64bits スタック
ポインター レジスター

```
00007FF765AD1014  add          ecx,eax
```

```
00007FF765AD1016  mov          eax,ecx
```

```
00007FF765AD1018  mov          dword ptr [rsp],eax
```

```
    return answer;
```

```
00007FF765AD101B  mov          eax,dword ptr [rsp]
```

```
}
```

```
00007FF765AD101E  add          rsp,18h
```

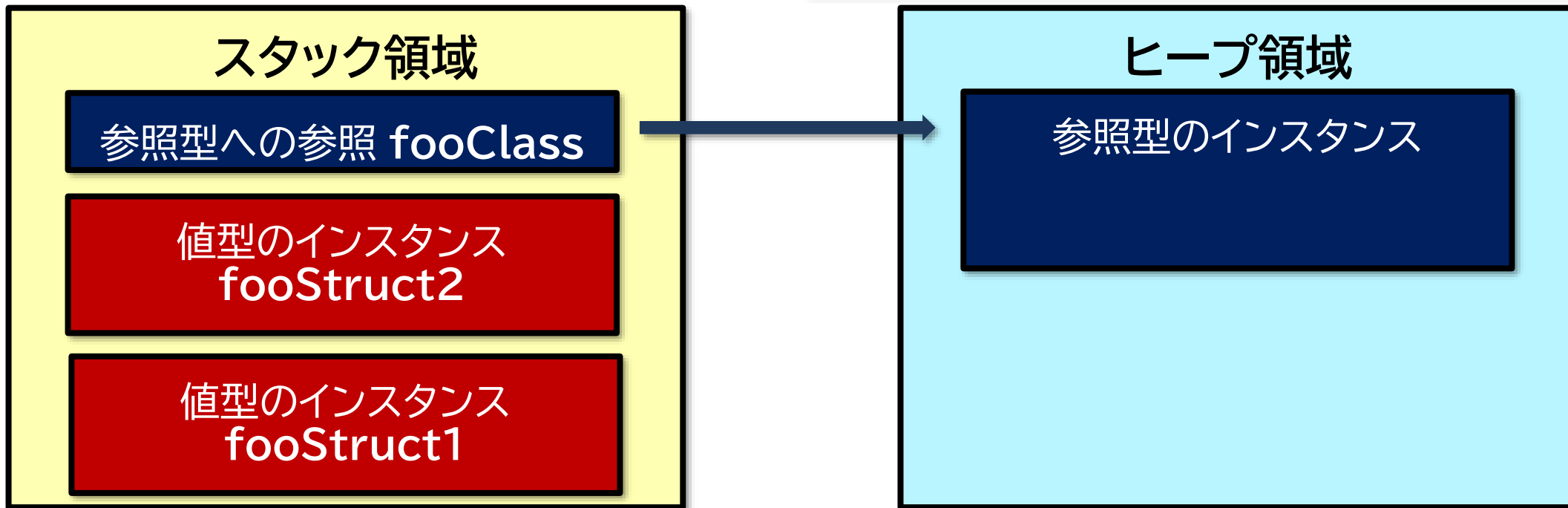
```
00007FF765AD1022  ret
```

- 自動変数やメソッド コールで使われる

復習: スタック領域とヒープ領域

```
struct FooStruct { /* ...省略... */ }  
class FooClass { /* ...省略... */ }
```

```
static void Main() {  
    FooStruct fooStruct1;  
    var fooStruct2 = new FooStruct();  
    var fooClass = new FooClass ();  
}
```



復習: スタック領域とヒープ領域

- スタック領域かヒープ領域か（それ以外か）を意識する
- やたらとヒープ領域を使わない

```
        'replace_interests' => false,  
        'send_welcome'      => false,  
    });  
  
    res.send('error', $result)) {  
        $result = array ('response'=>'error', 'message'  
        );  
        $result = array ('response'=>'success');  
    }  
    res.send($result);
```

C#の参照型 (class)

- (必ず) ヒープ領域
- new および 暗黙の new がハイコスト
- コピーはローコスト (Shallow Copy の場合)
- 要 null チェック

```
class FooClass
{
    public int Id { get; set; } = 0;
    public int Value { get; set; } = 1;
}
```


C#の値型 (struct)

- スタック領域にもおける（参照型のメンバーのときはヒープ領域）
 - （それ自体を）new するかどうかによらない
- いちいち動的確保しないのでローコスト
- コピーがハイコスト（16バイトくらいから）
 - 参照渡し/参照返しすればローコスト
- null チェック不要

```
struct FooStruct
{
    public int Id { get; set; }
    public int Value { get; set; }
}
```

値型 (struct) と参照型 (class) の例

- タプル

- Tuple は参照型
- ValueTuple は値型

- タスク

- Task は参照型
- ValueTask は値型

- 列挙型

- Enum は参照型
- enum は値型

- Nullable

- int? は参照型
- int は値型

これからは原則値型
(struct) の方を使おう!

自作の型も同様。
「struct 使いな」はあり得ない。



値型 (struct) を使うコツ

- 値型を使いスタック領域を活用
- 値型を使ってもヒープ領域が使われちゃう場合(後述)に注意
- コピーをさける (16バイトを目途に)
 - 参照渡し・参照渡し
- なるべく immutable に使う

コピーせず参照で高速にアクセスしつつも、
元の値を変更される危険を回避

```
readonly struct Point {  
    public readonly double X;  
    public readonly double Y;  
    public Point(double x, double y)  
        => (X, Y) = (x, y);  
}
```

いつのまにかヒープ領域が使われてしまう例



- 暗黙的に参照型が new されてヒープ領域が使われてしまう例
- ボックス化
- ラムダ式とキャプチャー
- yield return

```
Enumerable.Range(minimum, maximum)  
    .ForEach(value => sum += value);
```

暗黙の new

- object や interface への代入・初期化
 - 値型の引数を interface の仮引数で受けるだけでも…
- メソッドを引数で渡す
 - delegate が new される
- ラムダ式・ローカル メソッドなどでのメモリ空間のキャプチャー
 - => にカーソルを合わせて確認
- yield return
- async

```
Enumerable.Range(minimum, maximum)  
    .ForEach(value => sum += value);
```

📦 lambda expression

キャプチャされた変数: sum

値型 (struct) はなるべく immutable に使う

- readonly ref や readonly struct、
readonly メソッド (後述)

```
readonly struct Point
{
    public readonly double X;
    public readonly double Y;

    public Point(double x, double y) => (X, Y) = (x, y);
    public double AbsoluteValue => Math.Sqrt(X * X + Y * Y);
    public readonly double DotProduct(Point another) => X * another.X + Y * another.Y;
}
```

- 参考: C++ の const T& 渡し/返し、const メンバー関数などと同じ

3. C# プログラムの高速化



高速化

- 80:20の法則
- ボトルネックの解消を繰り返す
 - パフォーマンス プロファイラー
 - 余計な実行コードがないか ILSpy や LINQPad や SharpLab で確認
- 時間の掛かる処理（I/O、ネットワーク、重い計算）は非同期・別スレッドで



ツール: パフォーマンス プロファイラー



SpeedApp

現在のビュー: 呼び出し元/呼び出し先

SpeedApp.PlayerKojima::GetCommand

SpeedApp.dll

呼び出す関数	現在の関数	呼び出される関数
SpeedAppTest.Table::Game 389 (1.65%)	SpeedApp.PlayerKojima::GetCommand 390 (1.65%)	System.Collections.Generic.Dictionary'2+Enumerator[S... 87 (0.37%)
[ウォーク不可] 1 (0.00%)	関数本体 266 (1.13%)	[外部コード] 30 (0.13%)
		System.Collections.Generic.Dictionary'2[System. _Canon...7 (0.03%)

C:\Dropbox\20200201 BuriKaigi\SpeedApp\SpeedApp\PlayerKojima.cs:256

```
// var myOpenCard = playerOpenCard_NakamiwoKakikaetaraKorosu[this],
var myOpenCardCount = myOpenCard.Count;

foreach (var tableCard in tableCard_NakamiwoKakikaetaraKorosu) {
    for (var index = 0; index < myOpenCardCount; index++) {
        var cpuCard = myOpenCard[index];
        var difference = tableCard.Value.CardNumber - cpuCard.CardNumber;
        if (difference == 1 || difference == -1 || difference == 12 || difference == -12) {
            cpuPutCardCommand.PutCard = cpuCard;
            cpuPutCardCommand.PutPosition = tableCard.Key;
        }
    }
}

#if ATTACK_CPU_AND_MEMORY
    CpuMemoryAttacker.Start();
#endif // ATTACK_CPU_AND_MEMORY
return cpuPutCardCommand;
```

52 % 問題は見つかりませんでした 行: 255 文字: 26 SPC CRLF

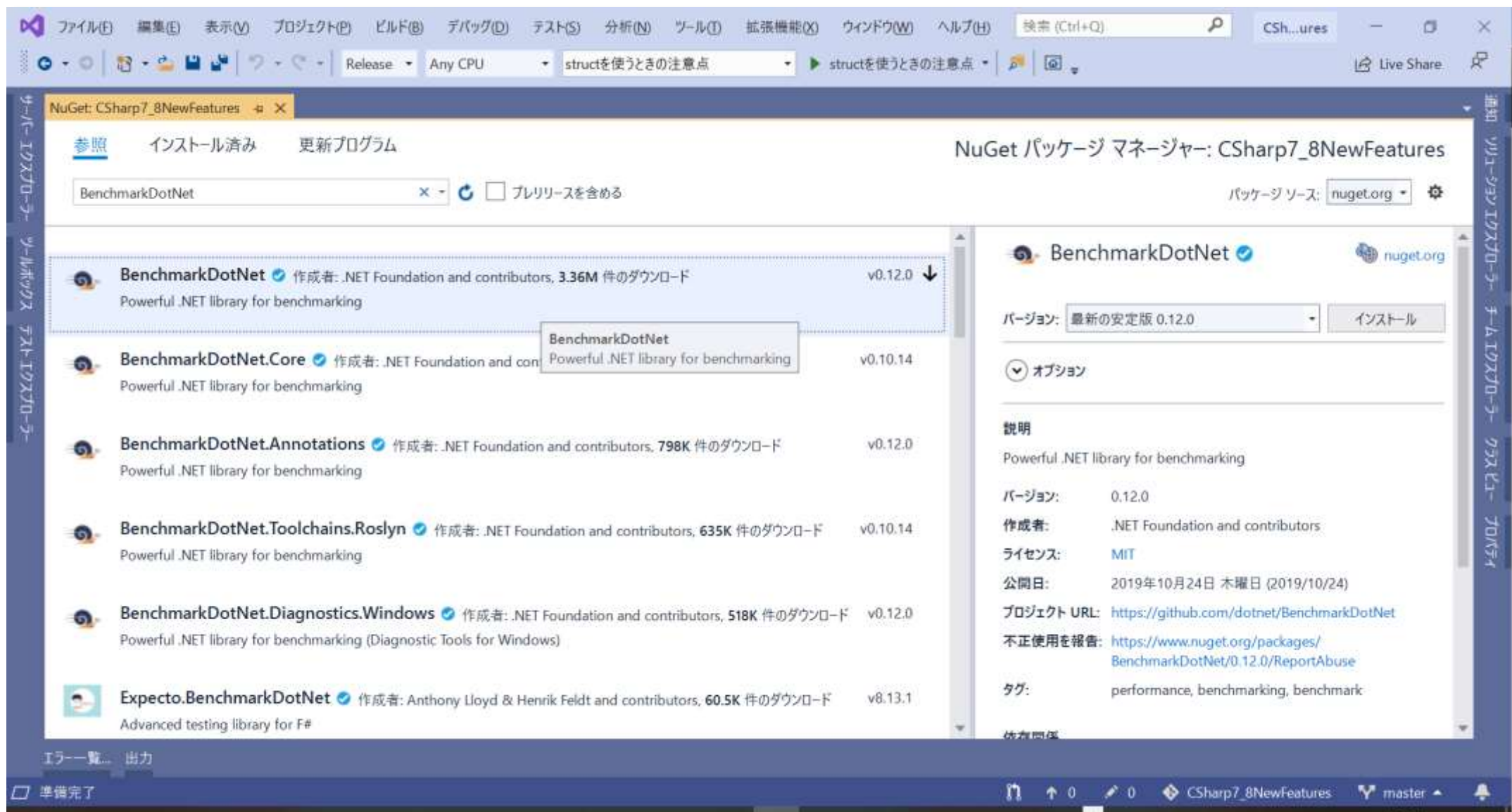
エラー一覧 出力

準備完了

ソース管理に追加

ツール: BenchmarkDotNet

NuGet



ツール: BenchmarkDotNet



コードを参考

- https://github.com/Fujiwo/CSharp7_8NewFeatures
 - [サンプルコード](#)

- [測定結果1 \(xlsx ファイル\)](#)
- [測定結果2 \(xlsxファイル\)](#)



struct を interface で受けると...

```
interface IValuable { int GetValue(); }
```

```
struct FooStruct : IValuable {  
    public int Value { get; set; }  
    public int GetValue() => Value;  
}
```

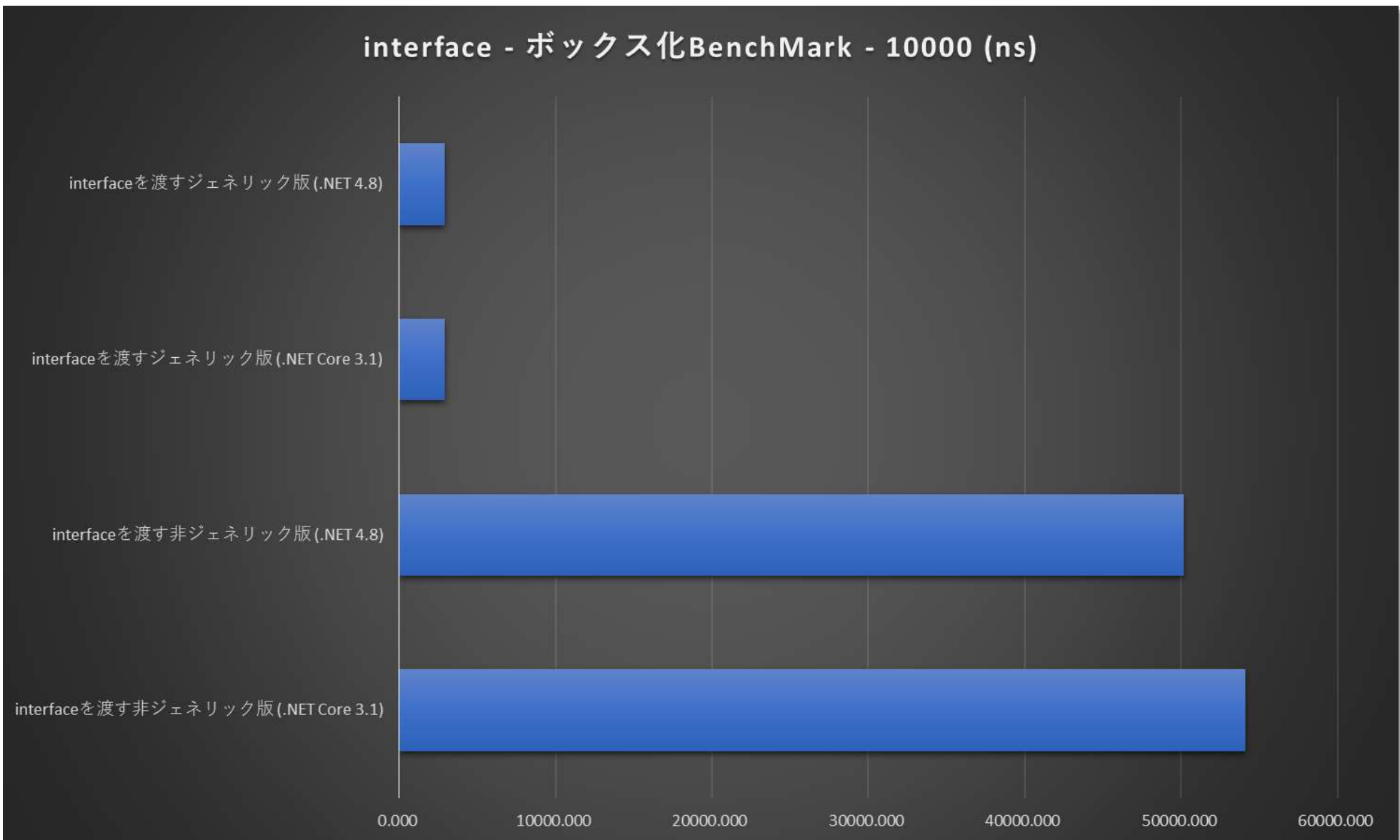
```
public int interfaceを渡す非ジェネリック版() {  
    var item = new FooStruct { Value = 0 };  
    var sum = 0;  
    for (var count = 0; count < Size; count++)  
        sum +=  
            interfaceを受け取る非ジェネリック版(item);  
    return sum;  
}
```

```
static int interfaceを受け取る非ジェネリック版  
(IValuable item) => item.GetValue();
```

```
public int interfaceを渡すジェネリック版() {  
    var item = new FooStruct { Value = 0 };  
    var sum = 0;  
    for (var count = 0; count < Size; count++)  
        sum +=  
            interfaceを受け取るジェネリック版(item);  
    return sum;  
}
```

```
static int interfaceを受け取るジェネリック版<T>(T  
item) where T : IValuable => item.GetValue();
```


ツール: BenchmarkDotNet



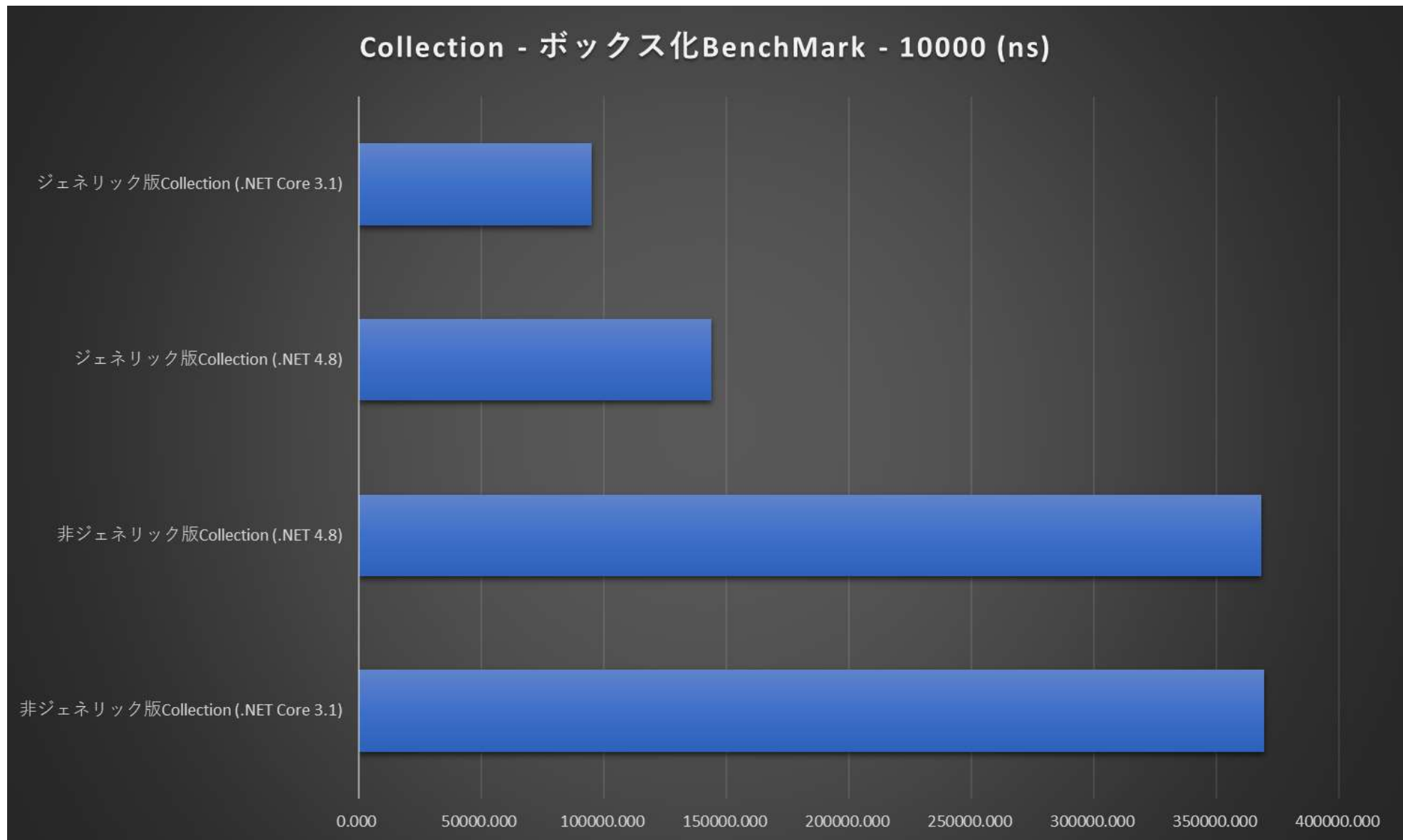
非ジェネリック版のコレクションを使うと…



```
struct FooStruct {  
    public int Value { get; set; }  
}  
  
public int 非ジェネリック版Collection()  
{  
    var list = new ArrayList();  
    for (var count = 0; count < Size; count++)  
        list.Add(new FooStruct { Value = 0 });  
    var sum = 0;  
    for (var index = 0; index < list.Count;  
        index++)  
        sum += ((FooStruct)list[index]).Value;  
    return sum;  
}
```

```
public int ジェネリック版Collection()  
{  
    var list = new List<FooStruct>();  
    for (var count = 0; count < Size; count++)  
        list.Add(new FooStruct { Value = 0 });  
    var sum = 0;  
    for (var index = 0; index < list.Count;  
        index++)  
        sum += list[index].Value;  
    return sum;  
}
```


ツール: BenchmarkDotNet



for/foreach の速度

- どれが速い?
- Array だとどう?

```
static int ListをCountを変数にしてからforする(List<Foo> foos)
{
    var sum = 0;
    var count = foos.Count;
    for (var index = 0; index < count; index++)
        sum += foos[index].Value;
    return sum;
}
```

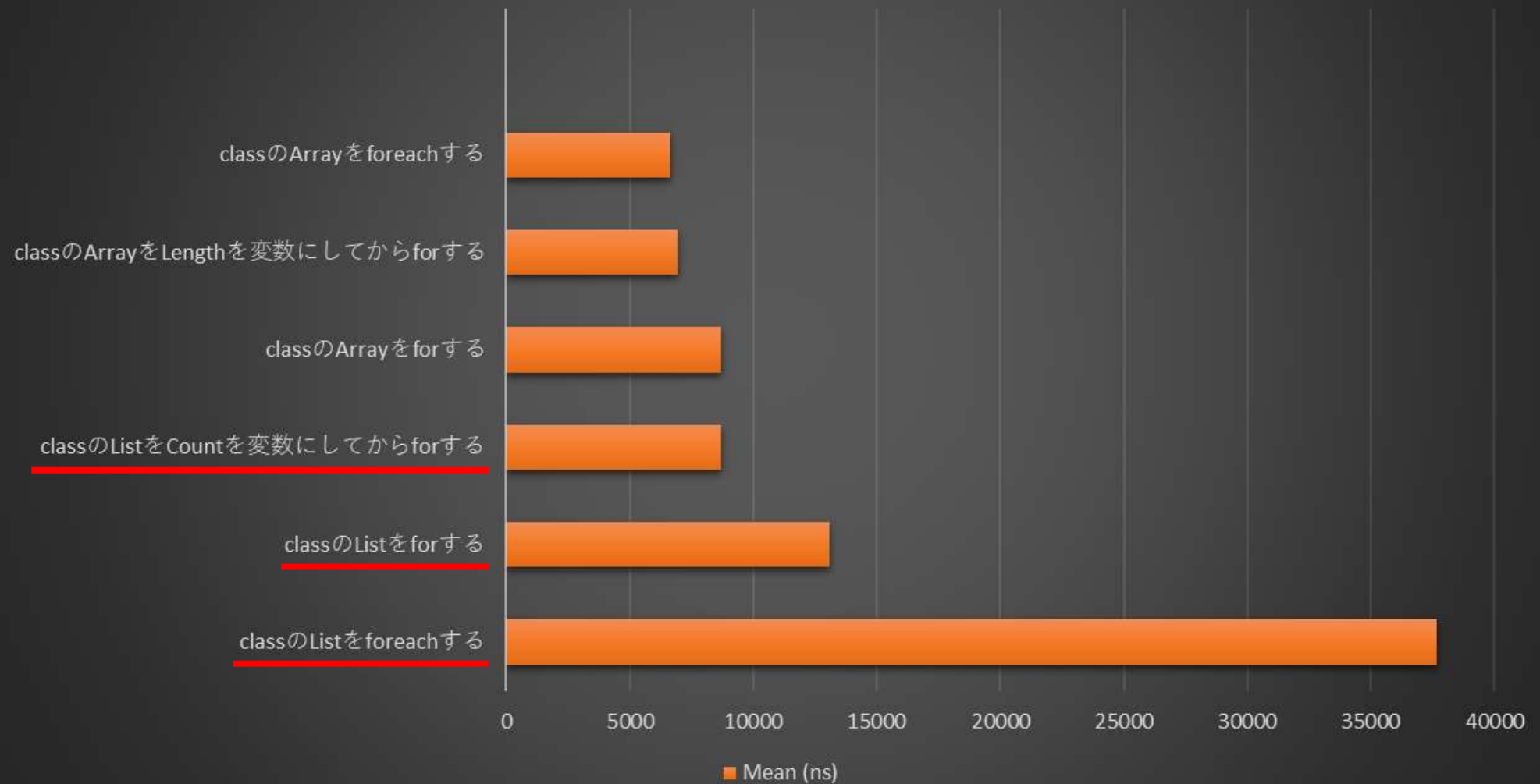
```
static int Listをforする(List<Foo> foos)
{
    var sum = 0;
    for (var index = 0; index < foos.Count; index++)
        sum += foos[index].Value;
    return sum;
}
```

```
static int Listをforeachする(List<Foo> foos)
{
    var sum = 0;
    foreach (var foo in foos)
        sum += foo.Value;
    return sum;
}
```

for/foreach の速度



配列順次アクセスの速度比較 10000要素 (.NET Core)



for/foreach の速度

```
// Enumerator が値型 (struct) の配列
public struct ClassEnumeratorArray<T>
    : IEnumerable<T>
{
    readonly T[] array;
    public StructEnumeratorArray(T[] array)
        => this.array = array;
    public Enumerator GetEnumerator()
        => new Enumerator(array);
}
```

```
public class Enumerator : IEnumerator<T>
{
    readonly T[] array;
    int index;

    internal Enumerator(T[] array)
        => (this.array, index) = (array, -1);
    public T Current => array[index];
    object? IEnumerator.Current => Current;
    public bool MoveNext()
        => ((uint)++index) < (uint)array.Length;
    public void Dispose() {}
    public void Reset() => index = -1;
}
}
```

for/foreach の速度

```
// Enumerator が値型 (struct) の配列
public struct StructEnumeratorArray<T>
{
    readonly T[] array;
    public StructEnumeratorArray(T[] array)
        => this.array = array;
    public Enumerator GetEnumerator()
        => new Enumerator(array);
}
```

```
public struct Enumerator : IEnumerator<T>
{
    readonly T[] array;
    int index;

    internal Enumerator(T[] array)
        => (this.array, index) = (array, -1);
    public T Current => array[index];
    object? IEnumerator.Current => Current;
    public bool MoveNext()
        => ((uint)++index) < (uint)array.Length;
    public void Dispose() {}
    public void Reset() => index = -1;
}
}
```

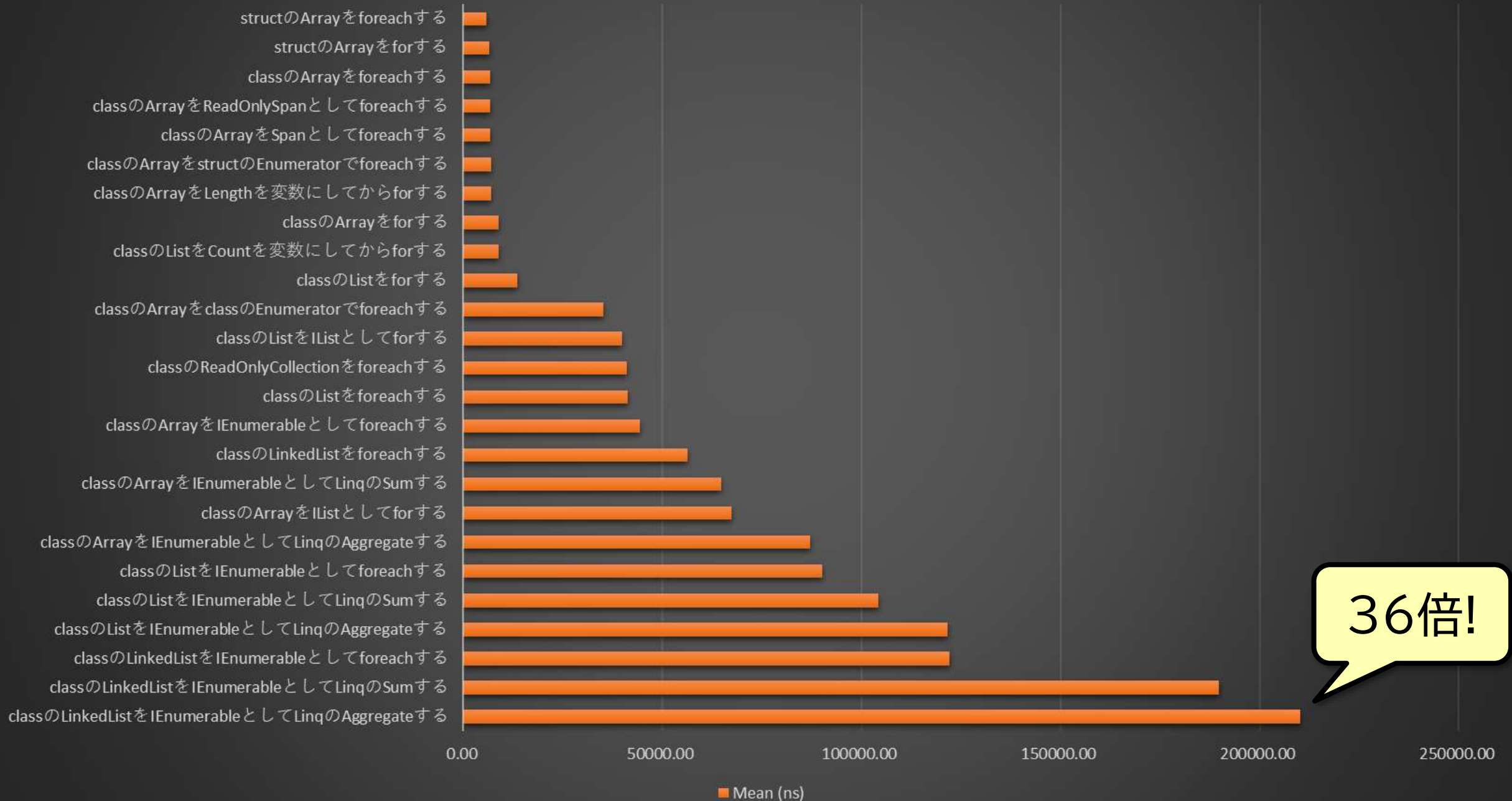
for/foreach の速度



配列順次アクセスの速度比較 10000要素 (.NET Core)

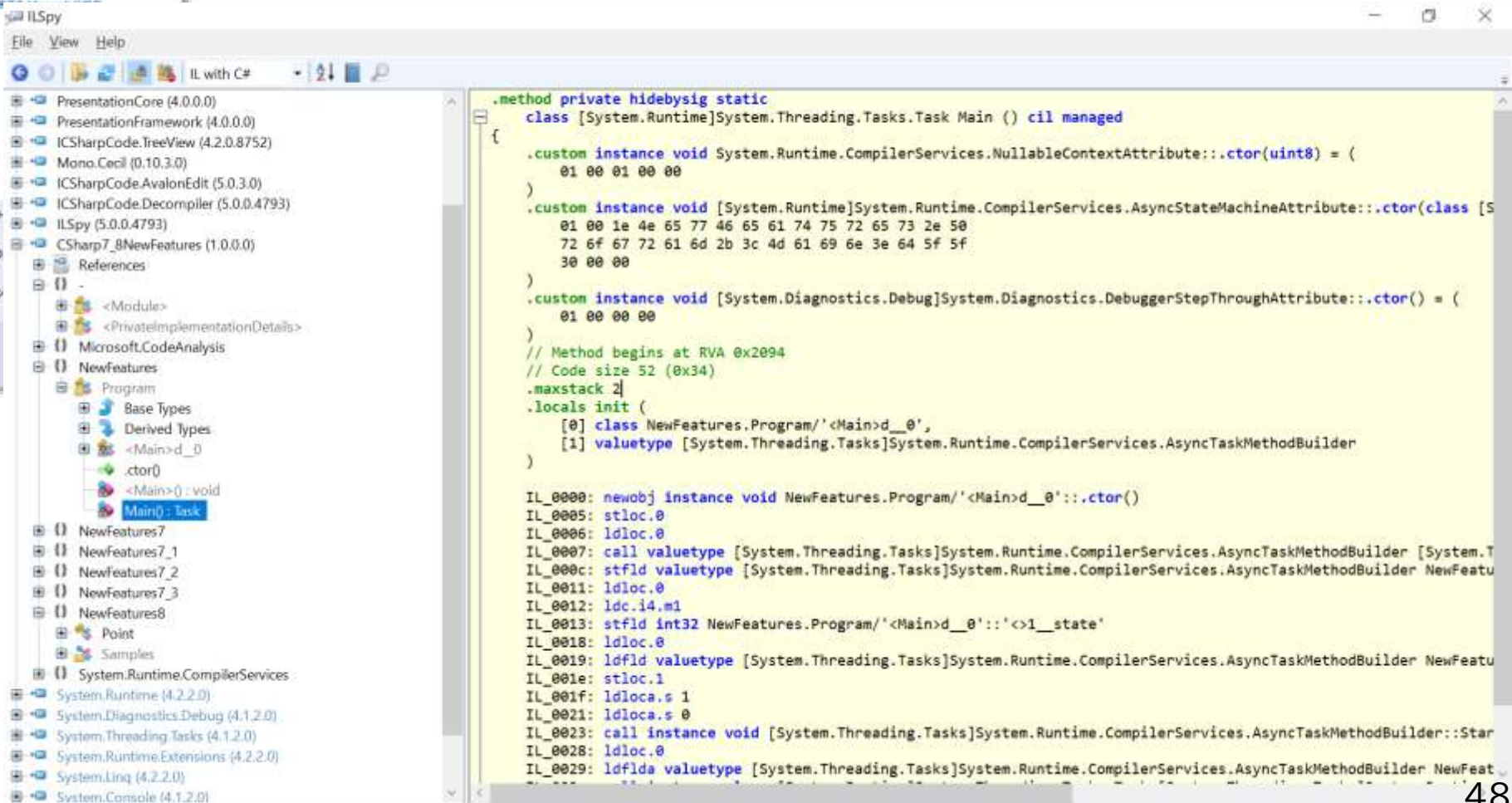
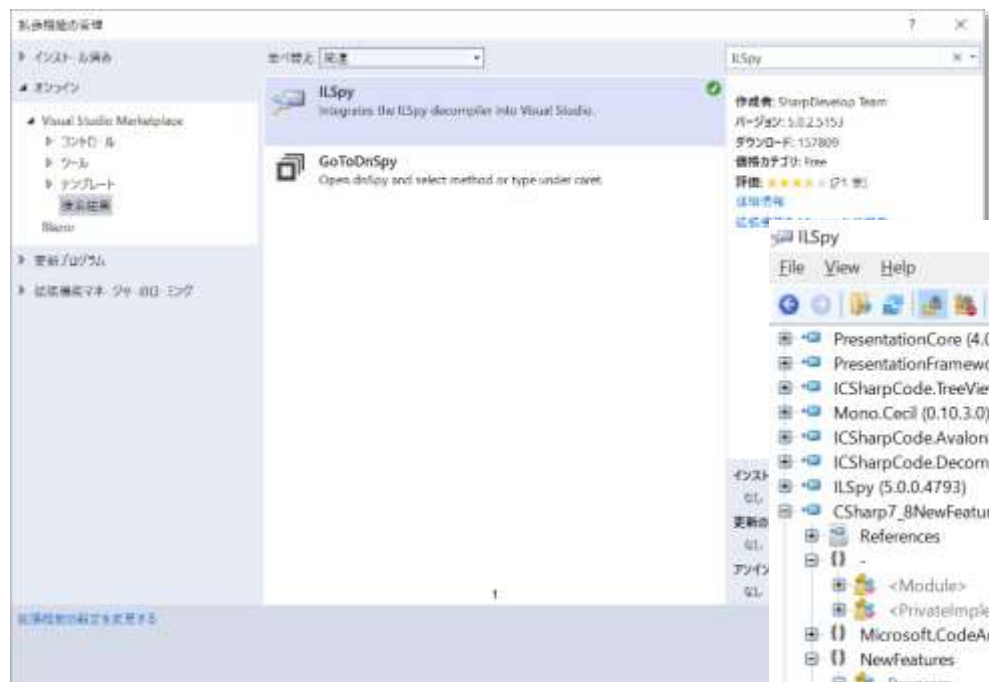


配列(10000要素)の順次アクセスの速度比較 (.NET Core)



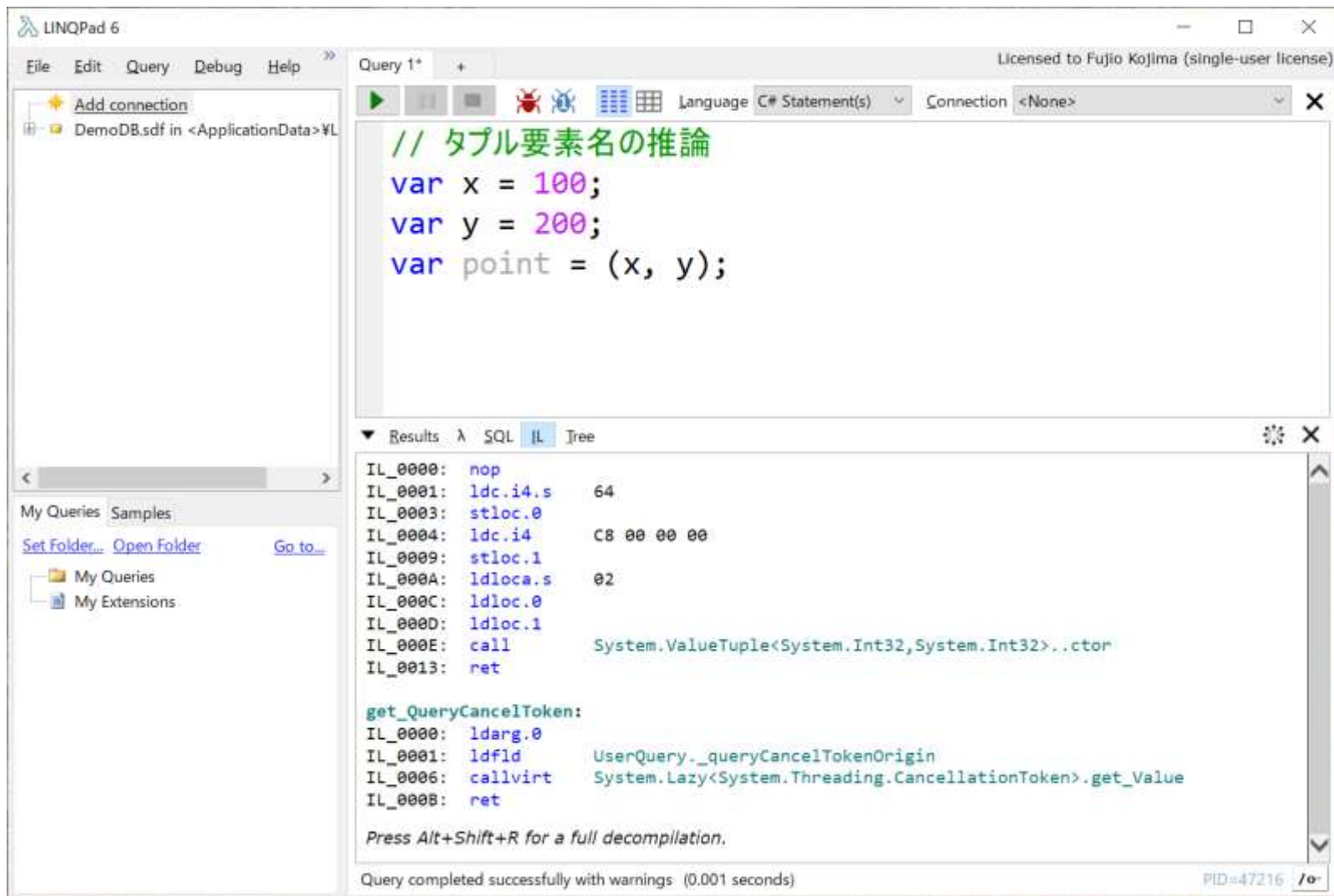
ツール: ILSpy

NuGet



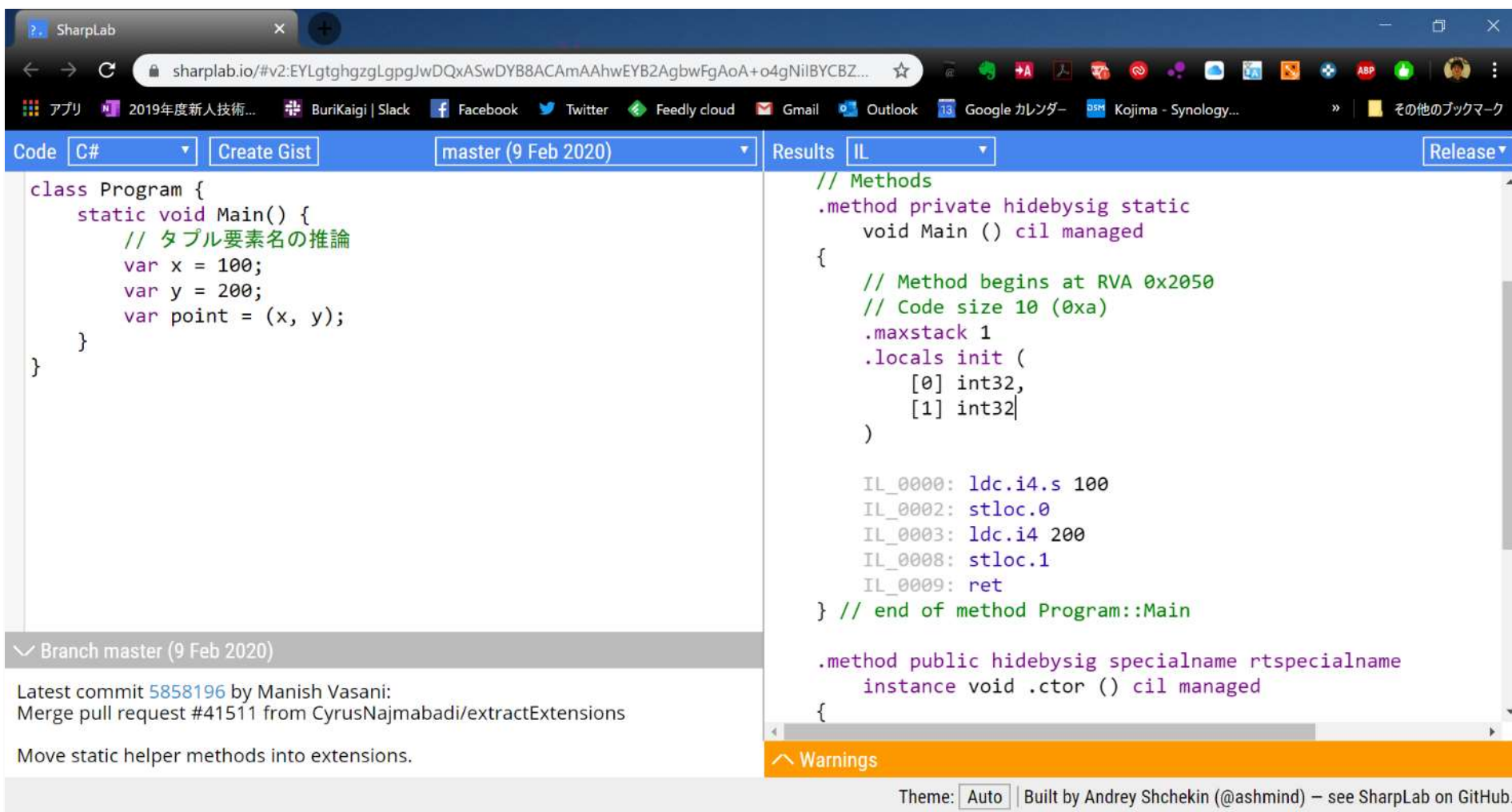
ツール: LINQPad

<https://www.linqpad.net>



ツール: SharpLab

- <https://sharplab.io>



The screenshot displays the SharpLab web application interface. The browser address bar shows the URL `sharplab.io/#v2:EYLgtghgzgLGpgJwDQxASwDYB8ACAmAAhwEYB2AgbwFgAoA+o4gNilBYCBZ...`. The interface is divided into two main panels: 'Code' on the left and 'Results' on the right.

Code Panel (C#):

```
class Program {  
    static void Main() {  
        // タプル要素名の推論  
        var x = 100;  
        var y = 200;  
        var point = (x, y);  
    }  
}
```

Results Panel (IL):

```
// Methods  
.method private hidebysig static  
    void Main () cil managed  
{  
    // Method begins at RVA 0x2050  
    // Code size 10 (0xa)  
    .maxstack 1  
    .locals init (  
        [0] int32,  
        [1] int32  
    )  
  
    IL_0000: ldc.i4.s 100  
    IL_0002: stloc.0  
    IL_0003: ldc.i4 200  
    IL_0008: stloc.1  
    IL_0009: ret  
} // end of method Program::Main  
  
.method public hidebysig specialname rtspecialname  
    instance void .ctor () cil managed  
{
```

Branch master (9 Feb 2020)

Latest commit [5858196](#) by Manish Vasani:
Merge pull request #41511 from CyrusNajmabadi/extractExtensions

Move static helper methods into extensions.

Warnings

Theme: **Auto** | Built by Andrey Shchekin (@ashmind) – see SharpLab on GitHub.

4. C# 7、8の新機能



C# 7、8の新機能

- 値型 (struct) の欠点を解消
 - 値型を immutable
 - 安全に有効活用
 - 値型をコピーさせないことで、高速に

もう「structを使うな」
などと言わせない



C# 7、8の新機能

- 参照型 (class) にも非 null を追加
 - null チェックだらけのコーディングから**安全に**脱却
- その他
 - パターンマッチングなど多数



C# 7、8の新機能

コードを参考

- https://github.com/Fujiwo/CSharp7_8NewFeatures
 - サンプルコード



タプル (ValueTuple)

- 分解
- 値の破棄
- ==, != 比較
- タプル要素名の推論

```
static int Compare(int? x, int? y)
{
    switch ((x, y)) {
        case (int value1, int value2):
            return value1.CompareTo(value2);
        case ({}, null):
        case (null, {}):
        case (null, null):
            return 0;
    }
}
```

分解と Deconstruct



```
var answer    = (dividend / divisor, dividend % divisor);  
var (quotient, remainder) = answer;  
  
var staff = new Person(id: 100, name: "志垣太郎");  
var (id, name) = staff;
```


参照渡し/参照返し



- 参照戻り値
- in 引数
- readonly struct
- ref readonly
- readonly 関数メンバー
- ref struct
- 参照ローカル変数
- ref再代入
- 条件演算子での ref 利用

Span

- stackalloc と Span

```
unsafe {  
    int* array = stackalloc int[size];  
    for (var index = 0; index < size; index++)  
        array[index] = index;  
}
```

```
Span<int> array = stackalloc int[size];  
for (var index = 0; index < array.Length; index++)  
    array[index] = index;
```

ローカル関数

- ローカル関数/静的ローカル関数

ラムダ式と
違って...

- 再帰呼び出し、引数の既定値、ジェネリック、yield return可

```
static int ToDecimal(this IEnumerable<int> @this)
{
    var number = 0;
    // ローカル関数 (number をキャプチャー)
    void Add(int digit) => number = number * 10 + digit;
    @this.ForEach(Add);
    return number;
}
```

nullable 許容参照型

- `<Nullable>enable</Nullable>`

```
<Nullable>enable</Nullable>
```

- `#nullable enable`

```
#nullable enable
```

```
public class Foo { }
```

```
#nullable restore
```

- `#nullable restore`

- `null` 合体代入

```
s ??= "default string";
```

パターンマッチング

- 型 switch
- switch 式
- プロパティ

```
static bool IsNullOrSpace(string? text)
    => text switch {
        null => true ,
        string { Length: 0 } => true ,
        string s when s.Trim().Length == 0
            => true ,
        _ => false
    };
```

インターフェイスのデフォルト実装



```
// インターフェイスのデフォルト実装
interface IEnumerable改<TElement> : IEnumerable<TElement>
{
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

非同期ストリーム

- 非同期ストリーム
- 非同期イテレーター
- 非同期foreach

```
static async IEnumerable<TResult>  
SelectAsync<TElement, TResult>(   
    // 非同期ストリーム  
    this IEnumerable<TElement> @this,  
    Func<TElement, TResult> selector  
)  
{  
    // 非同期foreach  
    await foreach (var item in @this)  
        // 非同期イテレーター  
        yield return selector(item);  
}
```

その他

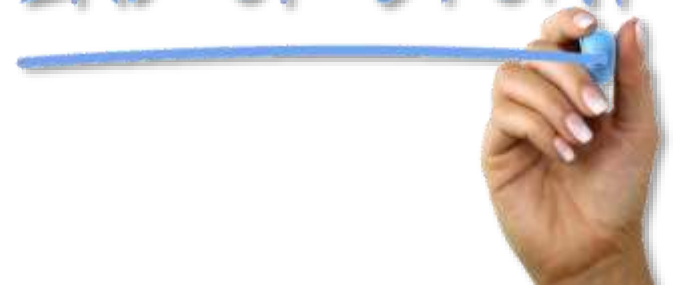


- is での変数宣言
- 数字区切り文字
- 2進数リテラル
- default 式
- 非末尾名前付き引数
- 範囲アクセス
- using 変数宣言
- パターン ベースな using
- 非同期Main

本日の内容

1. C#/.NETの今と近未来
2. C# の値型と参照型
3. C# プログラムの高速化
4. C# 7、8の新機能

END OF STORY



参考文献

- [C# | Wikipedia](#)
- [C# の歴史 - C# ガイド | Microsoft Docs](#)
- [C# 7 の新機能 - C# によるプログラミング入門 | ++C++; // 未確認飛行 C](#)
- [C# 7.1 の新機能 - C# によるプログラミング入門 | ++C++; // 未確認飛行 C](#)
- [C# 7.2 の新機能 - C# によるプログラミング入門 | ++C++; // 未確認飛行 C](#)
- [C# 7.3 の新機能 - C# によるプログラミング入門 | ++C++; // 未確認飛行 C](#)
- [C# 8.0 の新機能 - C# によるプログラミング入門 | ++C++; // 未確認飛行 C](#)
- [今日からできる! 簡単 .NET 高速化 Tips | slideshare](#)
- [foreach の掛け方いろいろ | ++C++; // 未確認飛行 C ブログ](#)