

.NET 9 の パフォーマンス改善

.NET Conf 2024 / dotnet lab

2024/12/21

何縫ねの。





何縫ねの。

NTT コミュニケーションズ
イノベーションセンター

Microsoft MVP for Developer Technologies (2024～)

.NET / Web Development



[nenonaninu](#)



[@nenomake](#)

ブログ <https://blog.neno.dev>
その他 <https://nenodev.com>

OSS

SignalR 周りのいろいろ開発しています

- **TypedSignalR.Client**

- C# の SignalR Client を強く型付けするための Source Generator

- **TypedSignalR.Client.TypeScript**

- TypeScript の SignalR Client を強く型付けするための .NET Tool / library

- **TypedSignalR.Client.DevTools**

- SignalR 向けに Swagger UI 相当の GUI を自動生成する library

- **AspNetCore.SignalR.OpenTelemetry**

- SignalR の OpenTelemetry 対応 + log 強化



Agenda

Dynamic PGO

Tier 0 Optimization

Object Stack Allocation

GC

VM

Threading

Reflection

SearchValues<string>

Span

Dynamic PGO

Dynamic PGO

過去に Dynamic PGO について詳細に語っています

.NET 8 で既定で有効になった Dynamic PGO について

.NET ラボ 2023/10/28

何縫ねの。

Dynamic PGO

過去に Dynamic PGO について詳細に語っています

.NET の Dynamic PGO について
日本語で解説している
最も詳しいスライド

.NET 8 で既定で有効になった Dynamic PGO について

.NET ラボ 2023/10/28

何縫ねの。

Dynamic PGO

- .NET 8 まで
 - Devirtualization
 - virtual method, interface method, delegate の呼び出し高速化
 - Inlining
- .NET 9 から
 - Cast の高速化
 - (T)obj
 - obj is T

Dynamic PGO

事前知識

```
class MyClass  
{  
    public int Value = 99;  
}
```

MyClass at 0x24F050A9268

Hex ▼

header

type handle

Value

00	00	00	00	00	00	00	00	A0	CC	27	78	FF	7F	00	00	63	00	00	00	00	00	00	00
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Dynamic PGO

事前知識

```
class MyClass  
{  
    public int Value = 99;  
}
```

オブジェクトの型情報は
Header の後ろに存在

MyClass at 0x24F050A9268

Hex ▼

header	type handle	Value	
00 00 00 00 00 00 00 00	A0 CC 27 78 FF 7F 00 00	63 00 00 00	00 00 00 00

Dynamic PGO

Cast に関する最適化

```
public class Tests
{
    private A _obj = new C();

    [Benchmark]
    public bool IsInstanceOf() => _obj is B;

    public class A { }
    public class B : A { }
    public class C : B { }
}
```

Dynamic PGO

.NET 8

```
; Tests.IsInstanceOf()
    push    rax
    mov     rsi,[rdi+8]
    mov     rdi,offset MT_Tests+B
    call    qword ptr [7F3D91524360]; CastHelpers.IsInstanceOfClass(Void*, System.Object)
    test    rax,rax
    setne   al
    movzx   eax,al
    add     rsp,8
    ret
; Total bytes of code 35
```

Dynamic PGO

.NET 8

```
; Tests.IsInstanceOf()
  push    rax
  mov     rsi,[rdi+8]
  mov     rdi,offset MT_Tests+B
  call   qword ptr [7F3D91524360]; CastHelpers.IsInstanceOfClass(Void*, System.Object)
  test   rax,rax
  setne  al
  movzx  eax,al
  add    rsp,8
  ret
; Total bytes of code 35
```

**必ず IsInstanceOfClass を呼び出して
obj が B かを判定**

Dynamic PGO

.NET 8

```
[DebuggerHidden]  
private static object? IsInstanceOfClass(void* toTypeHnd, object? obj)  
{
```

```
; Tests.IsInstanceOf()  
    push    rax  
    mov     rsi,[rdi+8]  
    mov     rdi,offset MT_Tests+B  
    call    qword ptr [7F3D91524360]; CastHelpers.IsInstanceOfClass(Void*, System.Object)  
    test    rax,rax  
    setne   al  
    movzx   eax,al  
    add     rsp,8  
    ret  
; Total bytes of code 35
```

**必ず IsInstanceOfClass を呼び出して
obj が B かを判定**

Dynamic PGO

.NET 8

B の TypeHandle と _obj が渡される

```
[DebuggerHidden]
private static object? IsInstanceOfClass(void* toTypeHnd, object? obj)
{
```

```
; Tests.IsInstanceOf()
    push    rax
    mov     rsi,[rdi+8]
    mov     rdi,offset MT_Tests+B
    call    qword ptr [7F3D91524360]; CastHelpers.IsInstanceOfClass(Void*, System.Object)
    test    rax,rax
    setne   al
    movzx   eax,al
    add     rsp,8
    ret
; Total bytes of code 35
```

必ず IsInstanceOfClass を呼び出して
obj が B かを判定

Dynamic PGO

.NET 8

B の TypeHandle と _obj が渡される

```
[DebuggerHidden]  
private static object? IsInstanceOfClass(void* toTypeHnd, object? obj)  
{
```

```
; Tests.IsInstanceOf()  
    push    rax  
    mov     rsi,[rdi+8]  
    mov     rdi,offset MT_Tests+B  
    call   qword ptr [7F3D91524360]; CastHelpers.IsInstanceOfClass(Void*, System.Object)  
    test   rax,rax  
    setne  al  
    movzx  eax,al  
    add    rsp,8  
    ret  
; Total bytes of code 35
```

必ず IsInstanceOfClass を呼び出して
obj が B かを判定

継承等の関係上単純な比較にはならない

Dynamic PGO

IsInstanceOfClass の実装詳細 或いはそもそも TypeHandle とは何か

- Type 毎に MethodTable が1つ存在する
- TypeHandle は MethodTable へのポインタ
- MethodTable には Type に関する様々な情報
 - 型の種別(class, struct, interface)
 - 実装している interface の数
 - 等々

The ObjectInstance of smallObj contains the TypeHandle that points to the MethodTable of the corresponding type. There will be one MethodTable for each declared type and all the object instances of the same type will point to the same MethodTable. This will contain information about the kind of type (interface, abstract class, concrete class, COM Wrapper, and proxy), the number of interfaces implemented, the interface map for method dispatch, the number of slots in the method table, and a table of slots that point to the implementations.

```
[DebuggerHidden]
private static object? IsInstanceOfClass(void* toTypeHnd, object? obj)
{
    if (obj == null || RuntimeHelpers.GetMethodTable(obj) == toTypeHnd)
        return obj;

    MethodTable* mt = RuntimeHelpers.GetMethodTable(obj)->ParentMethodTable;
    for (; ; )
    {
        if (mt == toTypeHnd)
            goto done;

        if (mt == null)
            break;

        mt = mt->ParentMethodTable;
        if (mt == toTypeHnd)
            goto done;

        if (mt == null)
            break;

        mt = mt->ParentMethodTable;
        if (mt == toTypeHnd)
            goto done;

        if (mt == null)
            break;

        mt = mt->ParentMethodTable;
        if (mt == toTypeHnd)
            goto done;

        if (mt == null)
            break;

        mt = mt->ParentMethodTable;
    }

    #if FEATURE_TYPEEQUIVALENCE
        // this helper is not supposed to be used with type-equivalent "to" type.
        Debug.Assert(!((MethodTable*)toTypeHnd)->HasTypeEquivalence);
    #endif // FEATURE_TYPEEQUIVALENCE

    obj = null;

done:
    return obj;
}
```

Dynamic PGO

.NET 9

```
; Tests.IsInstanceOf()
    push    rbp
    mov     rbp, rsp
    mov     rsi, [rdi+8]
    mov     rcx, rsi
    test   rcx, rcx
    je     short M00_L00
    mov     rax, offset MT_Tests+C
    cmp    [rcx], rax
    jne    short M00_L01
M00_L00:
    test   rcx, rcx
    setne  al
    movzx  eax, al
    pop    rbp
    ret
M00_L01:
    mov     rdi, offset MT_Tests+B
    call   CastHelpers.IsInstanceOfClass(Void*, System.Object)
    mov     rcx, rax
    jmp    short M00_L00
; Total bytes of code 62
```

Dynamic PGO

.NET 9

```
; Tests.IsInstanceOf()
    push    rbp
    mov     rbp, rsp
    mov     rsi, [rdi+8]
    mov     rcx, rsi
    test    rcx, rcx
    je      short M00_L00
    mov     rax, offset MT_Tests+C
    cmp     [rcx], rax
    jne     short M00_L01
M00_L00:
    test    rcx, rcx
    setne   al
    movzx   eax, al
    pop     rbp
    ret
M00_L01:
    mov     rdi, offset MT_Tests+B
    call    CastHelpers.IsInstanceOfClass(Void*, System.Object)
    mov     rcx, rax
    jmp     short M00_L00
; Total bytes of code 62
```

重要なのはここ。
JIT は C が B から継承されている事を知っている。
なので obj が C かを直接判定(cmp)し高速化

Dynamic PGO

.NET 9

```
; Tests.IsInstanceOf()
    push    rbp
    mov     rbp, rsp
    mov     rsi, [rdi+8]
    mov     rcx, rsi
    test    rcx, rcx
    je      short M00_L00
    mov     rax, offset MT_Tests+C
    cmp     [rcx], rax
    jne     short M00_L01
M00_L00:
    test    rcx, rcx
    setne   al
    movzx   eax, al
    pop     rbp
    ret
M00_L01:
    mov     rdi, offset MT_Tests+B
    call    CastHelpers.IsInstanceOfClass(Void*, System.Object)
    mov     rcx, rax
    jmp     short M00_L00
; Total bytes of code 62
```

重要なのはここ。
JIT は C が B から継承されている事を知っている。
なので obj が C かを直接判定(cmp)し高速化

C であれば下にそのまま処理が進み
C ではなかったら M00_L01 に飛ぶ

Tier 0 Optimization


流石にこんなコード書く人はいないはずですが...

Analyzer にも怒られますしね！

```
static void M(int value)
```

```
{  
    ... ArgumentNullException.ThrowIfNull(value);  
}
```



 class System.ArgumentNullException

The exception that is thrown when a null reference ([Nothing](#) in Visual Basic) is passed to a method that does not accept it as a valid argument.

[GitHub Examples and Documentation](#) (Alt+O)

[CA2264: Calling 'ArgumentNullException.ThrowIfNull' and passing a non-nullable value is a no-op](#)

[Show potential fixes](#) (Alt+Enter or Ctrl+.)

Tier 0 Optimization

でも generics なら？

まあ、あるよね。

```
M(99);

static void M<T>(T value)
{
    ... ArgumentNullException.ThrowIfNull(value);
}
```

Tier 0 Optimization

でも generics なら？

まあ、あるよね。

.NET 8 までは、Tier 0 では最適化されず boxing が発生
Tier 1 で最適化され、呼び出し自体が消し飛ぶ

```
M(99);  
  
static void M<T>(T value)  
{  
    ... ArgumentNullException.ThrowIfNull(value);  
}
```

Tier 0 Optimization

でも generics なら？

まあ、あるよね。

```
M(99);
```

```
static void M<T>(T value)
{
    ... ArgumentNullException.ThrowIfNull(value);
}
```

.NET 9 からは Tier 0 で呼び出しが消し飛ぶ

.NET 8 までは、Tier 0 では最適化されず boxing が発生
Tier 1 で最適化され、呼び出し自体が消し飛ぶ

Object Stack Allocation

heap に allocation が発生する所を stack に allocation する最適化

```
public class Tests
{
    [Benchmark]
    public int GetValue() => new MyObj(42).Value;

    private class MyObj
    {
        public MyObj(int value) => Value = value;
        public int Value { get; }
    }
}
```

Object Stack Allocation

heap に allocation が発生する所を stack に allocation する最適化

```
public class Tests
{
    [Benchmark]
    public int GetValue() => new MyObj(42).Value;

    private class MyObj
    {
        public MyObj(int value) => Value = value;
        public int Value { get; }
    }
}
```

普通に考えれば new MyObj() で
heap に allocation が走る

Object Stack Allocation

.NET 8

```
; Tests.GetValue()  
    push    rax  
    mov     rdi, offset MT_Tests+MyObj  
    call   CORINFO_HELP_NEWSFAST  
    mov     dword ptr [rax+8], 2A  
    mov     eax, [rax+8]  
    add     rsp, 8  
    ret  
; Total bytes of code 31
```

Object Stack Allocation

.NET 8

```
; Tests.GetValue()  
    push    rax  
    mov     rdi, offset MT_Tests+MyObj  
    call   CORINFO_HELP_NEWSFAST  
    mov     dword ptr [rax+8], 2A  
    mov     eax, [rax+8]  
    add     rsp, 8  
    ret  
; Total bytes of code 31
```

CORINFO_HELP_NEWSFAST で
allocate

Object Stack Allocation

.NET 8

```
; Tests.GetValue()  
    push    rax  
    mov     rdi, offset MT_Tests+MyObj  
    call   CORINFO_HELP_NEWSFAST  
    mov     dword ptr [rax+8], 2A  
    mov     eax, [rax+8]  
    add     rsp, 8  
    ret  
; Total bytes of code 31
```

new small object fast
の意(おそらく)

CORINFO_HELP_NEWSFAST で
allocate

Object Stack Allocation

.NET 9

```
; Tests.GetValue()  
    mov     eax, 2A  
    ret  
; Total bytes of code 6
```

Method	Runtime	Mean	Ratio	Code Size	Allocated	Alloc Ratio
GetValue	.NET 8.0	3.6037 ns	1.00	31 B	24 B	1.00
GetValue	.NET 9.0	0.0519 ns	0.01	6 B	–	0.00

Object Stack Allocation

.NET 9

```
; Tests.GetValue()  
    mov     eax, 2A  
    ret  
; Total bytes of code 6
```

Heap に対する allocation なし

Method	Runtime	Mean	Ratio	Code Size	Allocated	Alloc Ratio
GetValue	.NET 8.0	3.6037 ns	1.00	31 B	24 B	1.00
GetValue	.NET 9.0	0.0519 ns	0.01	6 B	–	0.00

Object Stack Allocation

.NET 9

```
; Tests.GetValue()  
    mov     eax, 2A  
    ret  
; Total bytes of code 6
```

Object stack allocation + inlining
により、定数を返すだけになる

Heap に対する allocation なし

Method	Runtime	Mean	Ratio	Code Size	Allocated	Alloc Ratio
GetValue	.NET 8.0	3.6037 ns	1.00	31 B	24 B	1.00
GetValue	.NET 9.0	0.0519 ns	0.01	6 B	–	0.00

Object Stack Allocation

.NET 9

```
; Tests.GetValue()  
    mov     eax, 2A  
    ret  
; Total bytes of code 6
```

Object stack allocation + inlining
により、定数を返すだけになる

Heap に対する allocation なし

流石にパフォーマンス
アピール用のコード
過ぎないか？

Method	Runtime	Mean	Ratio	Code Size	Allocated	Alloc Ratio
GetValue	.NET 8.0	3.6037 ns	1.00	31 B	24 B	1.00
GetValue	.NET 9.0	0.0519 ns	0.01	6 B	–	0.00

Object Stack Allocation

.NET 9

```
; Tests.GetValue()  
    mov     eax, 2A  
    ret  
; Total bytes of code 6
```

Object stack allocation + inlining
により、定数を返すだけになる

Heap に対する allocation なし

流石にパフォーマンス
アピール用のコード
過ぎないか？

実際のユースケースでは
どうだろうか？

Method	Runtime	Mean	Ratio	Code Size	Allocated	Alloc Ratio
GetValue	.NET 8.0	3.6037 ns	1.00	31 B	24 B	1.00
GetValue	.NET 9.0	0.0519 ns	0.01	6 B	-	0.00

Object Stack Allocation

そもそも object stack allocation の対象となるパターンは？

オブジェクト参照が現在のスタックフレームから離れないことを
簡単に証明できるケースのみ最適化

Object Stack Allocation

そもそも object stack allocation の対象となるパターンは？

オブジェクト参照が現在のスタックフレームから離れないことを簡単に証明できるケースのみ最適化

メソッド呼び出し間を分析 (interprocedural analysis) して最適化などは行わない

Object Stack Allocation

こういうユースケースで嬉しい

```
public class Tests
{
    [Benchmark]
    public void Test()
    {
        Dispose1<MyStruct>(default);
        Dispose2<MyStruct>(default);
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private bool Dispose1<T>(T o)
    {
        bool disposed = false;
        if (o is IDisposable disposable)
        {
            disposable.Dispose();
            disposed = true;
        }
        return disposed;
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private bool Dispose2<T>(T o)
    {
        bool disposed = false;
        if (o is IDisposable)
        {
            ((IDisposable)o).Dispose();
            disposed = true;
        }
        return disposed;
    }

    private struct MyStruct : IDisposable
    {
        public void Dispose() { }
    }
}
```

Object Stack Allocation

こういうユースケースで嬉しい

.NET 8

```
; Tests.Dispose1[[Tests+MyStruct, benchmarks]](MyStruct)
    push    rbx
    mov     rdi,offset MT_Tests+MyStruct
    call   CORINFO_HELP_NEWSFAST
    add    rax,8
    mov    ebx,[rsp+10]
    mov    [rax],bl
    mov    eax,1
    pop    rbx
    ret
; Total bytes of code 33

; Tests.Dispose2[[Tests+MyStruct, benchmarks]](MyStruct)
    mov    eax,1
    ret
; Total bytes of code 6
```

```
public class Tests
{
    [Benchmark]
    public void Test()
    {
        Dispose1<MyStruct>(default);
        Dispose2<MyStruct>(default);
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private bool Dispose1<T>(T o)
    {
        bool disposed = false;
        if (o is IDisposable disposable)
        {
            disposable.Dispose();
            disposed = true;
        }
        return disposed;
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private bool Dispose2<T>(T o)
    {
        bool disposed = false;
        if (o is IDisposable)
        {
            ((IDisposable)o).Dispose();
            disposed = true;
        }
        return disposed;
    }

    private struct MyStruct : IDisposable
    {
        public void Dispose() { }
    }
}
```

Object Stack Allocation

こういうユースケースで嬉しい

.NET 8

```
; Tests.Dispose1[[Tests+MyStruct, benchmarks]](MyStruct)
    push    rbx
    mov     rdi,offset MT_Tests+MyStruct
    call   CORINFO_HELP_NEWSFAST
    add    rax,8
    mov    ebx,[rsp+10]
    mov    [rax],bl
    mov    eax,1
    pop    rbx
    ret
; Total bytes of code 33

; Tests.Dispose2[[Tests+MyStruct, benchmarks]](MyS
    mov    eax,1
    ret
; Total bytes of code 6
```

MyStruct.Dispose() の実装が
空なので inlining され
ここまで圧縮される

```
public class Tests
{
    [Benchmark]
    public void Test()
    {
        Dispose1<MyStruct>(default);
        Dispose2<MyStruct>(default);
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private bool Dispose1<T>(T o)
    {
        bool disposed = false;
        if (o is IDisposable disposable)
        {
            disposable.Dispose();
            disposed = true;
        }
        return disposed;
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private bool Dispose2<T>(T o)
    {
        bool disposed = false;
        if (o is IDisposable)
        {
            ((IDisposable)o).Dispose();
            disposed = true;
        }
        return disposed;
    }

    private struct MyStruct : IDisposable
    {
        public void Dispose() { }
    }
}
```

Object Stack Allocation

こういうユースケースで嬉しい

.NET 8

```
; Tests.Dispose1[[Tests+MyStruct, benchmarks]](MyStruct)
    push    rbx
    mov     rdi,offset MT_Tests+MyStruct
    call   CORINFO_HELP_NEWSFAST
    add    rax,8
    mov    ebx,[rsp+10]
    mov    [rax],bl
    mov    eax,1
    pop    rbx
    ret
; Total bytes of code 33

; Tests.Dispose2[[Tests+MyStruct, benchmarks]](MyS
    mov    eax,1
    ret
; Total bytes of code 6
```

MyStruct.Dispose() の実装が
空なので inlining され
ここまで圧縮される

```
public class Tests
{
    [Benchmark]
    public void Test()
    {
        Dispose1<MyStruct>(default);
        Dispose2<MyStruct>(default);
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private bool Dispose1<T>(T o)
    {
        bool disposed = false;
        if (o is IDisposable disposable)
        {
            disposable.Dispose();
            disposed = true;
        }
        return disposed;
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private bool Dispose2<T>(T o)
    {
        bool disposed = false;
        if (o is IDisposable disposable)
        {
            ((IDisposable)o).Dispose();
            disposed = true;
        }
        return disposed;
    }

    private struct MyStruct : IDisposable
    {
        public void Dispose() { }
    }
}
```

using で dispose する場合は
この展開のされ方になるが
boxing は発生しない

Object Stack Allocation

こういうユースケースで嬉しい

.NET 8

```
; Tests.Dispose1[[Tests+MyStruct, benchmarks]](MyStruct)
push    rbx
mov     rdi,offset MT_Tests+MyStruct
call   CORINFO_HELP_NEWSFAST
add     rax,8
mov     ebx,[rsp+10]
mov     [rax],bl
mov     eax,1
pop     rbx
ret
; Total bytes of code 33

; Tests.Dispose2[[Tests+MyStruct, benchmarks]](MyS
mov     eax,1
ret
; Total bytes of code 6
```

.NET 9

```
; Tests.Dispose1[[Tests+MyStruct, benchmarks]](MyStruct)
mov     eax,1
ret
; Total bytes of code 6

; Tests.Dispose2[[Tests+MyStruct, benchmarks]](MyStruct)
mov     eax,1
ret
; Total bytes of code 6
```

MyStruct.Dispose() の実装が
空なので inlining され
ここまで圧縮される

```
public class Tests
```

```
{
    [Benchmark]
    public void Test()
    {
        Dispose1<MyStruct>(default);
        Dispose2<MyStruct>(default);
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private bool Dispose1<T>(T o)
    {
        bool disposed = false;
        if (o is IDisposable disposable)
        {
            disposable.Dispose();
            disposed = true;
        }
        return disposed;
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private bool Dispose2<T>(T o)
    {
        bool disposed = false;
        if (o is IDisposable disposable)
        {
            ((IDisposable)o).Dispose();
            disposed = true;
        }
        return disposed;
    }

    private struct MyStruct : IDisposable
    {
        public void Dispose() { }
    }
}
```

using で dispose する場合は
この展開のされ方になるが
boxing は発生しない

Object Stack Allocation

こういうユースケースで嬉しい

.NET 8

```
; Tests.Dispose1[[Tests+MyStruct, benchmarks]](MyStruct)
push    rbx
mov     rdi,offset MT_Tests+MyStruct
call   CORINFO_HELP_NEWSFAST
add     rax,8
mov     ebx,[rsp+10]
mov     [rax],bl
mov     eax,1
pop     rbx
ret
; Total bytes of code 33

; Tests.Dispose2[[Tests+MyStruct, benchmarks]](MyS
mov     eax,1
ret
; Total bytes of code 6
```

.NET 9

```
; Tests.Dispose1[[Tests+MyStruct, benchmarks]](MyStruct)
mov     eax,1
ret
; Total bytes of code 6

; Tests.Dispose2[[Tests+MyStruct, benchmarks]](MyStruct)
mov     eax,1
ret
; Total bytes of code 6
```

MyStruct.Dispose() の実装が
空なので inlining され
ここまで圧縮される

disposable はこのフレーム内で
参照が握られたりしない事が証明できるため、
object stack allocation による最適化の対象となる

using で dispose する場合は
この展開のされ方になるが
boxing は発生しない

```
public class Tests
{
    [Benchmark]
```

```
private bool Dispose1<T>(T o)
{
    bool disposed = false;
    if (o is IDisposable disposable)
    {
        disposable.Dispose();
        disposed = true;
    }
    return disposed;
}
```

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private bool Dispose2<T>(T o)
{
    bool disposed = false;
    if (o is IDisposable disposable)
    {
        ((IDisposable)o).Dispose();
        disposed = true;
    }
    return disposed;
}
```

```
private struct MyStruct : IDisposable
{
    public void Dispose() { }
}
```

Object Stack Allocation

こういうユースケースで嬉しい

.NET 8

```
; Tests.Dispose1[[Tests+MyStruct, benchmarks]](MyStruct)
push    rbx
mov     rdi,offset MT_Tests+MyStruct
call   CORINFO_HELP_NEWSFAST
add     rax,8
mov     ebx,[rsp+10]
mov     [rax],bl
mov     eax,1
pop     rbx
ret
; Total bytes of code 33

; Tests.Dispose2[[Tests+MyStruct, benchmarks]](MyS
mov     eax,1
ret
; Total bytes of code 6
```

.NET 9

```
; Tests.Dispose1[[Tests+MyStruct, benchmarks]](MyStruct)
mov     eax,1
ret
; Total bytes of code 6

; Tests.Dispose2[[Tests+MyStruct, benchmarks]](MyStruct)
mov     eax,1
ret
; Total bytes of code 6
```

MyStruct.Dispose() の実装が
空なので inlining され
ここまで圧縮される

disposable はこのフレーム内で
参照が握られたりしない事が証明できるため、
object stack allocation による最適化の対象となる

using で dispose する場合は
この展開のされ方になるが
boxing は発生しない

Method	Runtime	Mean	Ratio	Code Size	Allocated	Alloc Ratio
Test	.NET 8.0	5.726 ns	1.00	94 B	24 B	1.00
Test	.NET 9.0	2.095 ns	0.37	45 B	-	0.00

```
public class Tests
{
    [Benchmark]
```

```
private bool Dispose1<T>(T o)
{
    bool disposed = false;
    if (o is IDisposable disposable)
    {
        disposable.Dispose();
        disposed = true;
    }
    return disposed;
}
```

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private bool Dispose2<T>(T o)
{
    bool disposed = false;
    if (o is IDisposable disposable)
    {
        ((IDisposable)o).Dispose();
        disposed = true;
    }
    return disposed;
}

private struct MyStruct : IDisposable
{
    public void Dispose() { }
}
```

GC

GC の種別

Workstation GC

- メモリの消費量を抑える
- 一回あたりの GC の時間を抑える
 - タイムアウト有り
 - GUI をフリーズさせないため
- CPU のコア数が 1 つなら必ず Workstation GC
- ヒープは 1 つ

Server GC

- スループットの最大化
- メモリの消費量は大きめ
- 一回あたりの GC の時間長め
 - タイムアウト無し
- ASP.NET Core の既定
 - ただし CPU コアが2つ以上の場合
- CPUコア数=ヒープ数
 - ただし既定の場合

GC

AdaptationMode

- スループットは大事だけど、メモリの消費量も抑えたい
- DATAS (Dynamically Adapting To Application Sizes) の導入
 - .NET 8 で導入された、しかし既定で有効にはなっていなかった
 - .NET 9 から既定で有効に
- DATAS のお仕事は「アプリケーションのサイズに適応する」事
 - ヒープ数の調整
 - スループットと消費するメモリと GC の頻度に応じて調整
 - 完全な compaction を走らせるための allocation budget の調整

GC

AdaptationMode

- スループットは大事だけど、メモリの消費量も抑えたい
- DATAS (Dynamically Adapting To Application Sizes) の導入
 - .NET 8 で導入された、しかし既定で有効にはなっていなかった
 - .NET 9 から既定で有効に
- DATAS のお仕事は「アプリケーションのサイズに適応する」事
 - ヒープ数の調整
 - スループットと消費するメモリと GC の頻度に応じて調整
 - 完全な compaction を走らせるための allocation budget の調整

完全な compaction を走らせないと
何時まで経っても余計なメモリを食いつぱなし

GC

メモリの消費量気にせず
最大のスループットを求めるなら
今のところ AdaptationMode は切った方が良い

AdaptationMode

- スループットは大事だけど、メモリの消費量も抑えたい
- DATAS (Dynamically Adapting To Application Sizes) の導入
 - .NET 8 で導入された、しかし既定で有効にはなっていなかった
 - .NET 9 から既定で有効に
- DATAS のお仕事は「アプリケーションのサイズに適応する」事
 - ヒープ数の調整
 - スループットと消費するメモリと GC の頻度に応じて調整
 - 完全な compaction を走らせるための allocation budget の調整

完全な compaction を走らせないと
何時まで経っても余計なメモリを食いつぱなし

VM

FCALL から QCALL へ

- managed code から runtime code の呼び出し手段
 - QCALL
 - 実質 runtime で定義された関数への P/Invoke
 - GC を止めない
 - FCALL
 - より特殊で複雑なメカニズムを持つ呼び出し手段
 - QCALL に比較すると重い
 - GC を止める
- 以前は FCALL が主流だったが、リリース毎に徐々に QCALL に置き換えられている

VM

例外の高速化

- Native AOT の例外ハンドリングの実装を coreclr に移植
 - 例外処理が 3.5 ~ 4 倍高速化(自称)
 - 実際には 2~4倍くらい?
- global spinlock の排除

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

BenchmarkSwitcher.FromAssembly(typeof(Tests).Assembly).Run(args);

[MemoryDiagnoser(false)]
[HideColumns("Job", "Error", "StdDev", "Median", "RatioSD")]
public class Tests
{
    [Benchmark]
    public async Task ExceptionThrowCatch()
    {
        for (int i = 0; i < 1000; i++)
        {
            try { await Recur(10); } catch { }
        }
    }

    private async Task Recur(int depth)
    {
        if (depth <= 0)
        {
            await Task.Yield();
            throw new Exception();
        }

        await Recur(depth - 1);
    }
}
```

Method	Runtime	Mean	Ratio
ExceptionThrowCatch	.NET 8.0	123.03 ms	1.00
ExceptionThrowCatch	.NET 9.0	54.68 ms	0.44

Threading

System.Threading.Lock

- .NET 8 までは object で lock するのが常套手段
- .NET 9 からは System.Threading.Lock の利用を推奨

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

BenchmarkSwitcher.FromAssembly(typeof(Tests).Assembly).Run(args);

[HideColumns("Job", "Error", "StdDev", "Median", "RatioSD")]
public class Tests
{
    private readonly object _monitor = new();
    private readonly Lock _lock = new();
    private int _value;

    [Benchmark]
    public void WithMonitor()
    {
        lock (_monitor)
        {
            _value++;
        }
    }

    [Benchmark]
    public void WithLock()
    {
        lock (_lock)
        {
            _value++;
        }
    }
}
```

Method	Mean
WithMonitor	14.30 ns
WithLock	13.86 ns

Threading

```
public class MyClass
{
    private readonly object _lock = new();

    public void M()
    {
        lock(_lock)
        {
        }
    }
}
```

```
public class MyClass
{
    [Nullable(1)]
    private readonly object _lock = new object();

    public void M()
    {
        object @lock = _lock;
        bool lockTaken = false;
        try
        {
            Monitor.Enter(@lock, ref lockTaken);
        }
        finally
        {
            if (lockTaken)
            {
                Monitor.Exit(@lock);
            }
        }
    }
}
```

```
public class MyClass
{
    private readonly Lock _lock = new();

    public void M()
    {
        lock(_lock)
        {
        }
    }
}
```

C# 13 が Lock を特別扱い

```
public class MyClass
{
    [Nullable(1)]
    private readonly Lock _lock = new Lock();

    public void M()
    {
        Lock.Scope scope = _lock.EnterScope();
        try
        {
        }
        finally
        {
            scope.Dispose();
        }
    }
}
```

Threading

System.Threading.Interlocked

- Exchange / CompareExchange (に overload が追加
 - byte
 - sbyte
 - ushort
 - short

Threading

System.Threading.Interlocked

- Exchange / CompareExchange (に overload が追加
 - byte
 - sbyte
 - ushort
 - short

Parallel.ForAsync<T> (.NET 8~) 内部でも
利用され、パフォーマンス改善

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

BenchmarkSwitcher.FromAssembly(typeof(Tests).Assembly).Run(args);

[HideColumns("Job", "Error", "StdDev", "Median", "RatioSD")]
public class Tests
{
    [Benchmark]
    public async Task ParallelForAsync()
    {
        await Parallel.ForAsync('¥0', '¥uFFFF', async (c, _) =>
        {
        });
    }
}
```

Method	Runtime	Mean	Ratio
ParallelForAsync	.NET 8.0	42.807 ms	1.00
ParallelForAsync	.NET 9.0	7.184 ms	0.17

Threading

System.Threading.Interlocked

- Exchange / CompareExchange (に overload が追加
 - byte
 - sbyte
 - ushort
 - short

Parallel.ForAsync<T> (.NET 8~) 内部でも
利用され、パフォーマンス改善

- Exchange<T> / CompareExchange<T> の class 制約の排除
 - generics で primitive type が使えるようになったため、使い勝手向上
 - enum に対応

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

BenchmarkSwitcher.FromAssembly(typeof(Tests).Assembly).Run(args);

[HideColumns("Job", "Error", "StdDev", "Median", "RatioSD")]
public class Tests
{
    [Benchmark]
    public async Task ParallelForAsync()
    {
        await Parallel.ForAsync('¥0', '¥uFFFF', async (c, _) =>
        {
        });
    }
}
```

Method	Runtime	Mean	Ratio
ParallelForAsync	.NET 8.0	42.807 ms	1.00
ParallelForAsync	.NET 9.0	7.184 ms	0.17

Threading

System.Threading.Interlocked

- Exchange / CompareExchange (に overload が追加

- byte
- sbyte
- ushort
- short

Parallel.ForAsync<T> (.NET 8~) 内部でも
利用され、パフォーマンス改善

- Exchange<T> / CompareExchange<T> の class 制約の排除

- generics で primitive type が使えるようになったため、使い勝手向上
- enum に対応

CompareExchange の都合上 int (4 byte) にしていたところを .NET 9 からは
bool (1 byte) に置き換えられるため、オブジェクトのサイズを落とせる

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

BenchmarkSwitcher.FromAssembly(typeof(Tests).Assembly).Run(args);

[HideColumns("Job", "Error", "StdDev", "Median", "RatioSD")]
public class Tests
{
    [Benchmark]
    public async Task ParallelForAsync()
    {
        await Parallel.ForAsync('¥0', '¥uFFFF', async (c, _) =>
        {
        });
    }
}
```

Method	Runtime	Mean	Ratio
ParallelForAsync	.NET 8.0	42.807 ms	1.00
ParallelForAsync	.NET 9.0	7.184 ms	0.17

Threading

Task.WhenEach

- 完了した task を順次返してくれる新しい API

WhenEach が無い時代は
こんな事をする必要があった。O(N^2)

```
List<Task> tasks = new() { t1, t2, t3 };  
while (tasks.Count > 0)  
{  
    Task completed = await Task.WhenAny(tasks);  
    Handle(completed);  
    tasks.Remove(completed);  
}
```

.NET 9

```
await foreach (Task completed in Task.WhenEach([t1, t2, t3]))  
{  
    Debug.Assert(completed.IsCompleted);  
    Handle(completed);  
}
```

```
public class Tests  
{  
    [Params(10, 1_000)]  
    public int Count { get; set; }  
  
    [Benchmark]  
    public async Task WithWhenAny()  
    {  
        var tcs = Enumerable.Range(0, Count).Select(_ => new TaskCompletionSource()).ToList();  
  
        List<Task> tasks = tcs.Select(t => t.Task).ToList();  
        tcs[^1].SetResult();  
        while (tasks.Count > 0)  
        {  
            Task completed = await Task.WhenAny(tasks);  
            tasks.Remove(completed);  
            tcs.RemoveAt(tcs.Count - 1);  
  
            if (tasks.Count == 0) break;  
            tcs[^1].SetResult();  
        }  
    }  
  
    [Benchmark]  
    public async Task WithWhenEach()  
    {  
        var tcs = Enumerable.Range(0, Count).Select(_ => new TaskCompletionSource()).ToList();  
  
        int remaining = tcs.Count - 1;  
        tcs[remaining].SetResult();  
        await foreach (Task completed in Task.WhenEach(tcs.Select(t => t.Task)))  
        {  
            if (remaining == 0) break;  
            tcs[--remaining].SetResult();  
        }  
    }  
}
```

Method	Count	Mean	Allocated
WithWhenAny	10	3.232 us	3.47 KB
WithWhenEach	10	1.223 us	1.43 KB
WithWhenAny	1000	20,082.683 us	4207.12 KB
WithWhenEach	1000	102.759 us	94.24 KB

Reflection

同種の MethodInvoker も .NET 8 で導入

ConstructorInvoker (.NET 8)

- ConstructorInvoker 自体は .NET 8 で導入された型
- .NET 9 から Ms.Ex.DependencyInjection 内部で利用されるようになった

```
public class Tests
{
    private IServiceProvider _serviceProvider = new ServiceCollection().BuildServiceProvider();

    [Benchmark]
    public MyClass Create() => ActivatorUtilities.CreateInstance<MyClass>(_serviceProvider, 1, 2, 3);

    public class MyClass
    {
        public MyClass() { }
        public MyClass(int a) { }
        public MyClass(int a, int b) { }
        [ActivatorUtilitiesConstructor]
        public MyClass(int a, int b, int c) { }
    }
}
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
Create	.NET 8.0	163.60 ns	1.00	288 B	1.00
Create	.NET 9.0	83.46 ns	0.51	144 B	0.50

SearchValues<string>

.NET 9 から SearchValues<T> が string で使えるように...!

- .NET 8 で導入された SearchValues<T>
 - .NET 8 時点では byte と char のみの対応だった

```
SearchValues<string> daysOfWeek = SearchValues.Create(  
    [ "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" ],  
    StringComparison.OrdinalIgnoreCase  
);
```

```
ReadOnlySpan<char> textToSearch = "long long text monday".AsSpan();  
int i = textToSearch.IndexOfAny(daysOfWeek);
```

SearchValues<string>

SearchValues<string> を使わない書き方とそのパフォーマンス

```
public class Tests1
{
    private static readonly string Input = new HttpClient().GetStringAsync("https://gute
    private static readonly string[] DaysOfWeek = ["Monday", "Tuesday", "Wednesday", "Th

    public bool Contains_Iterate()
    {
        ReadOnlySpan<char> input = Input.AsSpan();

        for (int i = 0; i < input.Length; i++)
        {
            foreach (string dow in DaysOfWeek)
            {
                if (input.Slice(i).StartsWith(dow, StringComparison.OrdinalIgnoreCase))
                {
                    return true;
                }
            }
        }

        return false;
    }
}
```

```
public class Tests3
{
    private static readonly string Input = new HttpClient().GetStringAsync
    private static readonly string[] DaysOfWeek = ["Monday", "Tuesday", "W

    public bool Contains_ContainsEachNeedle()
    {
        ReadOnlySpan<char> input = Input;

        foreach (string dow in DaysOfWeek)
        {
            if (input.Contains(dow, StringComparison.OrdinalIgnoreCase))
            {
                return true;
            }
        }

        return false;
    }
}
```

```
public class Tests2
{
    private static readonly string Input = new HttpClient().GetStringAsync("https://gutenber
    public bool Contains_Iterate_Switch()
    {
        ReadOnlySpan<char> input = Input;

        for (int i = 0; i < input.Length; i++)
        {
            ReadOnlySpan<char> slice = input.Slice(i);
            switch ((char)input[i] | 0x20)
            {
                case 's' when slice.StartsWith("Sunday", StringComparison.OrdinalIgnoreCase) || slice.Sta
                case 'm' when slice.StartsWith("Monday", StringComparison.OrdinalIgnoreCase):
                case 't' when slice.StartsWith("Tuesday", StringComparison.OrdinalIgnoreCase) || slice.Sta
                case 'w' when slice.StartsWith("Wednesday", StringComparison.OrdinalIgnoreCase):
                case 'f' when slice.StartsWith("Friday", StringComparison.OrdinalIgnoreCase):
                return true;
            }
        }

        return false;
    }
}
```

```
public class Tests4
{
    private static readonly string Input = new HttpClient().GetStringAsync("https://gutenber
    private static readonly SearchValues<char> DaysOfWeek = SearchValues.Create(['S', 's', 'M', 'm',

    public bool Contains_IndexOfAnyFirstChars_SearchValues()
    {
        ReadOnlySpan<char> input = Input;

        int i;
        while ((i = input.IndexOfAny(DaysOfWeek)) >= 0)
        {
            ReadOnlySpan<char> slice = input.Slice(i);
            switch ((char)input[i] | 0x20)
            {
                case 's' when slice.StartsWith("Sunday", StringComparison.OrdinalIgnoreCase) || slic
                case 'm' when slice.StartsWith("Monday", StringComparison.OrdinalIgnoreCase):
                case 't' when slice.StartsWith("Tuesday", StringComparison.OrdinalIgnoreCase) || sli
                case 'w' when slice.StartsWith("Wednesday", StringComparison.OrdinalIgnoreCase):
                case 'f' when slice.StartsWith("Friday", StringComparison.OrdinalIgnoreCase):
                return true;
            }

            input = input.Slice(i + 1);
        }

        return false;
    }
}
```

SearchValues<string>

SearchValues<string> を使わない書き方とそのパフォーマンス

```
public class Tests1
{
    private static readonly string Input = new HttpClient().GetStringAsync("https://gute
    private static readonly string[] DaysOfWeek = ["Monday", "Tuesday", "Wednesday", "Th

    public bool Contains_Iterate()
    {
        ReadOnlySpan<char> input = Input.AsSpan();

        for (int i = 0; i < input.Length; i++)
        {
            foreach (string dow in DaysOfWeek)
            {
                if (input.Slice(i).StartsWith(dow, StringComparison.OrdinalIgnoreCase))
                {
                    return true;
                }
            }
        }

        return false;
    }
}

public class Tests3
{
    private static readonly string Input = new HttpClient().GetStringAsync("https://gute
    private static readonly string[] DaysOfWeek = ["Monday", "Tuesday", "Wednesday", "Th

    public bool Contains_ContainsEachNeedle()
    {
        ReadOnlySpan<char> input = Input.AsSpan();

        foreach (string dow in DaysOfWeek)
        {
            if (input.Contains(dow, StringComparison.OrdinalIgnoreCase))
            {
                return true;
            }
        }

        return false;
    }
}
```

```
public class Tests2
{
    private static readonly string Input = new HttpClient().GetStringAsync("https://gutenber
    public bool Contains_Iterate_Switch()
    {
        ReadOnlySpan<char> input = Input.AsSpan();

        for (int i = 0; i < input.Length; i++)
        {
            ReadOnlySpan<char> slice = input.Slice(i);
            switch ((char)input[i] | 0x20)
            {
                case 's' when slice.StartsWith("Sunday", StringComparison.OrdinalIgnoreCase) || slice.Sta
                case 'm' when slice.StartsWith("Monday", StringComparison.OrdinalIgnoreCase):
                case 't' when slice.StartsWith("Tuesday", StringComparison.OrdinalIgnoreCase) || slice.Sta
                case 'w' when slice.StartsWith("Wednesday", StringComparison.OrdinalIgnoreCase):
                case 'f' when slice.StartsWith("Friday", StringComparison.OrdinalIgnoreCase):
                return true;
            }
        }

        return false;
    }
}
```

Method	Mean	Ratio
Contains_Iterate	227.526 us	1.000
Contains_Iterate_Switch	13.885 us	0.061
Contains_ContainsEachNeedle	302.330 us	1.329
Contains_IndexOfAnyFirstChars_SearchValues	7.151 us	0.031

```
Client().GetStringAsync("https://gutenberg.org/cache/epub/24000/24000-h/24000-h.htm");
DaysOfWeek = SearchValues.Create(['S', 's', 'M', 'm', 'T', 't', 'W', 'w', 'F', 'f']);

Values()

)

);

ay", StringComparison.OrdinalIgnoreCase) || slice
ay", StringComparison.OrdinalIgnoreCase):
day", StringComparison.OrdinalIgnoreCase) || sli
uesday", StringComparison.OrdinalIgnoreCase):
ay", StringComparison.OrdinalIgnoreCase):
```

```
return false;
}
```

SearchValues<string>

SearchValue

```
public class Tests1
{
    ... private static readonly st
    ... private static readonly st

    ... public bool Contains_Itera
    ... {
    ...     ... ReadOnlySpan<char> inp
    ...     ... for (int i = 0; i < in
    ...     ... {
    ...         ... foreach (string do
    ...         ... {
    ...             ... if (input.Slic
    ...             ... {
    ...                 ... return tru
    ...             ... }
    ...         ... }
    ...     ... }
    ... }
    ... public class Tests3
    ... {
    ...     ... ret
    ...     ... }
    ... }
}
```

```
public class Tests5
{
    ... private static readonly string Input = new HttpClient().GetStringAsync("https://gute
    ... private static readonly SearchValues<string> DaysOfWeek = SearchValues.Create(
    ...     ... ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"],
    ...     ... StringComparison.OrdinalIgnoreCase
    ... );
    ... public bool Contains_StringSearchValues()
    ... {
    ...     ... return Input.AsSpan().ContainsAny(DaysOfWeek);
    ... }
}
```

Method	Mean	Ratio
Contains_Iterate	227.526 us	1.000
Contains_Iterate_Switch	13.885 us	0.061
Contains_ContainsEachNeedle	302.330 us	1.329
Contains_IndexOfAnyFirstChars_SearchValues	7.151 us	0.031

```
... public bool Contains_ContainsE
... {
...     ... ReadOnlySpan<char> input =
...     ... foreach (string dow in Day
...     ... {
...         ... if (input.Contains(dow
...         ... {
...             ... return true;
...         ... }
...     ... }
... }
... return false;
... }
```

```
... return false;
... }
```

```
... client().GetStringAsync("https://gutenberg.org/ca
... fWeek = SearchValues.Create(['S', 's', 'M', 'm',
... Values()
... )
... );
... day", StringComparison.OrdinalIgnoreCase) || slic
... day", StringComparison.OrdinalIgnoreCase):
... day", StringComparison.OrdinalIgnoreCase) || sli
... esday", StringComparison.OrdinalIgnoreCase):
... day", StringComparison.OrdinalIgnoreCase):
```

SearchValues<string>

SearchValue

```
public class Tests1
{
    ... private static readonly st
    ... private static readonly st

    ... public bool Contains_Itera
    ... {
    ...     ... ReadOnlySpan<char> inp
    ...     ... for (int i = 0; i < in
    ...     ... {
    ...         ... foreach (string do
    ...         ... {
    ...             ... if (input.Slic
    ...             ... {
    ...                 ... return tru
    ...             ... }
    ...         ... }
    ...     ... }
    ... }
    ... public class Tests3
    ... {
    ...     ... private static readonly string
    ...     ... private static readonly string
    ...     ... public bool Contains_ContainsE
    ...     ... {
    ...         ...     ... ReadOnlySpan<char> input =
    ...         ...     ... foreach (string dow in Day
    ...         ...     ... {
    ...         ...         ... if (input.Contains(dow
    ...         ...         ... {
    ...         ...             ... return true;
    ...         ...         ... }
    ...         ...     ... }
    ...         ...     ... }
    ...         ...     ... return false;
    ...         ...     ... }
    ...     ... }
    ... }
```

```
public class Tests5
{
    ... private static readonly string Input = new HttpClient().GetStringAsync("https://gute
    ... private static readonly SearchValues<string> DaysOfWeek = SearchValues.Create(
    ...     ... ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"],
    ...     ... StringComparison.OrdinalIgnoreCase
    ... );
    ... public bool Contains_StringSearchValues()
    ... {
    ...     ... return Input.AsSpan().ContainsAny(DaysOfWeek);
    ... }
}
```

Method	Mean	Ratio
Contains_Iterate	227.526 us	1.000
Contains_Iterate_Switch	13.885 us	0.061
Contains_ContainsEachNeedle	302.330 us	1.329
Contains_IndexOfAnyFirstChars_SearchValues	7.151 us	0.031
Contains_StringSearchValues	2.153 us	0.009

```
... org/cache/ej
... slice.Sta
... slice.Sta
... ):
... Client().GetStringAsync("https://gutenberg.org/ca
... fWeek = SearchValues.Create(['S', 's', 'M', 'm',
... Values()
... )
... );
... ay", StringComparison.OrdinalIgnoreCase) || slic
... ay", StringComparison.OrdinalIgnoreCase):
... day", StringComparison.OrdinalIgnoreCase) || sli
... esday", StringComparison.OrdinalIgnoreCase):
... ay", StringComparison.OrdinalIgnoreCase):
```

SearchValues<string>

Regex の中でも使われるので
.NET 9 で再コンパイルすれば高速になる可能性あり！

SearchValue

```
public class Tests1
{
    ...private static readonly st
    ...private static readonly st

    ...public bool Contains_Itera
    ...{
    ...    ReadOnlySpan<char> inp
    ...
    ...    for (int i = 0; i < in
    ...    {
    ...        foreach (string do
    ...        {
    ...            if (input.Slic
    ...            {
    ...                return tru
    ...            }
    ...        }
    ...    }
    ...}
    ...public class Tests3
    ...ret
    ...{
    ...private static readonly string
    ...private static readonly string

    ...public bool Contains_ContainsE
    ...{
    ...    ReadOnlySpan<char> input =
    ...
    ...    foreach (string dow in Day
    ...    {
    ...        if (input.Contains(dow
    ...        {
    ...            return true;
    ...        }
    ...    }
    ...
    ...    return false;
    ...}
    ...}
```

```
public class Tests5
{
    ...private static readonly string Input = new HttpClient().GetStringAsync("https://gute
    ...private static readonly SearchValues<string> DaysOfWeek = SearchValues.Create(
    ...    ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"],
    ...    StringComparison.OrdinalIgnoreCase
    ...);

    ...public bool Contains_StringSearchValues()
    ...{
    ...    return Input.AsSpan().ContainsAny(DaysOfWeek);
    ...}
}
```

Method	Mean	Ratio
Contains_Iterate	227.526 us	1.000
Contains_Iterate_Switch	13.885 us	0.061
Contains_ContainsEachNeedle	302.330 us	1.329
Contains_IndexOfAnyFirstChars_SearchValues	7.151 us	0.031
Contains_StringSearchValues	2.153 us	0.009

```
...org/cache/ej
... slice.Sta
... slice.Sta
...):
...
...Client().GetStringAsync("https://gutenberg.org/ca
...fWeek = SearchValues.Create(['S', 's', 'M', 'm',
...Values()
...
...0)
...);
...
...ay", StringComparison.OrdinalIgnoreCase) || slic
...ay", StringComparison.OrdinalIgnoreCase):
...day", StringComparison.OrdinalIgnoreCase) || slic
...esday", StringComparison.OrdinalIgnoreCase):
...ay", StringComparison.OrdinalIgnoreCase):
```

Span

ReadOnlySpan<T> を引数に持つメソッドへの params アノテーション祭り

- C# 13 の新機能である params collection にあらゆるところで対応
 - 再コンパイルするだけで高速化の可能性...!
 - params T[] が呼ばれていた箇所が params ReadOnlySpan<T> が利用されるようになるため
 - 実際に params が追加された API 群 (一部)
 - StringBuilder
 - Path / StreamWriter / TextWriter
 - Task / CancellationTokenSource
 - ImmutableArray / ImmutableHashSet / ImmutableList / ImmutableQueue / ImmutableStack
 - Counter / UpDownCounter / Histogram / Measurement / TagList

Span

ReadOnlySpan の初期化の改善

- .NET 8 時点で byte / sbyte / bool の定数の ReadOnlySpan<T> を作成する場合は高速
 - アセンブリのデータを ReadOnlySpan で読むだけなので非常に高速

```
class C
{
    public ReadOnlySpan<byte> Bytes => new byte[] { 1, 2, 3, 4 };
}
```



```
internal class C
{
    public unsafe ReadOnlySpan<byte> Bytes
    {
        get
        {
            return new ReadOnlySpan<byte>(Unsafe.AsPointer(ref <PrivateImplementationDetails>.9F64A747E1B97F131FABB6B447296C9B6F0201E79FB3C5356E6C77E89B6A806A), 4);
        }
    }
}

[CompilerGenerated]
internal sealed class <PrivateImplementationDetails>
{
    internal static readonly int 9F64A747E1B97F131FABB6B447296C9B6F0201E79FB3C5356E6C77E89B6A806A/* Not supported: data(01 02 03 04) */;
}
```

Span

ReadOnlySpan の初期化の改善

- .NET 8 時点で byte / sbyte / bool の定数の ReadOnlySpan<T> を作成する場合は高速
 - アセンブリのデータを ReadOnlySpan で読むだけなので非常に高速

```
class C
{
    public ReadOnlySpan<byte> Bytes => new byte[] { 1, 2, 3, 4 };
}
```



コレクションリテラル導入以前から
存在する最適化

```
internal class C
{
    public unsafe ReadOnlySpan<byte> Bytes
    {
        get
        {
            return new ReadOnlySpan<byte>(Unsafe.AsPointer(ref <PrivateImplementationDetails>.9F64A747E1B97F131FABB6B447296C9B6F0201E79FB3C5356E6C77E89B6A806A), 4);
        }
    }
}
```

```
[CompilerGenerated]
internal sealed class <PrivateImplementationDetails>
{
    internal static readonly int 9F64A747E1B97F131FABB6B447296C9B6F0201E79FB3C5356E6C77E89B6A806A/* Not supported: data(01 02 03 04) */;
}
```

Span

エンディアンの問題で
1 byte の primitive type に限られていた

ReadOnlySpan の初期化の改善

- .NET 8 時点で byte / sbyte / bool の定数の ReadOnlySpan<T> を作成する場合は高速
 - アセンブリのデータを ReadOnlySpan で読むだけなので非常に高速

```
class C
{
    public ReadOnlySpan<byte> Bytes => new byte[] { 1, 2, 3, 4 };
}
```

↓

```
internal class C
{
    public unsafe ReadOnlySpan<byte> Bytes
    {
        get
        {
            return new ReadOnlySpan<byte>(Unsafe.AsPointer(ref <PrivateImplementationDetails>.9F64A747E1B97F131FABB6B447296C9B6F0201E79FB3C5356E6C77E89B6A806A), 4);
        }
    }
}

[CompilerGenerated]
internal sealed class <PrivateImplementationDetails>
{
    internal static readonly int 9F64A747E1B97F131FABB6B447296C9B6F0201E79FB3C5356E6C77E89B6A806A/* Not supported: data(01 02 03 04) */;
}
```

コレクションリテラル導入以前から
存在する最適化

Span

```
[Intrinsic]
public static unsafe ReadOnlySpan<T> CreateSpan<T>(RuntimeFieldHandle fldHandle)
=> new ReadOnlySpan<T>(ref Unsafe.As<byte, T>(ref GetSpanDataFrom(fldHandle, typeof(T).TypeHandle, out int length)), length);
```

ReadOnlySpan の初期化の改善

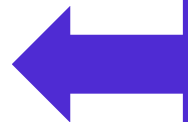
- .NET 9 ではすべての primitive type に同種の最適化が適用され高速に
 - RuntimeHelpers.CreateSpan を用いる事で解決される運びとなった

```
internal class C
{
    public ReadOnlySpan<int> Values
    {
        get
        {
            return RuntimeHelpers.CreateSpan<int>((RuntimeFieldHandle)*OpCode not supported: LdMemberToken*);
        }
    }
}
```

```
[CompilerGenerated]
internal sealed class <PrivateImplementationDetails>
{
    [StructLayout(LayoutKind.Explicit, Pack = 4, Size = 16)]
    internal struct __StaticArrayInitTypeSize=16_Align=4
    {
    }

    internal static readonly __StaticArrayInitTypeSize=16_Align=4 CF97ADEEDB59E05BFD73A2B4C2A8885708C4F4F70C84C64B27120E72AB733B724/* Not supported: data(01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00) */;
}
```

```
class C
{
    public ReadOnlySpan<int> Values => [1, 2, 3, 4];
}
```



```
// Methods
.method public hidebysig specialname
instance valuetype [System.Runtime]System.ReadOnlySpan`1<int32> get_Values () cil managed
{
    // Method begins at RVA 0x2050
    // Code size 11 (0xb)
    .maxstack 8

    IL_0000: ldtoken field valuetype '<PrivateImplementationDetails>'/ '__StaticArrayInitTypeSize=16_Align=4' '<PrivateImplementationDetails>':CF97ADEEDB59E05BFD73A2B4C2A8885708C4F4F70C84C64B27120E72AB733B724
    IL_0005: call valuetype [System.Runtime]System.ReadOnlySpan`1<int32> [System.Runtime]System.Runtime.CompilerServices.RuntimeHelpers::CreateSpan<int32>(valuetype [System.Runtime]System.RuntimeFieldHandle)
    IL_000a: ret
} // end of method C::get_Values
```

過去の資料

C# 13 / .NET 9 の新機能について語っています

C# 13 / .NET 9 の新機能 (RC 1 時点)

.NET ラボ 2024/09/28

何縫ねの。

References

- <https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-9/>
- <https://maoni0.medium.com/dynamically-adapting-to-application-sizes-2d72fcb6f1ea>
- <https://learn.microsoft.com/en-us/archive/msdn-magazine/2005/may/net-framework-internals-how-the-clr-creates-runtime-objects>
- <https://github.com/dotnet/runtime>
- <https://github.com/Maoni0/mem-doc>
- <https://speakerdeck.com/nenonaninu/dot-net-8-deji-ding-deyou-xiao-ninatuta-dynamic-pgo-nituite>