Span<T>

… and his little sister Memory<T>

Krzysztof Cwalina, Microsoft
11/29/2016

Why?

We need to provide building blocks for efficient, safe, and convenient buffer management in data transformation pipelines

Scalable data transformation APIs (e.g. parsing, formatting, compression, etc.) must never allocate buffers; the caller if these APIs wants to be in complete control over how, where, and when the buffers are allocated, and the caller will pass the buffers into these APIs.

Therefore, we (providers of these APIs) need a representation of a buffer of arbitrary memory: native, managed, stack, pooled … and it needs to be fast!

Span<T>

- Array-like type representing arbitrary memory: managed or native heap, or stack
- T* like performance; T[] like safety (or close to)
- Deterministic lifetime, very useful for buffer pooling
- Non-allocating slicing

```
public struct Span<T> {
    public Span(T[] array)
    public Span(T[] array, int index)
    public Span(T[] array, int index, int length)
    public unsafe Span(void* memory, int length)

    public static implicit operator Span<T> (ArraySegment<T> arraySegment);
    public static implicit operator Span<T> (T[] array);

    public int Length { get; } public ref T this[int index] { get; }

    public Span<T> Slice(int index);
    public Span<T> Slice(int index, int length);

    public bool TryCopyTo(T[] destination);
    public bool TryCopyTo(Span<T> destination);
    public T[] ToArray();
}
```

```
// managed memory
var arrayMemory = new byte[100];
var arraySpan = new Span<byte>(arrayMemory);
SafeSum(arraySpan);

// native memory
var nativeMemory = Marshal.AllocHGlobal(100);
Span<byte> nativeSpan;
unsafe {
    nativeSpan = new Span<byte>(nativeMemory.ToPointer(), 100);
}
SafeSum(nativeSpan);
Marshal.FreeHGlobal(nativeMemory);

// stack memory Span<byte> stackSpan;
unsafe {
    byte* stackMemory = stackalloc byte[100];
    stackSpan = new Span<byte>(stackMemory, 100);
}
SafeSum(stackSpan);
```

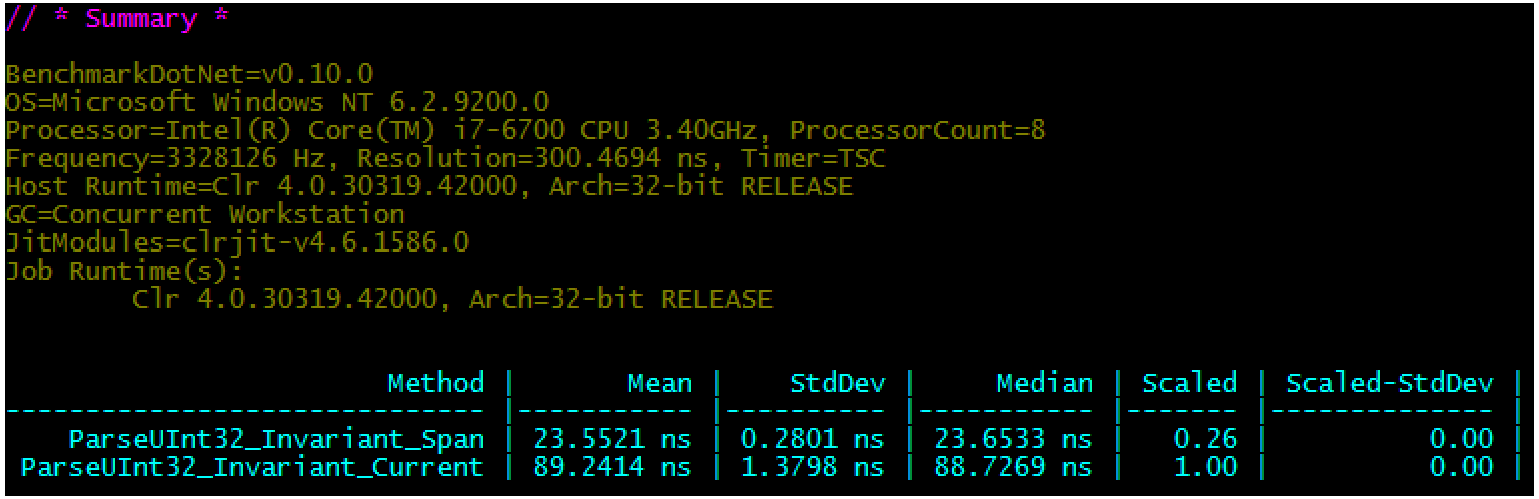
Data Transformation APIs

- Today, APIs need to chose between performance, safety, convenience

```
[Benchmark]
public void ParseUInt32_Invariant_Span() {
    uint value;
    PrimitiveParser.TryParseUInt32(uint_MaxSpan, out value);
}

[Benchmark(Baseline = true)]
public void ParseUInt32_Invariant_Current() {
    uint value;
    uint.TryParse(uint_MaxUtfStr, NumberStyles.None,
                  CultureInfo.InvariantCulture, out value);
}
```

```
public static class PrimitiveParser {
    public static bool TryParseInt16(ReadOnlySpan<byte> text, out short value);
    public static bool TryParseInt32(ReadOnlySpan<byte> text, out int value);
    public static bool TryParseInt64(ReadOnlySpan<byte> text, out long value);
    public static bool TryParseUInt16(ReadOnlySpan<byte> text, out ushort value);
    public static bool TryParseUInt32(ReadOnlySpan<byte> text, out uint value);
    public static bool TryParseUInt64(ReadOnlySpan<byte> text, out ulong value);
    …
}
```



```
// * Summary *

BenchmarkDotNet=v0.10.0
OS=Microsoft Windows NT 6.2.9200.0
Processor=Intel(R) Core(TM) i7-6700 CPU 3.40GHz, ProcessorCount=8
Frequency=3328126 Hz, Resolution=300.4694 ns, Timer=TSC
Host Runtime=Clr 4.0.30319.42000, Arch=32-bit RELEASE
GC=Concurrent Workstation
JitModules=clrjit-v4.6.1586.0
Job Runtime(s):
        Clr 4.0.30319.42000, Arch=32-bit RELEASE


                        Method |       Mean |     StdDev |      Median | Scaled | Scaled-StdDev |
------------------------------ |----------- |----------- |------------ |------- |-------------- |
    ParseUInt32_Invariant_Span | 23.5521 ns | 0.2801 ns | 23.6533 ns |   0.26 |          0.00 |
 ParseUInt32_Invariant_Current | 89.2414 ns | 1.3798 ns | 88.7269 ns |   1.00 |          0.00 |
```

Span makes the APIs fast, safe, and convenient

Same for other data transformation APIs …

- Formatting
- Base64 encoding
- Unicode encoding
- URI, HTML, JavaScript escaping
- HTTP parsing/writing
- URI parsing/writing
- Compression/Decompression
- Cryptography, TSL
- XML parsing/writing
- JSON parsing/writing
- JSON serialization
- Substring
- Binary reading/writing
- … see more at https://github.com/dotnet/corefxlab/blob/master/docs/specs/span.md#scenarios

Q: How come Span<T> is sooooooo great?

A:

```
public struct Span<T> {
    internal ref T _pointer;
    internal int _length;
}
```

Q: So what’s the catch?

A: Did you not notice the funny “ref” thingy?

Q: Will it hurt?

A: Hey, look! Here is a bandaid you might need.

Memory<T>

- So, Span<T> is stack-only; we provide Memory<T> as a ~~bandaid~~ complement
- Memory<T> is a heap friendly Span<T> factory
- Memory<T> is slower than Span<T>/T*/T[]
- Logically Memory<T> is:

```
public struct Memory<T> {
    void* _ptr;
    T[] _array;
    int _offset;
    int _length;

    public Span<T> Span => _ptr == null ? new Span<T>(_array, _offset, _length) : new Span<T>(_ptr, _length);
}
```

OwnedMemory<T>

Memory<T> is …

- a struct (so we can slice it), i.e. we cannot prevent copies being made
- heap friendly, i.e. can be stored “for later”, e.g. in a static

Q: Did we just lose the ability to safely pool memory buffers?

A: OwnedMemory<T>

```
public class OwnedMemory<T> {
    void* _ptr;
    T[] _array;
    int _offset;
    int _length;

    public Span<T> Span => _ptr == null ? new Span<T>(_array, _offset, _length) : new Span<T>(_ptr, _length);

    public void Dispose() { _ptr = null; _array = null; }
}
public struct Memory<T> {
    OwnedMemory<T> _owned;
    public Memory(OwnedMemory<T> owned) { ... }
    public Span<T> Span => _owned.Span;
}
```

Memory<T> -> Span<T> Safety

- OwnedMemory<T>.Span will fail if OwnedMemory<T> is disposed
- But, what if Span<T> is already on the stack when its memory is disposed?

```
var owned = new OwnedNativeMemory(1024);
var memory = owned.Memory;
var span1 = memory.Span; // of course works as it should

Task.Run(()=>{
    var span2 = memory.Span; // the following line is unsafe, if the Dispose call (below) executes at this point.
    span2[0] = 0;
});

owned.Dispose();

var span3 = memory.Span; // fails as it should because the memory instance is now pointing to Disposed OwnedMemory\<T\>.
```

- We can fix it with reference counting, but it’s slow
- We can fix it with optional reference counting, but it’s not 100% safe
- We are currently/still working on making this tradeoff (based on performance and higher level programming model)

Details

- Span<T> design document: https://github.com/dotnet/corefxlab/blob/master/docs/specs/span.md
- Memory<T> design document: https://github.com/dotnet/corefxlab/blob/master/docs/specs/memory.md
- Memory<T> sources:
- https://github.com/dotnet/corefxlab/blob/master/src/System.Slices/System/Buffers/Memory.cs
- Span<T> sources:
  - Fast: https://github.com/dotnet/coreclr/blob/master/src/mscorlib/src/System/Span.cs
  - Slow: https://github.com/dotnet/corefx/blob/master/src/System.Memory/src/System/Span.cs
- Package with Span<T>:
- https://dotnet.myget.org/feed/dotnet-core/package/nuget/System.Memory
- Package with Memory<T>:
- https://dotnet.myget.org/feed/dotnet-corefxlab/package/nuget/System.Slices

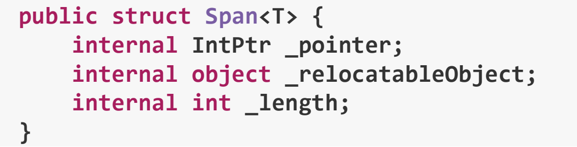
Backup

Other Topics

- OwnedMemory<T> pooling
    - https://github.com/dotnet/corefxlab/blob/master/docs/specs/memory.md#pooling-ownedmemoryt
- IOwnedMemory<T>
    - https://github.com/dotnet/corefxlab/blob/master/docs/specs/memory.md#iownedmemoryt
- ReadOnlySpan<T> and ReadOnlyMemory<T>
- Fast and Slow Span<T>
    - https://github.com/dotnet/corefxlab/blob/master/docs/specs/span.md#designrepresentation

Backup : Slow Span

- Representation for existing runtimes
- Not as fast, but makes Span<T> immediately available to all runtimes



```
public struct Span<T> {
    internal IntPtr _pointer;
    internal object _relocatableObject;
    internal int _length;
}
```